Re-architecting to cloud native: an evolutionary approach to increasing developer productivity at scale

Google Cloud

Google Cloud

Table of Contents

Why move to a cloud-native architecture?

Many companies build their custom software services using monolithic architectures. This approach has advantages: monolithic systems are relatively simple to design and deploy - at least, at first. However it can become hard to maintain developer productivity and deployment velocity as applications grow more complex, leading to systems that are expensive and time-consuming to change and risky to deploy. This paper shows how to re-architect your applications to a cloud-native paradigm that allows you to accelerate delivery of new features even as you grow your teams, while also improving software quality and achieving higher levels of stability and availability.

As services—and the teams responsible for them—grow, they tend to become more complex and harder to evolve and to operate. Testing and deployment becomes more painful, adding new features becomes harder, and maintaining reliability and availability can be a struggle.

Research by Google's DORA team shows that it is possible to achieve high levels of software delivery throughput and service stability and availability across organizations of all sizes and domains. High-performing teams are able to deploy multiple times per day, get changes out to production in less than a day, restore service in less than an hour, and achieve change fail rates of 0-15%[1].

Furthermore, high performers are able to achieve higher levels of developer productivity, measured in terms of deployments per developer per day, even as they increase the size of their teams. This is shown in Figure 1.

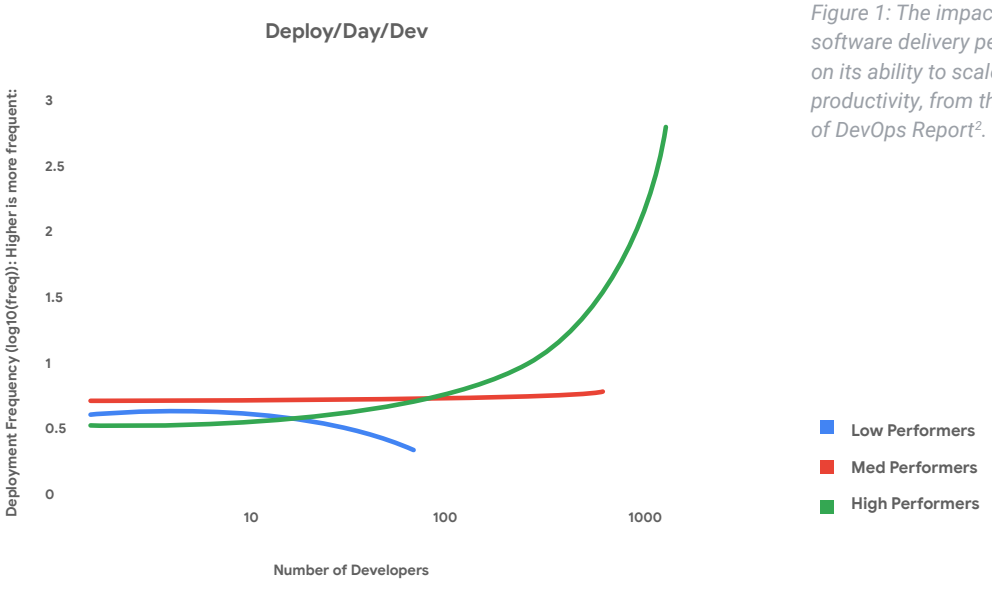

Figure 1: The impact of a team's software delivery performance on its ability to scale developer productivity, from the 2015 State of DevOps Report[2].

The rest of this paper shows how to migrate your applications to a modern cloud-native paradigm to help achieve these outcomes. By implementing the technical practices described in this paper, you can reach the following goals:

- **Increased developer productivity,** even as you increase your team sizes.
- **Faster time-to-market:** add new features and fix defects more quickly.
- **Higher availability:** increase the uptime of your software, reduce the rate of deployment failures, and reduce time-to-restore in the event of incidents.
- **Improved security:** reduce the attack surface area of your applications, and make it easier to detect and respond rapidly to attacks and newly discovered vulnerabilities.
- **Better scalability:** cloud-native platforms and applications make it easy to scale horizontally where necessary - and to scale down too.
- **Reduce costs:** a streamlined software delivery process reduces the costs of delivering new features, and effective use of cloud platforms substantially reduces the operating costs of your services.

What is a cloud-native architecture?

Monolithic applications must be built, tested, and deployed as a single unit. Often, the operating system, middleware, and language stack for the application are customized or custom-configured for each application. The build, test, and deployment scripts and processes are typically also unique to each application. This is simple and effective for greenfield applications, but as they grow, it becomes harder to change, test, deploy, and operate such systems.

Furthermore, as systems grow, so does the size and complexity of the teams that build, test, deploy, and operate the service. A common, but flawed approach, is to split teams out by function, leading to hand-offs between teams that tend to drive up lead times and batch sizes and lead to significant amounts of rework. DORA's research shows high-performing teams are twice as likely to be developing and delivering software in a single, cross-functional team.

Symptoms of this problem include:

- Long, frequently broken build processes
- Infrequent integration and test cycles
- Increased effort required to support the build and test processes
- Loss of developer productivity
- Painful deployment processes that must be performed out of hours, requiring scheduled downtime
- Significant effort in managing the configuration of test and production environments

In the cloud-native paradigm, in contrast[3]:

- Complex systems are decomposed into services that can be independently tested and deployed on a containerized runtime (a microservices or service-oriented architecture);
- Applications use standard platform-provided services, such as database management systems (DBMS), blob storage, messaging, CDN, and SSL termination;
- A standardized cloud platform takes care of many operational concerns, such as deployment, autoscaling, configuration, secrets management, monitoring, and alerting. These services can be accessed on-demand by application development teams;
- Standardized operating system, middleware, and language-specific stacks are provided to application developers, and the maintenance and patching of these stacks are done out-of-band by either the platform provider or a separate team;
- A single cross-functional team can be responsible for the entire software delivery lifecycle of each service.

This paradigm provides many benefits:

- **Faster delivery:** Since services are now small and loosely coupled, the teams associated with those services can work autonomously. This increases developer productivity and development velocity.
- **Reliable releases:** Developers can rapidly build, test, and deploy new and existing services on production-like test environments. Deployment to production is also a simple, atomic activity. This substantially speeds up the software delivery process and reduces the risk of deployments.
- **Lower costs:** The cost and complexity of test and production environments is substantially reduced because shared, standardized services are provided by the platform, and because applications run on shared physical infrastructure.
- **Better security:** Vendors are now responsible for keeping shared services, such as DBMS and messaging infrastructure up-to-date, patched, and compliant. It's also much easier to keep applications patched and up-to-date because there is a standard way to deploy and manage applications.
- **Higher availability:** Availability and reliability of applications is increased because of the reduced complexity of the operational environment, the ease of making configuration changes, and the ability to handle autoscaling and autohealing at the platform level.
- **Simpler, cheaper compliance:** Most information security controls can be implemented at the platform layer, making it significantly cheaper and easier to implement and demonstrate compliance. Many cloud providers maintain compliance with risk management frameworks such as SOC2 and FedRAMP, meaning applications deployed on top of them only have to demonstrate compliance with residual controls not implemented at the platform layer.

However, there are some trade-offs associated with the cloud-native model:

- All applications are now distributed systems, which means they make significant numbers of remote calls as part of their operation. This means thinking carefully about how to handle network failures and performance issues, and how to debug problems in production.
- Developers must use the standardised operating system, middleware, and application stacks provided by the platform. This makes local development harder.

- Architects need to adopt an event-driven approach to systems design, including embracing eventual consistency.

Migrating to cloud native

Many organizations have adopted a “lift-and-shift” approach to moving services to the cloud. In this approach, only minor changes are required to systems, and the cloud is basically treated as a traditional datacenter, albeit one that provides substantially better APIs, services, and management tooling compared with traditional datacenters. However, lift-and-shift by itself provides none of the benefits of the cloud-native paradigm described above.

Many organizations stall at lift-and-shift because of the expense and complexity of moving their applications to a cloud-native architecture, which requires rethinking everything from application architecture to production operations and indeed the entire software delivery lifecycle. This fear is rational: many large organizations have been burned by failed multi-year, “big bang” replatforming efforts.

The solution is to take an incremental, iterative, and evolutionary approach to re-architecting your systems to cloud native, enabling teams to learn how to work effectively in this new paradigm while continuing to deliver new functionality: an approach that we call “move-and-improve”.

A key pattern in evolutionary architecture is known as the **strangler fig application**[4]. Rather than completely rewriting systems from scratch, write new features in a modern, cloud-native style, but have them talk to the original monolithic application for existing functionality. Gradually shift existing functionality over time as necessary for the conceptual integrity of the new services, as shown in Figure 2.

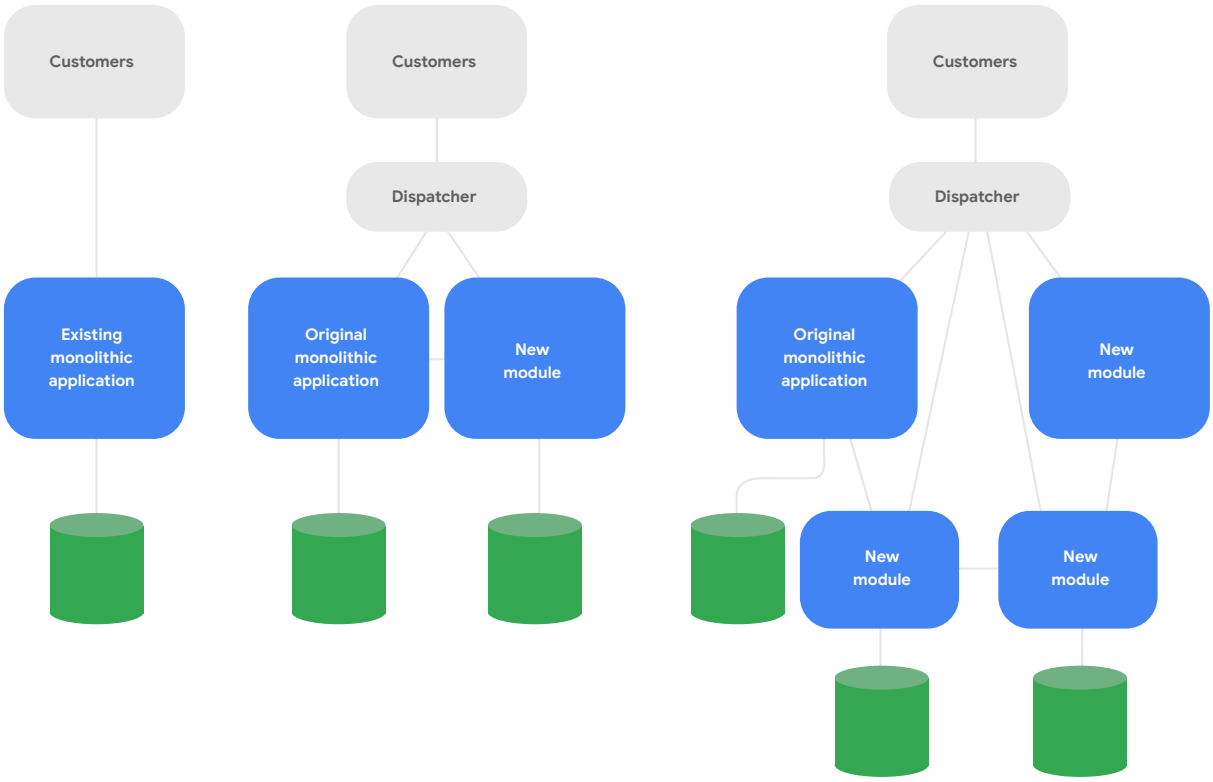

Figure 2: Using the strangler fig pattern to incrementally re-architect your monolithic application

Here are three important guidelines for successfully re-architecting:

First, **start by delivering new functionality fast** rather than reproducing existing functionality. The key metric is how quickly you can start delivering new functionality using new services, so that you can quickly learn and communicate good practice gained from actually working within this paradigm. Cut scope aggressively with the goal of delivering something to real users in weeks, not months.

Second, **design for cloud native**. This means using the cloud platform's native services for DBMS, messaging, CDN, networking, blob storage and so forth, and using standardized platform-provided application stacks wherever possible. Services should be containerized, making use of the serverless paradigm wherever possible, and the build, test, and deploy process should be fully automated. Have all applications use platform-provided shared services for logging, monitoring, and alerting. (It is worth noting that this kind of platform

architecture can be usefully deployed for any multi-tenant application platform, including a bare-metal on-premises environment.)

A high-level picture of a cloud-native platform is shown in Figure 3, below.

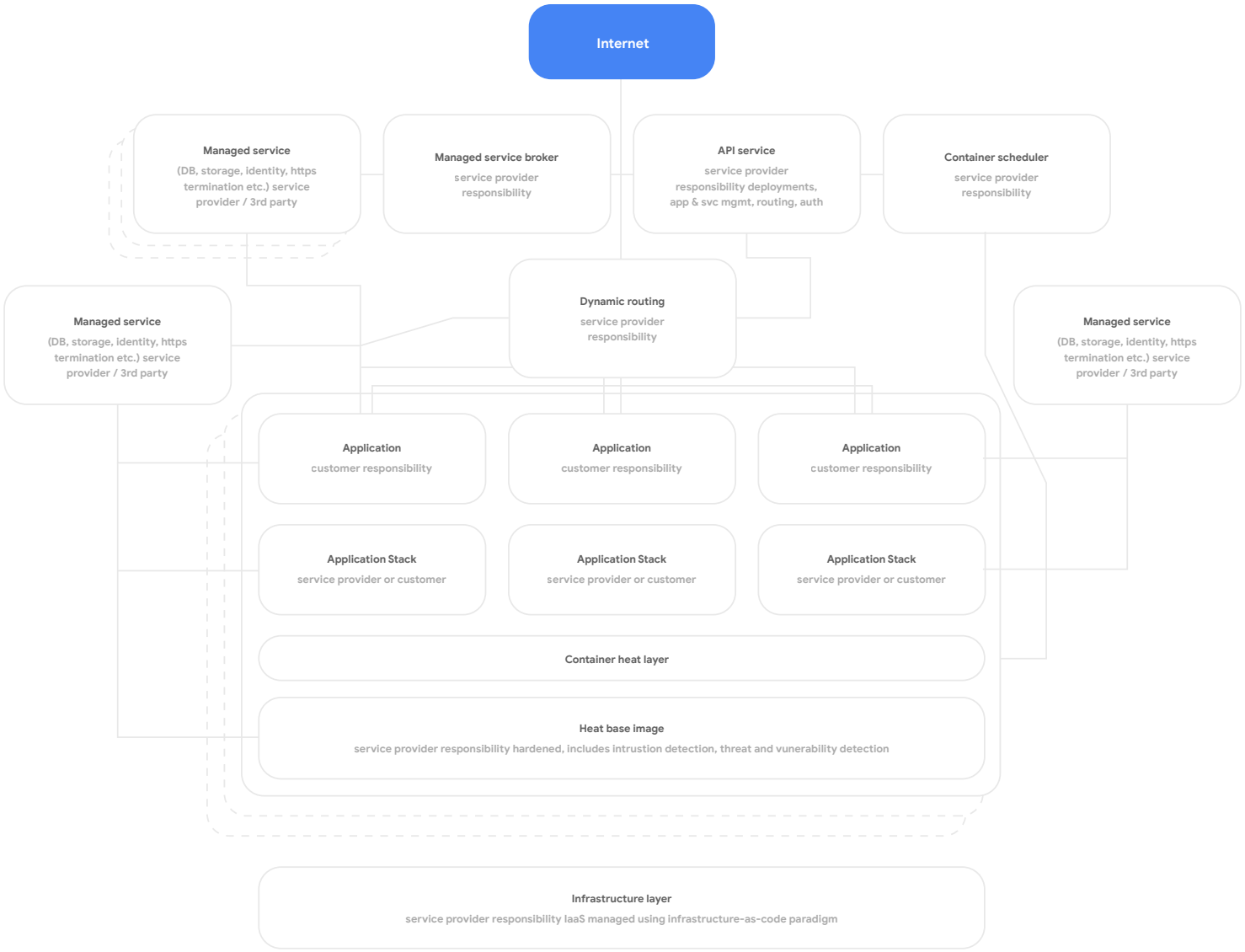

Figure 3: High-level anatomy of a cloud platform

Finally, **design for autonomous, loosely-coupled teams who can test and deploy their own services**. Our research shows that the most important architectural outcomes are whether software delivery teams can answer “yes” to these six questions:

- We can make large-scale changes to the design of our system without the permission of somebody outside the team;
- We can make large-scale changes to the design of our system without depending on other teams to make changes in their systems or creating significant work for other teams;
- We can complete our work without communicating and coordinating with people outside the team;
- We can deploy and release our product or service on demand, regardless of other services it depends upon;
- We can do most of our testing on demand, without requiring an integrated test environment;
- We can perform deployments during normal business hours with negligible downtime.

Our research shows that the extent to which teams agreed with these statements strongly predicted high software performance: the ability to deliver reliable, highly available services multiple times per day. This, in turn, is what enables high-performing teams to increase developer productivity (measured in terms of number of deployments per developer per day) even as the number of teams increases.

Check on a regular basis whether teams are working towards these goals, and prioritize achieving them. This usually involves re-thinking both organizational and enterprise architecture.

In particular, organizing teams so that all the various roles needed to build, test, and deploy software, including product managers, are working together and using modern product management practices to build and evolve the services they are working on is crucial. This need not involve changes to organizational structure. Simply having those people work together as a team on a day-to-day basis (sharing a physical space where possible) rather than having developers, testers, and release teams operating independently can make a big difference to productivity.

Microservices architecture principles and practices

When adopting a microservices or service-oriented architecture, there are some important principles and practices you must ensure you follow. It's best to be very strict about following these from the beginning, as it is more expensive to retrofit them later on.

- **Every service should have its own database schema**. Whether you're using a relational database or a nosql solution, each service should have its own schema that no other service accesses. When multiple services talk to the same schema, over time the services become tightly coupled together at the database layer. These dependencies prevent services from being independently tested and deployed, making them harder to change and more risky to deploy.

- **Services should only communicate through their public APIs over the network**. All services should expose their behavior through public APIs, and services should only talk to each other through these APIs. There should be no "back door" access, or services talking directly to other services' databases. This avoids services becoming tightly coupled, and ensures inter-service communication uses well-documented and supported APIs.

- **Services are responsible for backwards compatibility for their clients**. The team building and operating a service is responsible for making sure that updates to the service don't break its consumers. This means planning for API versioning and testing for backwards compatibility, so that when you release new versions, you don't break existing customers. Teams can validate this using canary releasing. It also means making sure deployments do not introduce downtime, using techniques such as blue/green deployments or staged roll-outs.

- **Create a standard way to run services on development workstations.** Developers need to be able to stand up any subset of production services on development workstations on demand using a single command. It should also be possible to run stub versions of services on demand—make sure you use emulated versions of cloud services that many cloud providers supply to help you. The goal is to make it easy for developers to test and debug services locally.

- **Invest in production monitoring and observability.** Many problems in production, including performance problems, are emergent and caused by interactions between

multiple services. Our research shows it's important to have a solution in place that reports on the overall health of systems (for example, are my systems functioning? do my systems have sufficient resources available?) and that teams have access to tools and data that helps them trace, understand, and diagnose infrastructure problems in production environments, including interactions between services.

- **Set service-level objectives (SLOs) for your services and perform disaster recovery tests regularly.** Setting SLOs for your services sets expectations on how it will perform and helps you plan how your system should behave if a service goes down (a key consideration when building resilient distributed systems.) Test how your production system behaves in real life using techniques such as controlled failure injection as part of your disaster recovery testing plan—DORA's research shows that organizations that conduct disaster recovery tests using methods like this are more likely to have higher levels of service availability. The earlier you get started with this the better, so you can normalize this kind of vital activity.

This is a lot to think about, which is why it's important to pilot this kind of work with a team that has the capacity and capability to experiment with implementing these ideas. There will be successes and failures—it's important to take lessons from these teams and leverage them as you spread the new architectural paradigm out through your organization.

Our research shows that companies that succeed use proofs-of-concept and provide opportunities for teams to share learnings, for example by creating communities of practice. Provide time, space, and resources for people from multiple teams to meet regularly and exchange ideas. Everybody will also need to learn new skills and technologies. Invest in the growth of your people by giving them budget for buying books, taking training courses, and attending conferences. Provide infrastructure and time for people to spread institutional knowledge and good practice through company mailing lists, knowledge bases, and in-person meetups.

Reference architecture

In this section, we'll describe a reference architecture based on the following guidelines:

- Use containers for production services and a container scheduler such as Cloud Run or Kubernetes for orchestration
- Create effective CI/CD pipelines
- Focus on security

Containerize production services

The foundation of a containerized cloud application is a container management and orchestration service. While many different services have been created, one is clearly dominant today: Kubernetes. According to Gartner, “Kubernetes has emerged as the de facto standard for container orchestration, with a vibrant community and support from most of the leading commercial vendors.” Figure 4 summarizes the logical structure of a Kubernetes cluster.

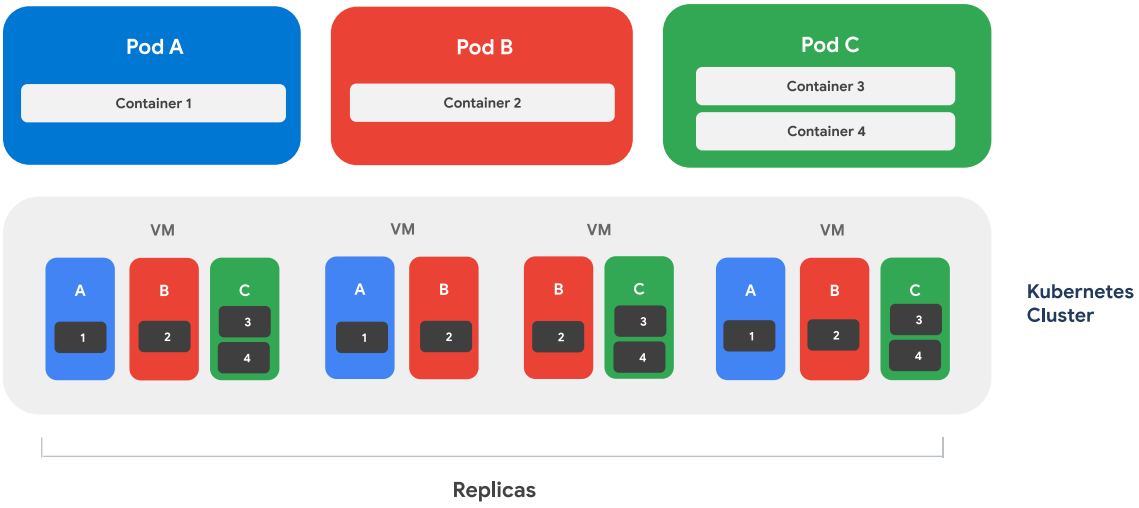

Figure 4: A Kubernetes cluster runs workloads, with each pod comprising one or more containers.

Kubernetes defines an abstraction called a pod. Each pod often includes just one container, like pods A and B in Figure 4, although a pod can contain more than one, as in pod C. Each Kubernetes service runs a cluster containing some number of nodes, each of which is typically a virtual machine (VM). Figure 4 shows only four VMs, but a real cluster might easily contain a hundred or more. When a pod is deployed on a Kubernetes cluster, the service determines which VMs that pod’s containers should run in. Because containers specify the resources they need, Kubernetes can make intelligent choices about which pods are assigned to each VM.

Part of a pod’s deployment information is an indication of how many instances—replicas—of the pod should run. The Kubernetes service then creates that many instances of the pod’s containers and assigns them to VMs. In Figure 4, for example, pod A’s deployment

requested three replicas, as did pod C's deployment. Pod B's deployment, however, requested four replicas, and so this example cluster contains four running instances of container 2. And as the figure suggests, a pod with more than one container, such as pod C, will always have its containers assigned to the same node.

Kubernetes also provides other services, including:

- **Monitoring running pods**, so if a container fails, the service will start a new instance. This makes sure that all the replicas requested in a pod's deployment remain available.
- **Load balancing traffic**, spreading requests made to each pod intelligently across a container's replicas.
- **Automated zero-downtime rollout of new containers**, with new instances gradually replacing existing instances until a new version is fully deployed.
- **Automated scaling,** with a cluster autonomously adding or deleting VMs based on demand.

Create effective CI/CD pipelines

Some of the benefits of refactoring a monolithic application, such as lower costs, flow directly from running on Kubernetes. But, one of the most important benefits—the ability to update your application more frequently—is only possible if you change how you build and release software. Getting this benefit requires you to create effective CI/CD pipelines in your organization.

Continuous integration relies on automated build and test workflows that provide fast feedback to developers. It requires every member of a team working on the same code (for example, the code for a single service) to regularly integrate their work into a shared mainline or trunk. This integration should happen at least daily per developer, with each integration verified by a build process that includes automated tests. Continuous delivery aims at making deployment of this integrated code quick and low risk, largely by automating the build, test, and deployment process so that activities such as performance,

security, and exploratory testing can be performed continuously. Put simply, CI helps developers detect integration issues quickly, while CD makes deployment reliable and routine.

To make this clearer, it's useful to look at a concrete example. Figure 5 shows how a CI/CD pipeline might look using Google tools for containers running on Google Kubernetes Engine.

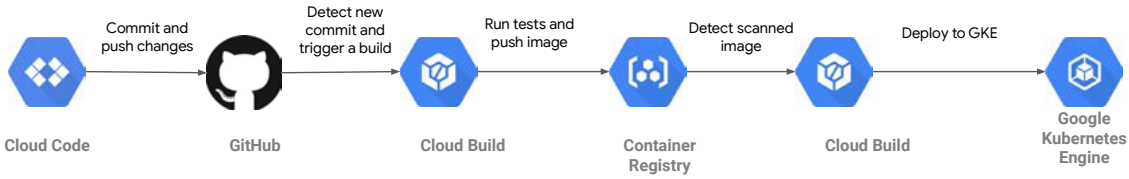

Figure 5: A CI/CD pipeline has several steps, from writing code to deploying a new container.

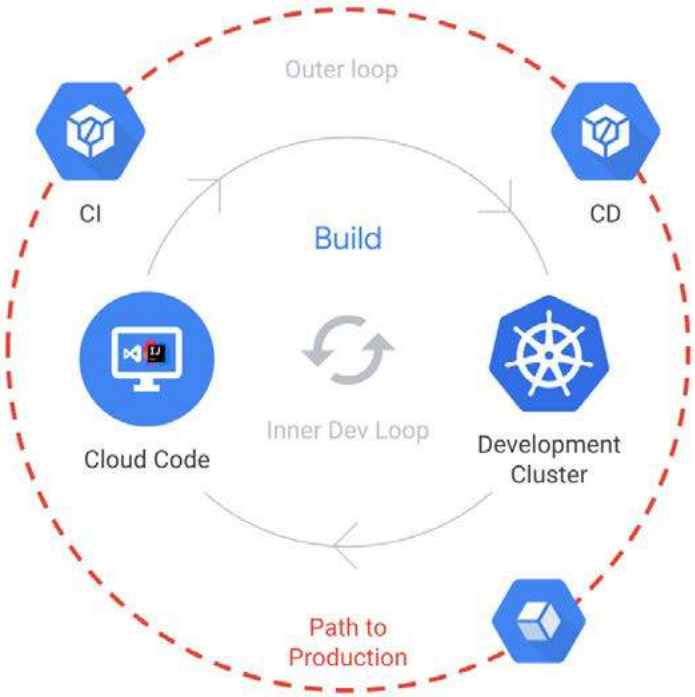

Figure 6: Local and remote development loops

It's useful to think of the process in two chunks, as shown in Figure 6:

- Local Development: The goal here is to speed up an inner development loop and provide developers with tooling to get fast feedback on the impact of local code changes. This includes support for linting, auto-complete for YAML, and faster local builds.

- Remote Development: When a pull request (PR) is submitted, the remote development loop kicks off. The goal here is to drastically reduce the time it takes to validate and test the PR via CI and perform other activities like vulnerability scanning and binary signing, while driving release approvals in an automated fashion.

Here is how Google Cloud tools can help during this process:

Local development: Making developers productive with local application development is essential. This local development entails building applications that can be deployed to local and remote clusters. Prior to committing changes to a source control management system like GitHub, having a fast local development loop can ensure developers get to test and deploy their changes to a local cluster.

Google Cloud as a result provides Cloud Code. Cloud Code comes with extensions to IDEs, such as Visual Studio Code and Intellij to let developers rapidly iterate, debug, and run code on Kubernetes. Under the covers Cloud Code uses popular tools, such as Skaffold, Jib, and Kubectl to enable developers to get continuous feedback on code in real time.

Continuous integration: With the new Cloud Build GitHub App, teams can trigger builds on different repo events - pull requests, branch, or tag events right from within GitHub. Cloud Build is a fully serverless platform and scales up and down in response to load with no requirement to pre-provision servers or pay in advance for extra capacity. Builds triggered via the GitHub App automatically post status back to GitHub. The feedback is integrated directly into the GitHub developer workflow, reducing context switching.

Artifact Management: Container Registry is a single place for your team to manage Docker images, perform vulnerability scanning, and decide who can access what with fine-grained access control. The integration of vulnerability scanning with Cloud Build lets developers identify security threats as soon as Cloud Build creates an image and stores it in the Container Registry.

Continuous delivery: Cloud Build uses build steps to let you define specific steps to be performed as a part of build, test, and deploy. For example, once a new container has been created and pushed to Container Registry, a later build step can deploy that container to Google Kubernetes Engine (GKE) or Cloud Run - along with the related configuration and policy. You can also deploy to other cloud providers in case you are pursuing a multi-cloud strategy. Finally, if you are someone looking to pursue GitOps-style continuous delivery, Cloud Build lets you describe your deployments declaratively using files (for example, Kubernetes manifests) stored in a Git repository.

Deploying code isn't the end of the story, however. Organizations also need to manage that code while it executes. To do this, GCP provides operations teams with tools, such as Cloud Monitoring, and Cloud Logging.

Using Google's CI/CD tools certainly isn't required with GKE—you're free to use alternative toolchains if you wish. Examples include leveraging Jenkins for CI/CD or Artifactory for artifact management.

If you're like most organizations with VM-based cloud applications, you probably don't have a well-oiled CI/CD system today. Putting one in place is an essential part of getting benefits from your re-architected application, but it takes work. The technologies needed to create your pipelines are available, due in part to the maturity of Kubernetes. But the human changes can be substantial. The people on your delivery teams must become cross-functional, including development, testing, and operations skills. Shifting culture takes time, so be prepared to devote effort to changing the knowledge and behaviors of your people as they move to a CI/CD world.

Focus on security

Re-architecting monolithic applications to a cloud-native paradigm is a big change. Unsurprisingly, doing this introduces new security challenges that you'll need to address. Two of the most important are:

- Securing access between containers
- Ensuring a secure software supply chain

The first of these challenges stems from an obvious fact: breaking your application up into containerized services (and perhaps microservices) requires some way for those services to communicate. And even though they're all potentially running on the same Kubernetes cluster, you still need to worry about controlling access between them. After all, you might be sharing that Kubernetes cluster with other applications, and you can't leave your containers open to these other apps.

Controlling access to a container requires authenticating its callers, then determining what requests these other containers are authorized to make. It's typical today to solve this problem (and several others) by using a service mesh. A leading example of this is Istio, an open source project created by Google, IBM, and others. Figure 7 shows where Istio fits in a Kubernetes cluster.

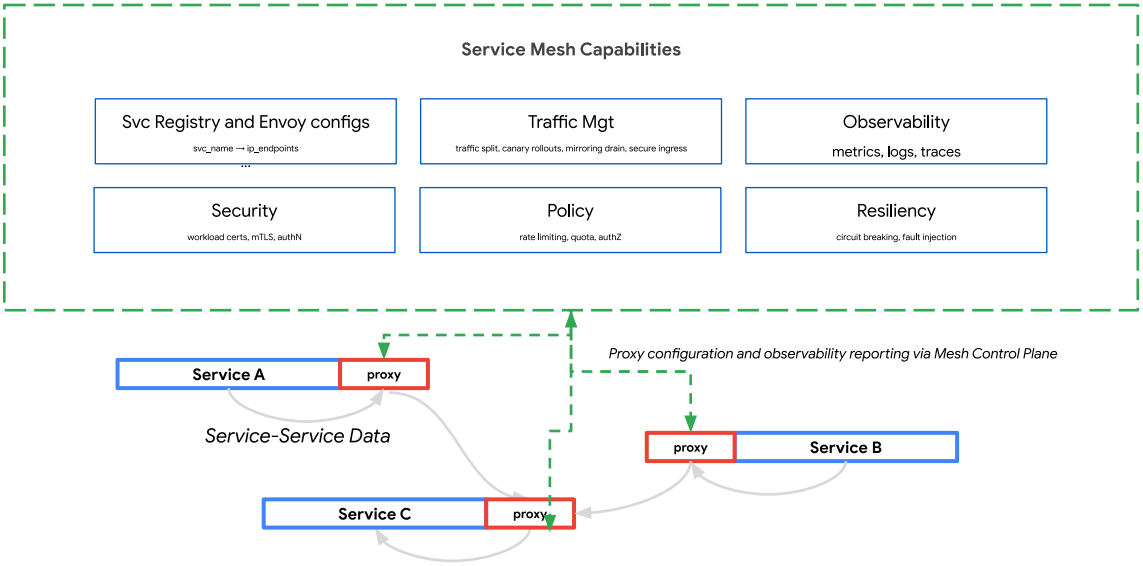

Figure 7: In a Kubernetes cluster with Istio, all traffic between containers goes through this service mesh.

As the figure shows, the Istio proxy intercepts all traffic between containers in your application. This lets the service mesh provide several useful services without any changes to your application code. These services include:

- **Security,** with both service-to-service authentication using TLS and end-user authentication
- **Traffic management,** letting you control how requests are routed among the containers in your application
- **Observability,** capturing logs and metrics of communication among your containers

GCP lets you add Istio to a GKE cluster. And even though using a service mesh isn't required, don't be surprised if knowledgeable customers of your cloud applications start to ask whether your security is up to the level that Istio provides. Customers care deeply about security, and in a container-based world, Istio is an important part of providing it.

Along with supporting open source Istio, Google Cloud offers Traffic Director, a fully GCP-managed service mesh control plane that delivers global load balancing across clusters and VM instances in multiple regions, offloads health checking from service proxies, and provides sophisticated traffic management and other capabilities described above.

One of the unique capabilities of Traffic Director is automatic cross-region failover and overflow for microservices in the mesh (shown in Figure 8). You can couple global resiliency with security for your services in the service mesh using this capability.

Figure 8: Traffic Director provides support for automatic cross-region failover and overflow

Traffic Director offers several traffic management features that can help improve the security posture of your service mesh. For example - the traffic mirroring feature shown in Figure 9 can be easily set up as a policy to allow a shadow application to receive a copy of the real traffic being processed by the main version of the app. Copy responses received by the shadow service are discarded after processing. Traffic mirroring can be a powerful tool to test for security anomalies and debug errors in production traffic without impacting or touching production traffic.

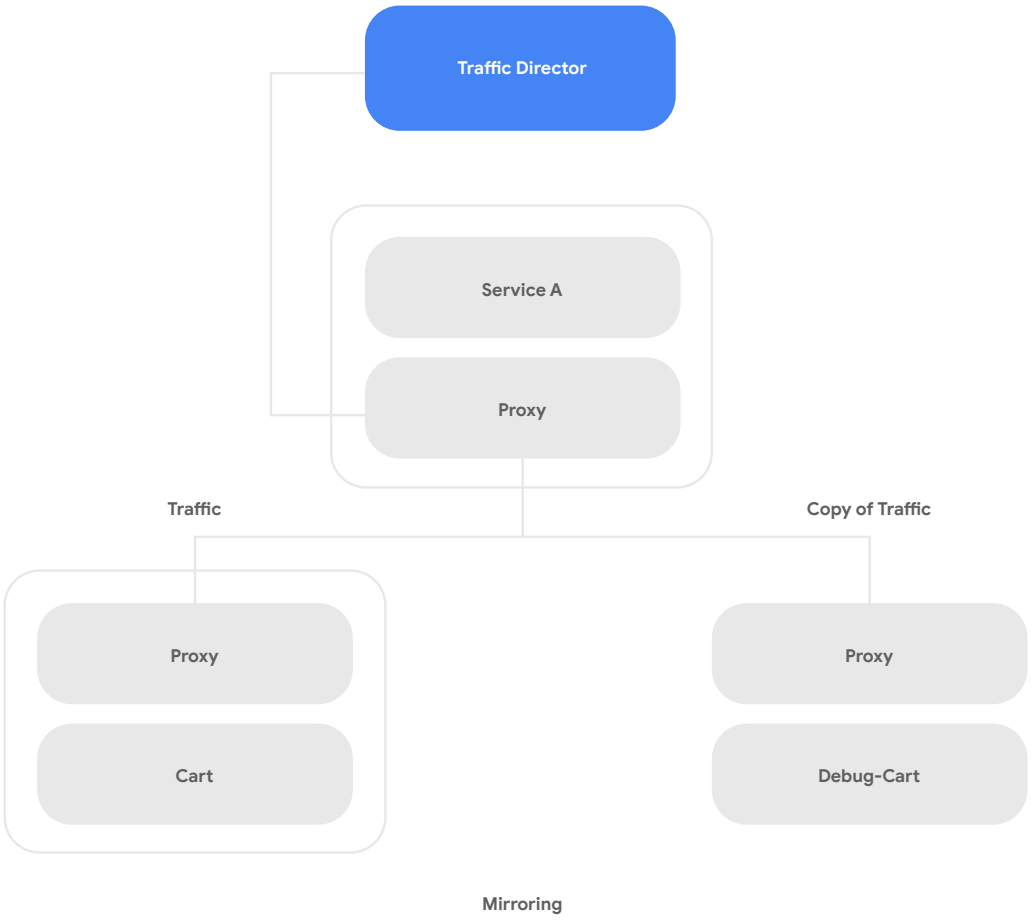

Figure 9: Traffic mirroring in Traffic Director

But protecting interactions among your containers isn't the only new security challenge that a refactored application brings. Another concern is ensuring that the container images you run are trustworthy. To do this, you must make sure that your software supply chain has security and compliance baked in.

Doing this requires doing two main things (shown in Figure 10):

- **Vulnerability scanning:** Container Registry Vulnerability Scanning lets you get quick feedback on potential threats and identify issues as soon as your containers are built by Cloud Build and stored in Container Registry. Package vulnerabilities for Ubuntu, Debian, and Alpine are identified right during the application development process, with support for CentOS and RHEL on the way. This helps avoid costly inefficiencies and reduces the time required to remediate known vulnerabilities.

- **Binary Authorization:** By integrating Binary Authorization and Container Registry vulnerability scanning, you can gate deployments based on vulnerability scanning findings as part of the overall deploy policy. Binary Authorization is a deploy-time security control that ensures only trusted container images are deployed on GKE without any manual intervention.

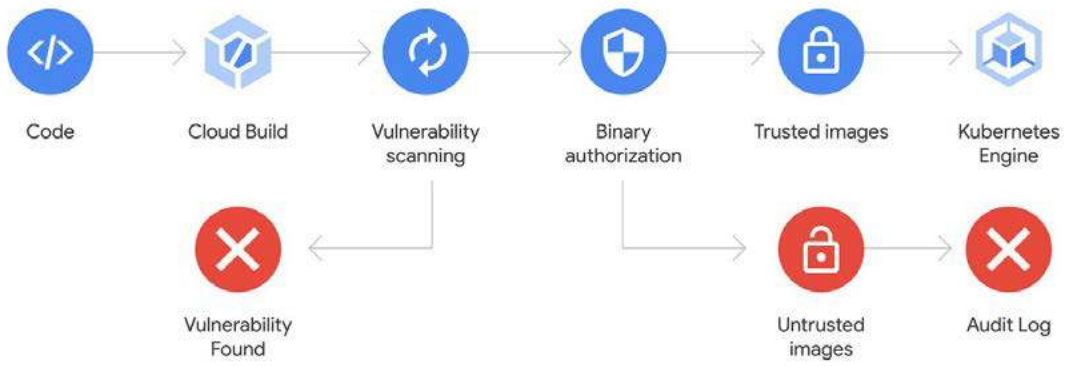

Figure 10: Workflow for vulnerability scanning and binary authorization

Securing access between containers with a service mesh and ensuring a secure software supply chain are important aspects of creating secure container-based applications. There are plenty more, including verifying the security of the cloud platform infrastructure you're building on. What's most important, though, is to realize that moving from a monolithic application to a modern cloud-native paradigm introduces new security challenges. To successfully make this transition, you'll need to understand what these are, then create a concrete plan for addressing each one.

Getting started

Don't treat moving to a cloud-native architecture as a multi-year, big-bang project. Instead, start now by finding a team with the capacity and expertise to get started with a proof of concept - or find one that has already done this. Then, take the lessons learned and start to spread them throughout the organization. Have teams adopt the strangler fig pattern, moving services incrementally and iteratively over to a cloud-native architecture as they continue to deliver new functionality.

In order to succeed, it's essential teams have the capacity, resources, and authority to make evolving their systems architecture part of their daily work. Set clear architectural goals for new work - following the 6 architectural outcomes presented previously—but give teams freedom to decide how to get there.

Most important of all—don't wait to get started! Increasing the productivity and agility of your teams and the security and stability of your services is going to become ever more critical to the success of your organization. The teams that do best are the ones that make disciplined experimentation and improvement part of their daily work.

Google invented Kubernetes, based on software we have used internally for years, which is why we have the deepest experience with cloud-native technology. Google Cloud Platform has a strong focus on containerized applications, as evidenced by our CI/CD and security offerings. The truth is clear: GCP is today's leader in support for containerized applications.

Visit cloud.google.com/devops to take our Quick Check to find out how you're doing and for advice on how to proceed, including implementing patterns discussed in this white paper, such as a loosely coupled architecture.

Many GCP partners have already helped organizations like yours make this transition. Why blaze a rearchitecting trail on your own when we can connect you with an experienced guide?

To get started, contact us to arrange a meeting with a Google solutions architect. We can help you understand the change, then work with you on how to make it happen.

Further reading

cloud.google.com/devops - six years of the State of DevOps Report, a set of articles with in-depth information on the capabilities that predict software delivery performance, and a quick check to help you find out how you're doing and how to get better.

Site Reliability Engineering: How Google Runs Production Systems (O'Reilly 2016)

The Site Reliability Workbook: Practical Ways to Implement SRE (O'Reilly 2018)

Building Secure and Reliable Systems: Best Practices for Designing, Implementing and Maintaining Systems (O'Reilly 2020)

"How to break a Monolith into Microservices: What to decouple and when" by Zhamak Dehghani https://martinfowler.com/articles/break-monolith-into-microservices.html

---

"Microservices: a definition of this new architectural term" by Martin Fowler https://martinfowler.com/articles/microservices.html

---

"Strangler Fig Application" by Martin Fowler https://martinfowler.com/bliki/StranglerFigApplication.html

Footnotes

[1] Find out how your team is performing based on these four key metrics at https://cloud.google.com/devops/

---

[2] https://services.google.com/fh/files/misc/state-of-devops-2015.pdf

---

[3] This is not intended to be a complete description of what “cloud native” means: for a discussion of some of the principles of cloud-native architecture, visit https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it.

---

[4] Described in https://martinfowler.com/bliki/StranglerFigApplication.html

---


Google Cloud