Application modernization and the decoupling of infrastructure services and teams

By Eric Brewer and Jennifer Lin, Google

Google Cloud

Table of Contents

Application Modernization and the Decoupling of Infrastructure, Services and Teams

Summary

Modernizing applications on a public cloud offers many advantages in terms of cost and productivity, but these benefits are often presented as an "all or nothing" choice: move to the cloud and modernize. Enterprises want to modernize independently from the migration to a public cloud, and to enable an incremental path for migration that mixes on-prem and cloud-based solutions as needed. This modernization is critical to enable business innovation, for example, incorporating advanced machine learning and data analytics.

In addition, the various public clouds provide quite different platforms, which makes workload portability quite difficult, both technically and in terms of developers' skills. Yet most enterprises want the option to use multiple clouds, change providers over time, and in general preserve some independence from their vendors.

These two broad patterns—the need for modernization on-premises and the desire for multi-cloud solutions—call for a new platform that can run consistently in many environments, and that provides modern, agile development and operations across these environments. Kubernetes is the established leader for container orchestration and workload portability. It has rapidly become the de facto standard to **orchestrate platform-agnostic applications**. It is the foundation of the broad platform described above. By simultaneously addressing the needs of agile development and enterprise operations teams, the industry can now focus on its original objective: building a truly autonomous cloud service platform—one that allows dynamic distributed systems to evolve organically, enables extensive and varied use of machine learning, but that

also delivers a consistent developer experience and a single point of administrative control.

Google has built Anthos (formerly known as Cloud Services Platform or CSP) to **accelerate application modernization** for SaaS providers, developers, IT operators and their end users. In order to balance developer agility, operational efficiency and platform governance, the Anthos framework enables decoupling across critical components:

- Infrastructure is decoupled from the applications (via containers and Kubernetes)
- Teams are decoupled from each other (via Kubernetes pods and services)
- Development is decoupled from operations (via service and policy management)
- Security is decoupled from development and operations (via structured policy management)

Successful decoupling of infrastructure, services and teams minimizes the need for manual coordination, reduces cost and complexity, and significantly increases developer velocity, operational efficiency and business productivity. It delivers a framework, implementation, and operating model to ensure consistency across an open, hybrid and multi-cloud future.

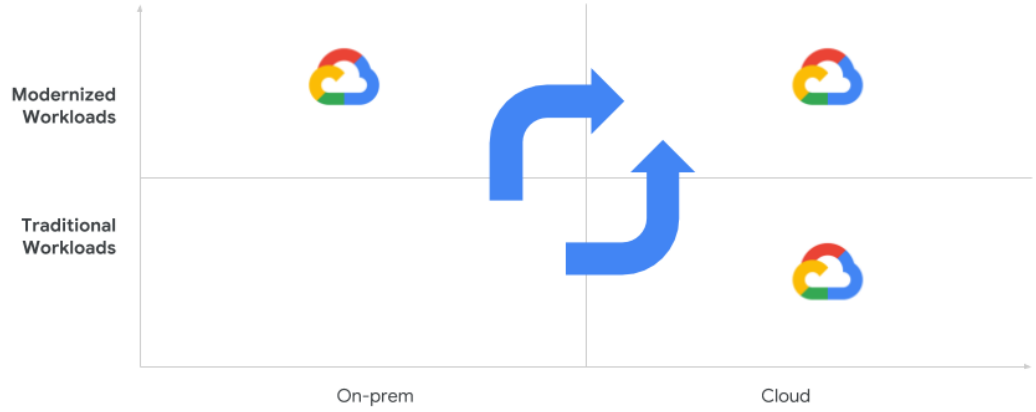

Figure 1: Paths for Modernization

Accelerating Application Modernization via Decoupling

Decoupling Infrastructure and Applications: Containers and Kubernetes

The “cloud-native” transition is really about creating a higher level of abstraction beyond just virtual machines. Virtual machines (VMs) help with many challenges, but by definition they represent infrastructure not applications. In particular, using VM images for applications tightly couples the VM, the operating system (OS), the application and its libraries.

Thus the first role of containers, driven by Docker, is to package up an application independently from the infrastructure, including ideally the machine, libraries and the OS. In fact, one of Docker’s first successful use cases was to build and test applications on a laptop and then deploy them to the cloud. For example, the containers define the correct library dependencies and bring them to the infrastructure, ignoring what might already be there, and thus allow potentially conflicting libraries to coexist on the same machine.

The second role of containers, sometimes referred to as “Linux containers,” is to provide isolation among different applications on the same machine. Google pioneered this line of work over a decade ago so that we could pack many applications onto the same hardware without worrying (much) about how they interfere with one another. This allows us to view scheduling on a cluster as mostly a bin-packing problem that is agnostic to the particular applications.[1]

Given their success with ‘write once, run anywhere’ portability and performance isolation on shared infrastructure, containers have proven to be a critical first

[1] Google solved the packaging problem internally not via Docker-like containers, but rather by a mix of static linking and forced use of some low-level libraries. This works well when you have full control of all your applications, but it is heavy handed and not as flexible as the Docker model. See https://research.google.com/pubs/pub43438.html?hl=es

step to decoupling applications from infrastructure and a foundational element of application modernization.

Decoupling Cloud Teams: Pods and Services

For simple applications, containers provide enough structure: a team can build their application as a container and then deploy it. But for more complex applications involving multiple teams we want to decouple the teams from one another. This is evident most clearly with teams responsible for shared infrastructure services, such as monitoring or logging. The logging team should be able to evolve logging independently from application updates. If the customer uses only a single container per app, they must build a version of logging into their container and deploy that version. To upgrade logging then requires that customer to deploy a new container – coupling the application and logging teams, requiring coordination and slowing them both down.

What About Pods?

Kubernetes introduced *pods* to help with this kind of decoupling. A pod is a group of containers that are co-scheduled (run on the same machine), can share machine resources such as local volumes, and are allocated a single IP address. In the logging example, the pod contains both the application container and the logging container, and the two communicate via shared volumes and/or localhost networking. This loose coupling enables independent upgrades: the application container only includes the logging API client, but not the logging implementation. This pattern of having a helper container in the pod is often referred to as a "sidecar."

Each pod having its own IP address also enables decoupling — every pod has the full range of ports available to it. This is critical because by convention most applications assume availability of a fixed port, such as port 80 for HTTP. DNS fosters this convention by not including port numbers in resolution (in normal use), so the port is assumed.[2] Traditionally, ports are both an application concept and a machine

[2] Google's internal solution, which we are moving away from, was to change DNS to include port numbers and then dynamically allocate ports to applications as needed. This is not practical for most packaged applications, and in general, we want to avoid needing to change the applications.

concept, so it follows that to have many applications per machine, we need many IP addresses per machine. Pods solve this problem cleanly and in fact, a machine with 100 pods will have at least 100 IP addresses.

Basic Services

Modern architectures are "service oriented" but before we can answer **"what is a service?"** we will start with the service abstraction that is built into Kubernetes. In later sections, we'll discuss more advanced functionality built on top of these basic services.

A basic service in Kubernetes is usually a dynamic set of pods behind a grouping mechanism that implements various policies and maintains the service IP. This simple kind of service is sometimes called a *microservice*, but services need not be particularly small — it is more about the building block and the team responsible for the service. Google runs more than 10,000 services at all times.

By far the most important aspect of a service is that it has a highly available, persistent name. The primary name is just an IP address: Kubernetes services use a "service IP" that is independent from the IPs of the underlying pods that implement that service.[3] The "grouping mechanism" mentioned above isoften a proxy, but it can also be various forms of routing or NAT that map the service IP to the pod IPs dynamically. The non-proxy forms are less flexible, but often have better performance.

Decoupling the service's name from its implementation enables several key customer benefits:

- **Online upgrades** are possible as the lifecycle of services and constituent pods are decoupled;
- **Scaling** is easier as pods can adapt to load (manually or automatically) transparently to clients;
- **Policy definition and enforcement** can be unified via proxies to simplify administration for more complex properties such as security and operations.

Services make it **easy to mix languages across teams and application components**.

[3] Kubernetes also exposes DNS to find services by name

Each service is implemented by its own pod(s), which can be implemented in any language or style. Services represent APIs, which can be defined with IDLs and schema descriptions, often JSON over HTTP. Similarly, clients can be handled in idiomatic ways in a wide variety of languages. Services that need high performance can use protobufs and gRPC[4], which are implemented in over a dozen languages.

Language independence is also a big plus for sidecars, like the logging agent. Because the communication channel is local, it is quite common for sidecars to be written in a different language than the primary application. Only the logging client API has to be written in the application's language, and it can often be generated automatically.

Services lead most groups to independently deploy upgrades that are backward compatible. **Version numbers** help decoupled teams understand whether clients are safe to use a new version. In practice, teams typically use conventions around version numbers to denote semantic changes.[5]

The basic services in Kubernetes are also stateless. By definition, all of the pods in a service come from the same specification and might be restarted from that spec at any time. This abstraction is sufficient for most services and also for 12-factor apps. Such cases assume that durable storage is managed inside another service.

Ultimately though, the real goal of services is to decouple *teams*. A team is decoupled if they can mostly deploy their work independently from other teams, which has been traditionally easier said than done. Services are the unit of deployment — they are the unit of encapsulation and the basis of APIs, a better version of "objects" in that they are bigger, long-lived and highly available. All the advice about what to hide inside objects also applies to services, and as with object-oriented programming, microservices can be taken too far.[6]

---

[4] https://grpc.io/
[5] See https://semver.org/ for an example
[6] For a good discussion of defining service boundaries and size, see the discussion of "modules" in John Ousterhout's recent book, ***A Philosophy of Software Design***.

The Power of the Proxy

As described above, a proxy is a core part of the definition of a service. This has been true since at least the 1990s[7] and it remains the key to several forms of decoupling, and it is our most important control point for implementing policy.

The initial use case for the proxy is just load balancing: spread the incoming requests across the active set of pods. In addition to making the service IP highly available, this also enables splitting traffic across versions for canary testing and more generally for A/B testing. It is also the mechanism used for a progressive rollout of a new version.

As described thus far, the proxy operates at Layer 4 (L4), working at the level of TCP and IP. This is sufficient for basic services and works well with legacy applications that expect DNS, an IP address, and a port number. Services are at layer 7 (L7) typically use HTTPS requests, which provide more information and thus enable more sophisticated policies. Longer term, we expect L4 to be used for coarse-grained access control (to a cluster or a namespace within a cluster), and L7 to be used to implement the more complex policies required by modern enterprises and dynamic applications. These L7 policies can be managed using modern source control mechanisms that include explicit review and acceptance of source code changes (e.g. GitOps).

The proxy is also a great point for telemetry: it can measure service-level indicators (SLIs) for the service, such as request latency and throughput, without needing to know anything about the service (a hallmark of decoupling). It can also health check pods and remove them from duty, and its telemetry provides one basis for auto-scaling the service.

We discuss two different uses of proxies in the next two sections. The first kind manages the traffic among services within an application, which is sometimes

[7] See https://people.eecs.berkeley.edu/~brewer/papers/TACC-sosp.pdf.

called “East-West” traffic based on the typical architecture diagram with users on top (North) and services spread left to right (West to East). The second kind is a more traditional user-facing proxy, which manages traffic into the application (North-South) and provides user authentication, access control, various kinds of API management and mediation.

Microservices and Lifecycle Management with Istio

Kubernetes proxies provide a service abstraction mechanism, but with somewhat limited functionality. Recent proxy implementations (e.g. Envoy) have popularized the concept of a sidecar proxy which adds significant functionality and flexibility. As it is inefficient to configure individual proxies, seamless injection of and systematic management across proxies is required. This is the essence of Istio, an open-source project that Google started with Lyft and IBM[8] to address the unique challenges of microservice management.

Traffic Management

Istio manages a group of proxies that tie together the components of a *service mesh*. The mesh represents the set of services inside of a larger user-facing application. Istio manages service-to-servicetraffic (East-West) and relies on proxies in front of each service. This enables client-side load balancing across services instances as well as ingress capabilities, such as A/B testing and canary releases for user-facing services. Mesh proxies also enable egress capabilities such as timeouts, retries and circuit breakers and improve fault tolerance when routing to external web services. Critically, Istio provides **systematic centralized management of these proxies**, and thus of the policies they implement. This is arguably the most important form of decoupling: decoupling the policies from the services. In particular, this allows developers to avoid encoding policies inside the services, which means they can change the policies without redeploying the service. Instead, to change policies

[8] https://cloud.google.com/blog/products/gcp/istio-modern-approach-to-developing-and

they just update the configuration of the relevant proxies. Because the management is centralized, it is easy to be consistent across all of the services, and it's possible to apply updates in a controlled rollout to improve safety.

At a higher level, this model also **decouples operations from development**. A classic problem in IT is that operations folks want to enforce policies uniformly, but often have to work closely with developers to make sure every service does the right thing. This results in expensive coordination, such as blocking launches for policy reviews. In the Istio model, less coordination is needed since the policies live outside of the services. In turn, development teams have higher velocity, can launch every day (or more), and generally have more autonomy and higher morale as well. Broadly speaking, we want service developers to be accountable for those aspects of operations that have to do with availability and rollouts, while leaving broad policy specification and enforcement to more centralized groups.

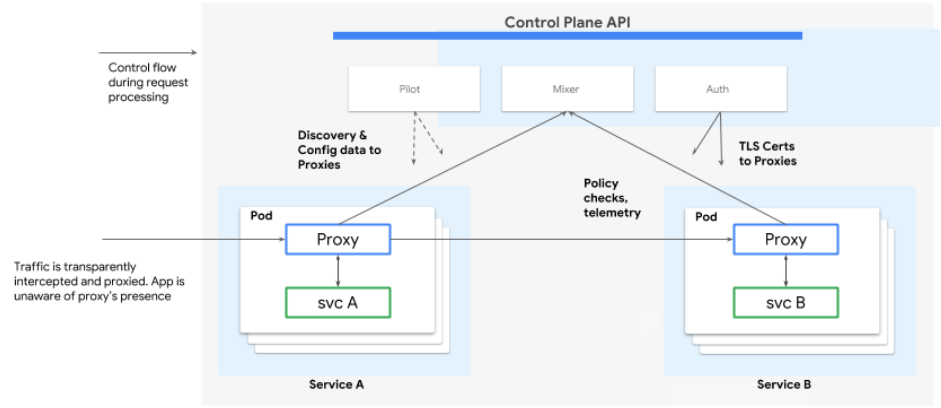

Figure 2: Istio Architecture

Istio takes these proxies, as shown in Figure 2, and manages them, providing a range of benefits that turn basic services into advanced services with consistent properties and policies.

Security

In addition to traffic management, Istio authenticates services to each other, encrypts inter-service traffic (mutual TLS), provides access-control for calling services, and generates an audit trail. These features are added transparently without the need for application changes or tedious certificate management. Istio authentication automates service identity and authorization to enable granular application-layer controls.For example, it would be bad if an adversary's process could invoke the the credit card service, and cause unintended payments. (See Istio Security for more details.)

Observability

Visibility of services is critical to the goal of running in production securely, and understanding telemetry in real time is the cornerstone of building a secure platform. Istio automates the instrumentation of services by auto-generating and collecting monitoring data and logs for them. By doing instrumentation this way, Istio enables consistent data collection across diverse services and backends[9]. This makes it easy to generate dashboards and provide common SLIs such as latency distributions, throughput, and error rates. In turn, the consistency of metrics simplifies automation tasks such as auto-scaling and health checks (see Istio Telemetry examples).

Stateful services and databases

While the majority of Kubernetes workloads today are stateless, Kubernetes can accommodate stateful services with the concept of StatefulSets, where each pod has a stable identity and dedicated disk that is maintained across restarts. This extends container orchestration to distributed databases, streaming data pipelines and other stateful services. While Kubernetes focuses on container orchestration at scale, new programming models are emerging to develop applications and runtimes which can manage business logic, data consistency and increasingly distributed workflows.

[9] Istio Mixer adapters allow pluggable backends for data such as metrics and logs (https://istio.io/docs/reference/config/policy-and-telemetry/adapters/)

Serverless and Events

Open source frameworks such as Knative codify the best practices around development of cloud native applications via source-driven functions and dynamic, event-driven microservices. Producers can generate events and consumers can subscribe to events a priori, only paying for active usage of request-driven compute. The Knative serving framework builds on Kubernetes and Istio with middleware that enables rapid deployment of serverless containers, auto-scaling to zero, intelligent routing and snapshotting of deployed code and configs.

Zero-Trust Security and Cloud Assurance Modernization

Google pioneered the zero-trust access security model using North-South proxies to enforce more granular controls while maintaining a consistent user experience (see BeyondCorp: *A New Approach to Enterprise Security*[10] and BeyondCorp: *The Access Proxy*[11]). A key component of the overall Google security architecture (more at *https://cloud.google.com/security/overview/whitepaper*), this context-aware access model enforces service and user authentication and applies dynamic access controls across all applications.

By inserting an application-layer gateway between the user/browser and backend

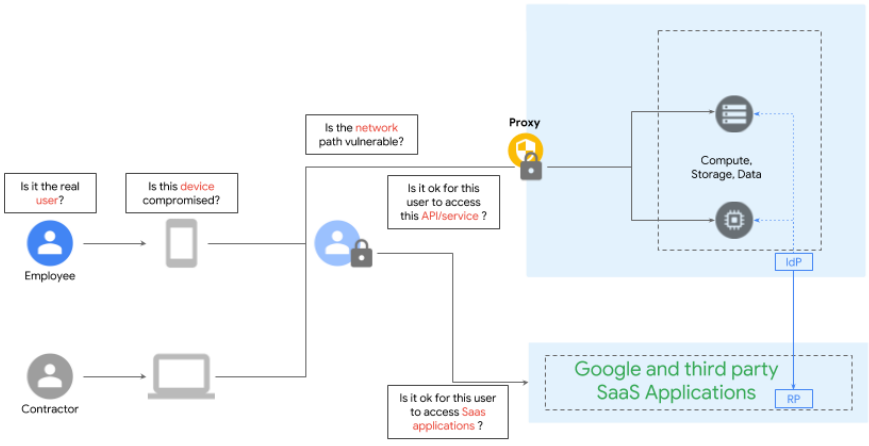

Figure 3: The zero-trust model across users, resources and applications/services

[10] https://ai.google/research/pubs/pub43231
[11] https://research.google.com/pubs/pub45728.html?hl=tr

service endpoints, this approach addresses a key challenge with traditional enterprise models, which still rely on a hard network-level perimeter with firewalls and VPNs. As cloud SaaS and mobile applications represent the majority of enterprise workloads, trusted access uses 'real-time' context to properly enforce access policies for cloud endpoints and to minimize costly data breaches.

In addition to securing transactions, this proxy-based gateway approach helps to ensure that service producers can efficiently develop, deploy and manage the APIs serving any cloud backend.

The granular telemetry, control plane automation and technical/operational consistency of open cloud architectures allows enterprises to better assess their assets/services/users against cloud controls that reflect the best practices of cloud security governance. It also allows enterprises to evolve their 'shared responsibility' model for cloud, where customized security controls (vertical, region, role definitions, managed services, etc.) are defined upstream into the policies that govern the infrastructure (compute, storage, network), platform (automated CI/CD and GitOps pipelines, secure runtime, service authentication and lifecycle management, user identity and access privileges) and the application-layer (business logic, user experience).
Modernized end-to-end secure software supply chain and cloud commerce with zero-trust principles accelerates development and build, delivery and consumption of services, and delivers on these key principles and benefits:

All accesses for users or services are verified and granted via least-privileged access policies:

- User access to data is authenticated, authorized and logged against clear role-based policies
- Centralized policy/config administration for cloud resources, as well as users and services

- Strong service isolation so that any compromise has limited impact to other domains

Application and user context-aware access policies to protect sensitive data (non-replayable identities):

- Multiple factors (beyond RBAC, tokens) are used to authorize users, resources, services
- Data is encrypted at rest by default and data in transit travels across encrypted connections
- Binary authorization ensures proper signing and verification of portable workloads (VMs, containers)

Central monitoring and ongoing administration of security posture of resources, users and services:Consistent telemetry for granular service accesses, not just IP and infrastructure logging and monitoring

- Ongoing discovery to locate and index data, and controls to analyze, classify, protect data
- All secrets and certificates are signed with a valid root certificate subject to ongoing certificate management
- Automated patches and security updates are maintained by trusted industry experts

More than ever, enterprises are rapidly embracing open software frameworks (Kubernetes, Istio, Knative) in their existing enterprise/IT environments. This architectural evolution enables portability of workloads, vendor-agnostic cloud service abstractions and centralized administration of policies across heterogeneous environments, **improving security/cost governance and dramatically simplifying future cloud migrations.**

The Declarative Config Model and Kubernetes

So far we have discussed the use of containers and pods, and the use of managed proxies to create well-decoupled advanced services. The final foundational piece of our cloud service platform is the configuration model.

Configuration turns out to be one of the hardest aspects of modern applications, and over the years, Google has built and thrown out a wide variety of config systems, from the very simple to the overly complex.

Since nearly all aspects of applications and infrastructure require configuration, we need a model that is both consistent and extensible, but that at the same time works well with automation. Ideally, configuration tools should also be decoupled from the systems being configured.

The first question is what to use for the configuration language. It's very tempting, including at times for Google, to use a general-purpose language such as Python or bash scripts. General-purpose languages are very powerful, since they can solve pretty much any problem, but they lead to complex systems, can easily forfeit clarity, and their interpretation at runtime can lead to unexpected behavior in production.

Another challenge is that executing code as part of config makes it much harder to do automation. For example, config blocks occasionally fail when they execute. A person might be able to interpret that failure and fix it manually, but an automated system generally cannot, as it is hard to know what state the system is in after a failure. For example, is it OK to simply re-execute the config block? Or did it partially complete, leaving side effects that first need to be undone?

Instead, we use a restricted language, YAML, that expresses the desired state of a resource, but does not compute on it directly. The goal is to be declarative, that is,

the YAML should represent (“declare”) the desired state of the system. YAML can be used directly or it can be easily generated by tools, and we can provide smart tools for merging changes when there are multiple actors involved in configuration (which becomes common in larger systems).

We still need to enable automation, but rather than use code in the configuration, we use background agents, called controllers, to perform automation as needed. The basic approach is called reconciliation, and the job of the controller is to make adjustments to the running system (such as adding a new pod) such that the actual state is brought in line (“reconciled”) with the desired state. This is a robust form of eventual consistency — it may take some time for reconciliation to complete, but when failures cause the actual state to deviate from the (unchanged) desired state, the same process brings it back in line.

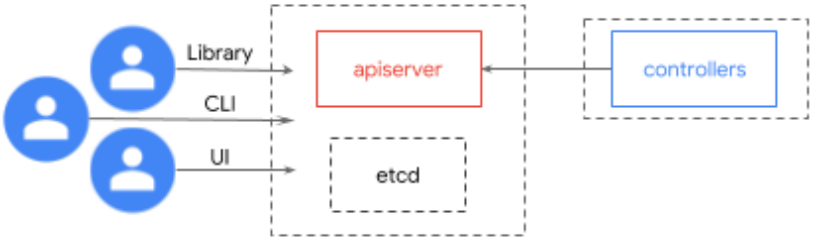

Figure 4: Kubernetes configuration model

Bringing it all together, the YAML file is the intended state, and it is managed using a RESTful interface to the apiserver (as shown below). The apiserver can be driven by different forms of UI, or by other tools, and it stores the desired state in a durable key/value store (currently based on etcd). Controllers watch for changes in the desired state and then take action accordingly.

One of the advantages of this declarative model is that it works extremely well with modern version-control systems like git: the config files are treated like source

code, and can and should be managed with the same notions of versioning, testing, and rollout. This system is very general and very consistent; for example, there is common metadata for the config files, and well-defined semantics for updates, validation, and deletion of resources. The resource business logic resides inside the controllers, while the rest is quite generic and thus reusable and extensible.

For example, developers can create a "custom resource definitions" (CRD) that defines the config schema for a new resource, and a controller that implements the resource's behavior. The Kubernetes API server machinery can manage new resources as service endpoints without requiring any modification.

Policy as Code

This extensible, declarative model enables automated configuration management for services running in Kubernetes and the cloud. Istio policy specifications (YAML-based CRDs) can be enforced via managed controllers that automate policy updates to proxies. "Policy as code" and automated CI/CD pipelines ensure progressive rollouts and improved governance, such as audits, compliance, and cost control.

Declarative policies help scale diverse cloud environments and ensure consistency. Namespaces provide a **logical service isolation boundary** that are not bound by proprietary implementations. Namespaces also allow policy admins to set up guardrails and delegate tenant-specific (local) policies. Additionally, they can be grouped hierarchically for policy inheritance and used with metadata labels to enforce cloud tenant policies. Finally, namespaces enable 'bring your own' and federation of identities for user/service authentication and authorization. As enterprises embrace zero-trust access models for their cloud environments, namespaces help ensure consistent programmatic access controls and dynamic enforcement of user and service identities across existing and future environments.

With Policy as Code, enforcing proprietary data stewardship policies are a mandatory component of service definition and activation.

Conclusion

The key to modern development is to decouple teams so that they can be more productive. Decoupling shows up in many forms, that together lead to faster evolution and better systems in less time and with less toil. We covered several forms of decoupling:

- Containers to decouple applications and libraries from the underlying OS and machine.
- Basic services in Kubernetes that decouple services from pods, allowing each to evolve independently
- Istio proxies to provide capabilities, including security and telemetry, across all services in a uniform way that is decoupled from each services' source code.
- Proxy-enabled policy deployment and enforcement decouples the management of policies from specific services so that policies can evolve without service disruption and security posture can be improved without infrastructure changes (via policy as code)
- The use of services as the unit of deployment that often correspond to specific teams, allowing teams to operate more independently. Although this may result in hundreds of services for a large-scale application, the Kubernetes and Istio service infrastructure makes it simple to manage.

By decoupling infrastructure, services and teams, enterprises can improve developer agility and accelerate innovation without compromising operational efficiency, security and governance.