Kong

BY MARCO PALLADINO

# The Difference Between API Gateways and Service Mesh

## Why API Management and Service Mesh are Complementary Patterns for Different Use Cases

# Content

# Introduction

For many years, API Management (APIM) — and the adoption of API gateways — was the primary technology used to implement modern API use cases both inside and outside the data center. API gateway technology has evolved a lot in the past decade, capturing bigger and more comprehensive use cases in what the industry calls "full lifecycle API management." It's not just the runtime that connects, secures and governs our API traffic on the data plane of our requests but also a series of functionalities that enable the creation, testing, documentation, monetization, monitoring and overall exposure of our APIs in a much broader context — and target a wider set of user personas from start to finish. That is, there is a full lifecycle of creating and offering APIs as a product to users and customers, not just the management of the network runtime that allows us to expose and consume the APIs (RESTful or not).

Then around 2017, another pattern emerged from the industry: service mesh. Almost immediately, the industry failed to recognize how this pattern played with the API gateway pattern, and a big cloud of confusion started to emerge. This was in part caused by the complete lack of thought leadership of pre-existing APIM vendors that have failed to respond adequately to how service mesh complemented the existing APIM use cases. It was also in part because service mesh started to be marketed to the broader industry by the major cloud vendors (first by Google, later by Amazon and finally by Microsoft) at such a speed that the developer marketing clout of this new pattern preceded the actual mainstream user adoption, therefore creating a misperception in the industry as to what service mesh really was (developer marketing) and was not (technology implementations). It was almost like a mystical pattern that everybody spoke about but very few mastered.

Over time, the technology implementations caught up with the original vision of service mesh, and more and more users implemented the pattern and told their stories. This allows us to now have a more serious rationalization as to what is service mesh (and what it is not) and what is the role of API gateways (and APIM) in a service mesh view of the world.

Many people have already attempted to describe the differences between API gateways and service meshes, and it's been commonly communicated that API gateways are for north-south traffic and service meshes are for east-west traffic. This is **not** accurate, and if anything, it underlines a fundamental misunderstanding of both patterns.

In this piece, I want to illustrate the differences between API gateways and service mesh — and when to use one or the other in a pragmatic and objective way.

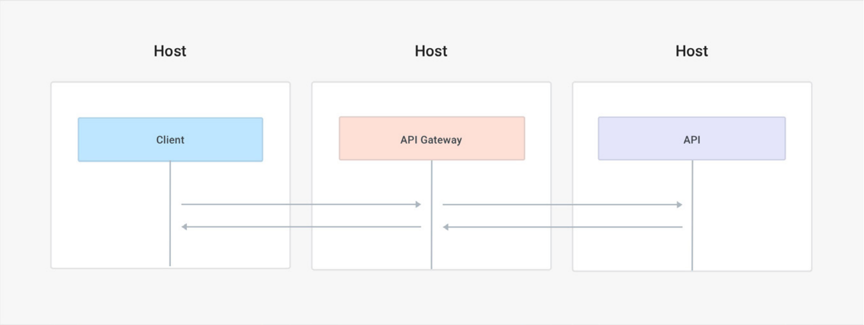
## API Gateways

The API gateway pattern describes an additional hop in the network that every request will have to go through in order to consume the underlying APIs. In this context, some people call the API gateway a centralized deployment.

Being on the execution path of every API request, the API gateway is a data plane that receives requests from a client and can enforce traffic and user policies before finally reverse proxying those requests to the underlying APIs. It can — and most likely will — also enforce policies on the response received from the underlying API before proxying the request back to the original client.

An API gateway can either have a built-in control plane to manage and configure what the data plane does, or both the data plane and the control plane can all be bundled together into the same process. While having a separate control plane is certainly better, some API gateway implementations were able to thrive with a DP+CP bundle in the same process because the number of API gateway nodes we would be deploying was usually of a manageable size and updates could be propagated with existing CI/CD pipelines.

The API gateway is deployed in its own instance (its own VM, host or pod) separate from the client and separate from the APIs. The deployment is therefore quite simple because it is fully separated from the rest of our system and it fully lives in its own architectural layer.



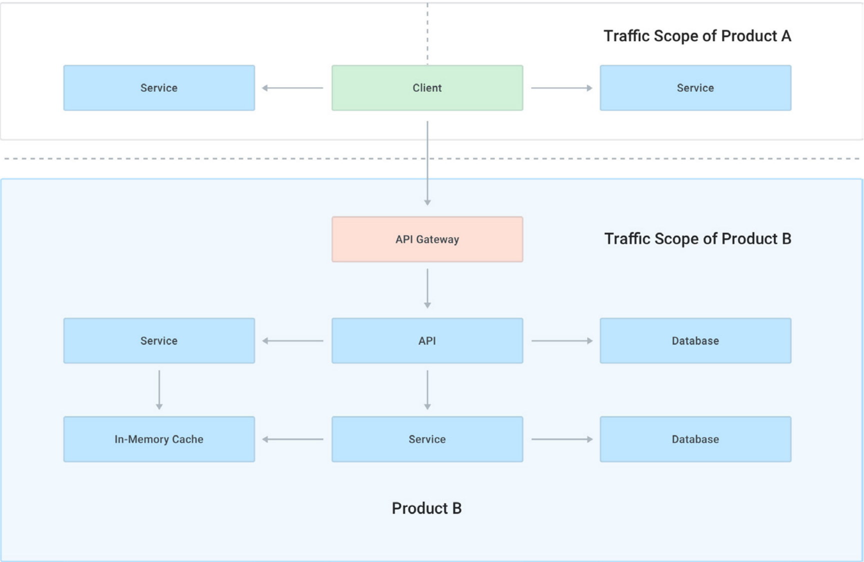
API gateways usually cover three primary API use cases for both internal and external service connectivity as well as for both north-south (outside the datacenter) and east-west (inside the datacenter) traffic.

## 1. APIs as a Product

The first use case is about packaging the API as a product that other developers, partners or teams will consume.

The client applications that they build can initiate requests from outside of the organization (like in the case of a mobile application) or from inside the same company (like in the case of another product, perhaps built by another team or another line of business). Either way, the client applications will run outside of the scope of the product (that's exposing the API) that they are consuming.

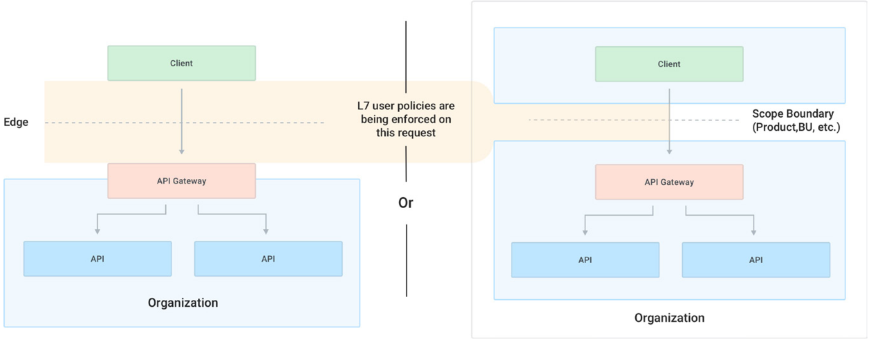
This use case is very common whenever different products/applications need to talk to each other, especially if they have been built by different teams.



When offering APIs as a product, an API gateway will encapsulate common requirements that govern and manage requests originating from the client to the API services — for example, AuthN/AuthZ use cases, rate-limiting, developer on-boarding, monetization or client application governance. These are higher level use cases implemented by **L7 user policies** that go above and beyond the management of the underlying protocol since they govern how the users will use the API product.

The APIs exposed by an API gateway are most likely running over the HTTP protocol (i.e., REST, SOAP, GraphQL or gRPC), and the traffic can be both north-south or east-west depending if the client application runs inside or outside the data center. A mobile application will run mostly north-south traffic to the API gateway, while another product within the organization could be running east-west traffic if it's being deployed in the same data center as the APIs it's consuming. The direction of traffic is fundamentally irrelevant.

API gateways are also used as an abstraction layer that allow us to change the underlying APIs over time without having to necessarily update the clients consuming them. This is especially important in those scenarios where the client applications are built by developers outside of the organization that cannot be forced to update to the greatest and latest APIs every time we decide to update them. In this instance, the API gateway can be used to keep the backwards compatibility with those client applications as our underlying APIs change over time.

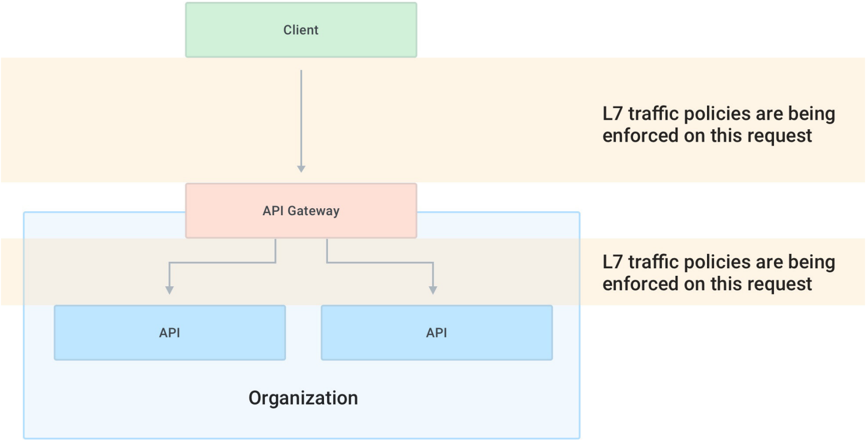

Client
Edge
API Gateway
API
API
Organization
L7 user policies are being enforced on this request
Or
Client
Scope Boundary (Product,BU, etc.)
API Gateway
API
API
Organization

## 2 - Service Connectivity

The **second use case** is about enforcing networking policies to connect, secure, encrypt, protect and observe the network traffic between the client and the API gateway, as well as between the API gateway and the APIs. They can be called **L7 traffic policies** because they operate on the underlying network traffic as opposed to governing the user experience.

Once a request is being processed by the API gateway, the gateway itself will have to then make a request to the underlying API in order to get a response (the gateway is, after all, a reverse proxy). Usually we want to secure the request via mutual TLS, log the requests, and overall protect and observe the networking communication. The gateway also acts as a load balancer and will implement features like HTTP routing, support proxying the request to different versions of our APIs (in this context, it can also enable blue/green and canary deployments use cases), as well as fault injection and so on.

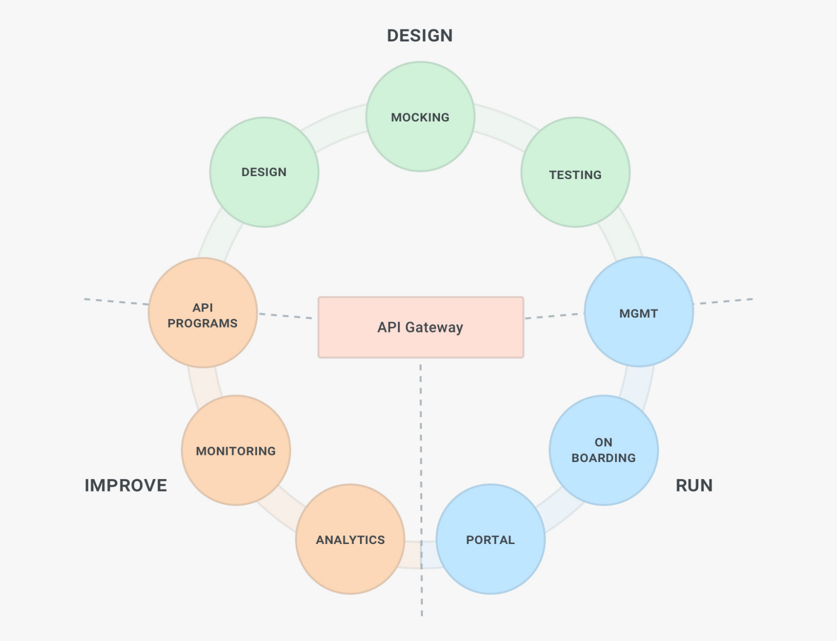
The underlying APIs that we are exposing through the API gateway can be built in any architecture (monolithic or microservices) since the API gateway makes no assumption as to how they are built as long as they expose a consumable interface. Most likely the APIs are exposing an interface consumable over HTTP (i.e., REST, SOAP, GraphQL or gRPC).


Client
L7 traffic policies are being enforced on this request
API Gateway
L7 traffic policies are being enforced on this request
API
API
Organization

### 3. Full Lifecycle API Management

The **third use case** of an API gateway is being one piece of a larger puzzle in the broader context of API management.

As we all know, managing the APIs, their users and client applications, and their traffic at runtime are only some of the many steps involved in running a successful API strategy. The APIs will have to be created, documented, and tested and mocked. Once running, the APIs will have to be monitored and observed in order to detect anomalies in their usage. Furthermore, when offering APIs as a product, the APIs will have to provide a portal for end users to register their applications, retrieve the credentials and start consuming the APIs.



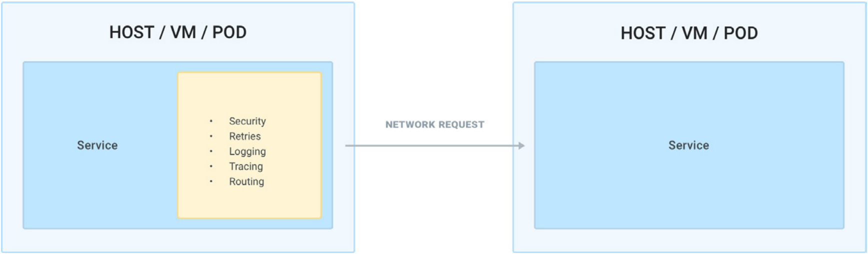
This broader experience, which is end-to-end and touches various points of the API lifecycle (and most likely different personas will be responsible for different parts of the lifecycle), is called full lifecycle API management, and effectively most APIM solutions provide a bundled solution to implement all of the above concerns in one or more products that will in turn connect to the API gateway to execute policy enforcement.

## Service Mesh

With service mesh, we are identifying a pattern that fundamentally improves how we build service-to-service connectivity among two or more services running in our systems. Every time a service wants to make a network request to another service (for example, a monolith consuming the database or a microservice consuming another microservice), we want to take care of that network request by making it more secure and observable, among other concerns.

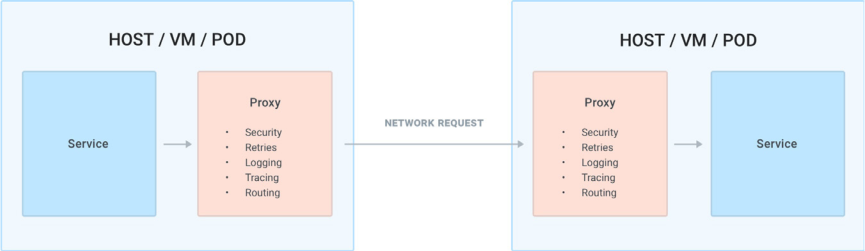
Service mesh as a pattern can be applied on **any architecture** (i.e., monolithic or microservice-oriented) and on any platform (i.e., VMs, containers, Kubernetes).

In this regard, service mesh does not introduce new use cases, but it better implements existing use cases that we already had to manage prior to introducing service mesh. Even before implementing service mesh, the application teams were implementing traffic policies like security, observability and error handling within their applications so they could enhance the connectivity of any outbound — or inbound — network requests that their application would either make or receive. The application teams were implementing these use cases by writing more code in their services. This means that different teams would be re-implementing the same functionality over and over again — and in different programming languages, creating fragmentation and security risks for the organization in managing the networking connectivity.



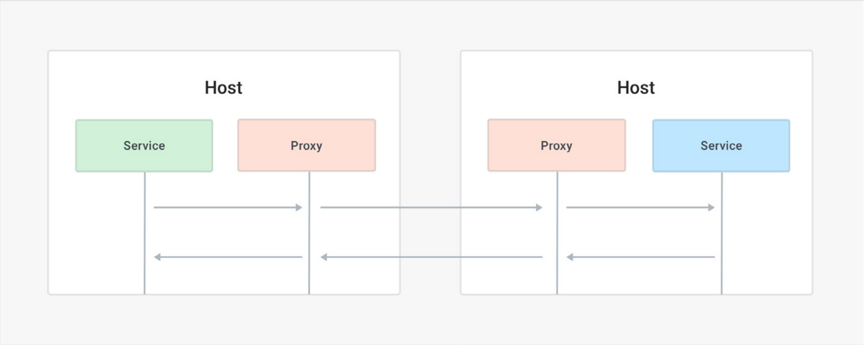
*Prior to service mesh, the teams are writing and maintaining code to manage the network connectivity to third-party services. Different implementations will exist to support different languages/frameworks.*

With the service mesh pattern, we are outsourcing the network management of any inbound or outbound request made by any service (not just the ones that we build but also third-party ones that we deploy) to an out-of-process application (the proxy) that will manage every inbound and outbound network request for us, and because it lives outside of the service, it is by default portable and agnostic in order to support any service written in any language or framework. The proxy will be on the execution path of every request and it's therefore a data plane process, and since one of the use-cases is implementing end-to-end mTLS encryption and observability, we would run one instance of the proxy alongside every service so that we can seamlessly implement those features without requiring the application teams to do too much work and abstracting those concerns away from them.

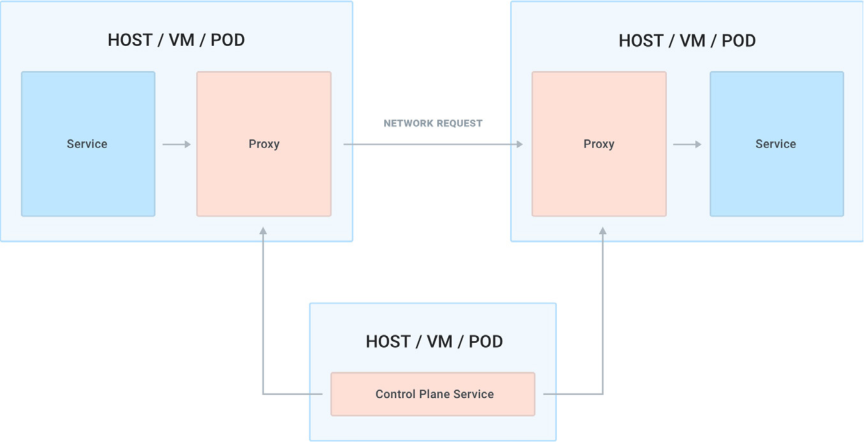


*We run one instance of the proxy (in peach) alongside every instance of our services.*

Because the data plane proxy will run alongside every replica of every service, some will call service mesh a decentralized deployment (as opposed to the API gateway pattern, which is a centralized deployment). Also, since we are going to be having extra hops in the network and in order to keep the latency at a minimum, we would run the data plane proxy on the same machine (VM, host, pod) as the service that we are running. Ideally, if the benefits of the proxy are valuable enough and the latency low enough, the equation would still turn in favor of having the proxying as opposed to having fragmentation in how the organization manages the network connectivity among our services.



The proxy application acts as both a proxy when the request is outgoing and as a reverse proxy when the request is incoming. Because we are going to be running one instance of the proxy application for each replica of our services, we are going to be having many proxies running in our systems. In order to configure them all, we would need a control plane that acts as the source of truth for the configuration and behavior we want to enforce and that would connect to the proxies to dynamically propagate the configuration. Because the control plane only connects to the proxies, it is not on the execution path of our service-to-service requests.



The service mesh pattern, therefore, is more invasive than the API gateway pattern because it requires us to deploy a data plane proxy next to each instance of every service, requiring us to update our CI/CD jobs in a substantial way when deploying our applications. While there are other deployment patterns for service mesh, the one described above (one proxy per service replica) is considered to be the industry standard since it guarantees the best, highest availability and allows us to assign a unique identity (via a mTLS certificate) to every replica of every service.

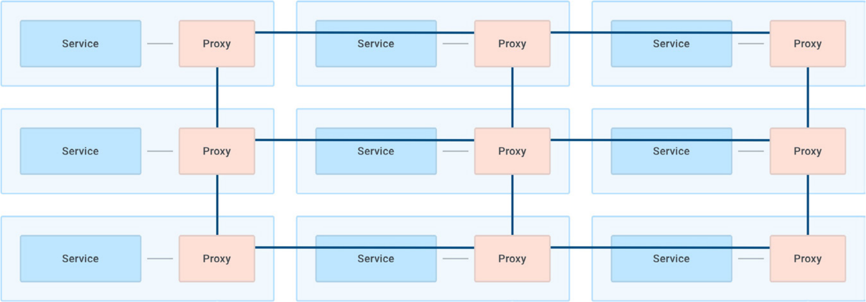
With service mesh, we are fundamentally dealing with one primary use case.

### 1. Service Connectivity

By outsourcing the network management to a third-party proxy application, the teams can avoid implementing network management in their own services. The proxy can then implement features like mutual TLS encryption, identity, routing, logging, tracing, load-balancing and so on for every service and workload that we deploy, including third-party services like databases that our organization is adopting but not building from scratch.

Since service connectivity within the organization will run on a large number of protocols, a complete service mesh implementation will ideally support not just HTTP but also any other TCP traffic, regardless if it’s north-south or east-west. In this context, service mesh supports a broader range of services and implements **L4/L7 traffic policies**, whereas API gateways have historically been more focused on L7 policies only.

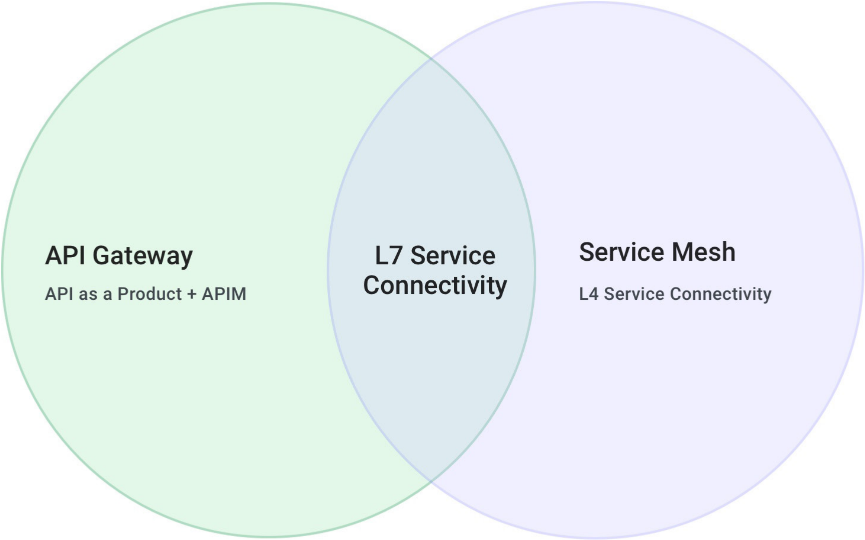
From a conceptual standpoint, service mesh has a very simple view of the workloads that are running in our systems: everything is a service, and services can talk to each other. Because an API gateway is also a service that receives requests and makes requests, an API gateway would just be a service among other services in a mesh.



Because every replica of every service requires a data plane proxy next to it and the data plane proxies are effectively client load-balancers so they can route outgoing requests to other proxies (and therefore other services), the control plane of a service mesh must know the address of each proxy so that the L4/L7 routing capability can be performed. The address can be associated with any meta-data, like the service name. By doing so, a service mesh essentially provides a built-in service discovery that doesn't necessarily require a third-party solution. A service discovery tool can still be used to communicate outside of the mesh but most likely not for the traffic that goes inside the mesh.

## API Gateway vs. Service Mesh

It is clear by looking at the use cases that there is an area of overlap between API gateways and service meshes, and that is the service connectivity use case.



The service connectivity capabilities that service mesh provides are conflicting with the API connectivity features that an API gateway provides. However, because the ones provided by service mesh are more inclusive (L4 + L7, all TCP traffic, not just HTTP and not just limited to APIs but to every service), they are in a way more complete. But as we can see from the diagram above, there are also use cases that service mesh does not provide, and that is the “API as a product” use case as well as the full API management lifecycle, which still belong to the API gateway pattern.

Since service mesh provides all the service connectivity requirements for a broader range of use-cases (L4+L7), it is natural to think that it would take over those concerns away from the API gateway (L7 only). This conclusion is valid only if we can leverage the service mesh deployment model, and as we will explore, this is not always the case.

One major divergent point between the two patterns is indeed the deployment model: in a service mesh pattern, we must deploy a proxy data plane alongside every replica of every service. This is easy to do when a team wants to deploy service mesh within the scope of its own product, or perhaps its own line of business, but it gets harder to implement when we want to deploy the proxy outside of that scope for four reasons:

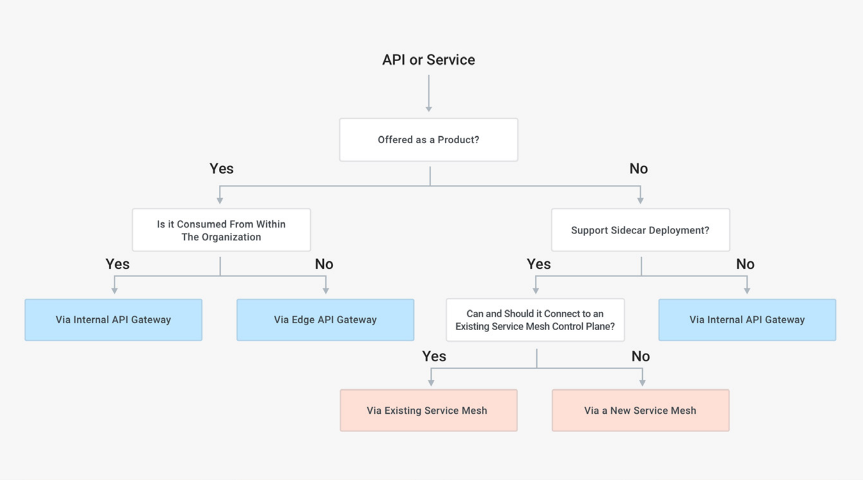
1. Deploying a proxy application alongside every service of every product within the organization can be met with resistance, since different products, teams and lines of business may have fundamentally different ways to build, run and deploy their software.
2. Every data plane proxy must initiate a connection to the control plane, and in certain cases, we don’t want -- or we can’t -- grant access to the control plane from services that are deployed outside of the boundaries of a product, a team or a line of business within the organization.

3. It is not possible to deploy the proxy data plane alongside every service because we do not control all the services in the first place, like in the case of a third-party application built by a developer, customer or partner that is external to the organization.
4. Services deployed in the same service mesh will have to use the same CA (Certificate Authority) in order to be provided with a valid TLS certificate to consume each other, and sharing a CA may not be possible or desirable among services that belong to different products or teams. In this instance, two separate service meshes (each one with its own CA) can be created, and they can communicate to each other via an intermediate API gateway.

Given that API gateways and service meshes focus on different use cases, I propose the following cheat sheet to determine when to use an API gateway and when to use a service mesh, with the assumption that in most organizations, both will be used since both use cases (the product/user use cases and the service connectivity one) will have to be implemented.

## Cheat Sheet

It is clear by looking at the use cases that there is an area of overlap between API gateways and service meshes, and that is the service connectivity use case.


API or Service
Offered as a Product?
Yes
No
Is it Consumed From Within The Organization
Support Sidecar Deployment?
Yes
No
Yes
No
Via Internal API Gateway
Via Edge API Gateway
Can and Should it Connect to an Existing Service Mesh Control Plane?
Via Internal API Gateway
Yes
No
Via Existing Service Mesh
Via a New Service Mesh

We will use an API gateway to offer APIs “as a product” to internal or external clients/users via a centralized ingress point and to govern and control how they are being exposed and on-boarded via a full lifecycle APIM platform. Commonly used when different applications need to talk to each other, and also used to create an abstraction layer between the clients and the underlying APIs.

We will use service mesh to build reliable, secure and observable L4/L7 traffic connectivity among all the services that are running in your systems via a decentralized sidecar deployment model that can be adopted and enforced on every service. Commonly used within the scope of an application, and to create point-to-point connectivity among all the services that belong to the application.

Most likely, the organization will have both of these use cases, and therefore an API gateway and service mesh will be used simultaneously.

## Example: A Financial Institution

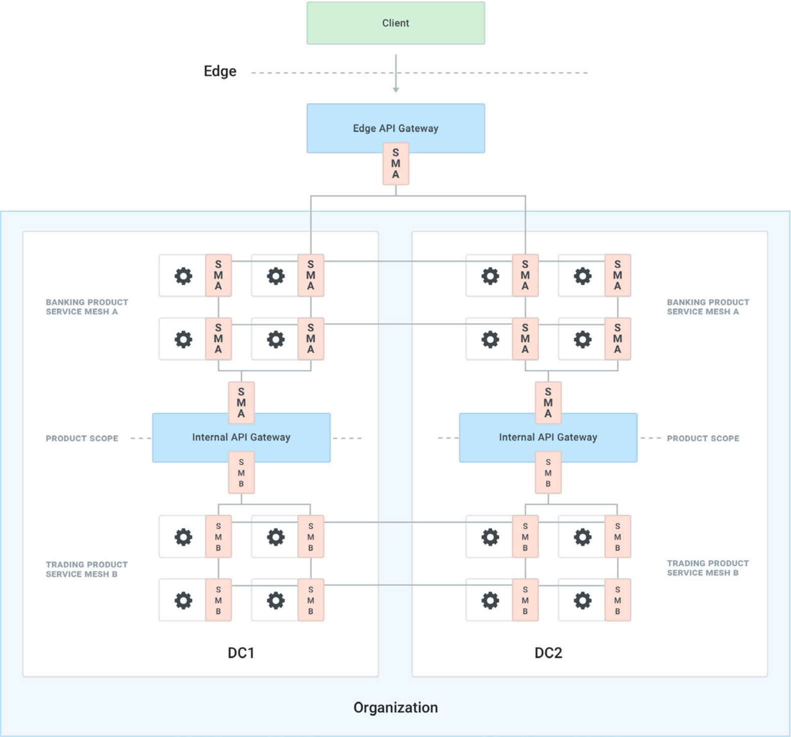
Given the chart above, we can provide the following example.

It is very common for an organization to have different teams building different products, and these products will have to talk to each other (i.e., a financial institution would have a “banking product” to perform banking activities and a “trading product” that would allow trading on the stock market, but the two products will have to communicate to share information between them).

These teams will also decide at one point in the roadmap to implement service mesh in order to improve the service connectivity among the services that are making up the final product. Because different teams run at different speeds, they will implement two service meshes that are isolated from each other: “Service Mesh A” and “Service Mesh B.”

Let's assume that in order to be highly available, both products are being deployed on two different data centers, "DC1" and "DC2."

The banking team wants to offer its service as a product to their internal customer, the trading team. Therefore they want to set up policies in place to on-board the team as if it was an external user via an internal API gateway. The mobile team also will have to consume both products, and they will have to go through an edge API gateway ingress point in order to do that. The architecture would look like this:



## About the Author

Marco Palladino is an inventor, software developer and entrepreneur. He is the CTO and co-founder of Kong, the most widely adopted open source API platform.

Kong provides API gateway and service mesh products via Kong Gateway and Kuma, both open source and freely downloadable.

Trusted by startups to Fortune 500 enterprises, Kong offers the leading service control platform that gives technology teams the architectural freedom to power connections for modern software architectures and applications across clouds. Kong's customers span across all industries, including Cargill, WeWork, SoulCycle, Yahoo! Japan, Verifone and Just Eat.

Kong

Konghq.com

**Kong Inc.**
contact@konghq.com

150 Spear Street, Suite 1600
San Francisco, CA 94105
USA