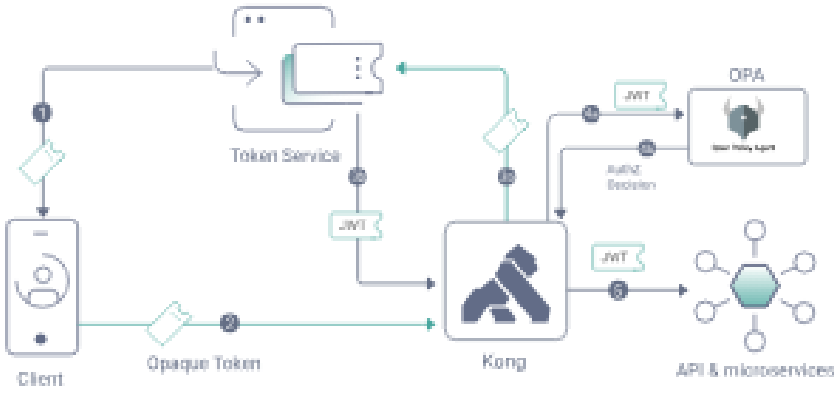
# API Gateway

## Kong

K

## Kong and OPA




API Authorization using Open Policy Agent and Kong

### Token Service

Kong and Azure AD/ Entra ID

## Service-to-service Auth

For service-to-service (also known as machine-to-machine) authentication and authorization, OAuth 2.0 is generally the more appropriate protocol to use. Here's why:

#### OAuth 2.0 for Service-to-Service

**OAuth 2.0** is primarily designed for authorization and provides mechanisms to grant limited access to resources on behalf of a user or service. It supports several grant types, and for service-to-service communication, the **Client Credentials Grant** is commonly used.

##### Why OAuth 2.0?

1. **Authorization**: OAuth 2.0 is designed for scenarios where a service needs to access another service's resources. It allows you to grant and manage access permissions.
2. **Token-Based**: It uses access tokens to authorize requests, which can be scoped to provide fine-grained access control.
3. **Granularity**: You can define scopes that precisely specify what resources and operations the requesting service can access.
4. **Flexibility**: OAuth 2.0 is flexible and widely supported, making it easy to integrate with many different systems and services.

#### OIDC for Service-to-Service?

**OIDC** (OpenID Connect) builds on top of OAuth 2.0 to add authentication capabilities, primarily intended for user authentication and obtaining user profile information. It is less suited for pure service-to-service interactions where user identity verification is not needed.

##### Why Not OIDC?

1. **User-Centric**: OIDC is designed for scenarios where user authentication and identity information are required, which is not typically needed in service-to-service communications.
2. **Additional Overhead**: OIDC introduces additional overhead (ID tokens, userinfo endpoint) that is unnecessary for pure service-to-service interactions.

### Typical OAuth 2.0 Client Credentials Grant Flow

Here's how the **Client Credentials Grant** works in OAuth 2.0 for service-to-service communication:

1. **Service A (Client) Requests a Token**:
   - Service A sends a POST request to the authorization server's token endpoint with its client ID and client secret.

```
1 http
```

Copy code

```
POST /token Host: authorization-server.com Content-Type: application/x-www-form-urlencoded
grant_type=client_credentials&client_id=CLIENT_ID&client_secret=CLIENT_SECRET&scope=SCOPE
```

2. **Authorization Server Issues a Token**:
   - The authorization server validates the client credentials and the requested scopes. If valid, it responds with an access token.

```
1 json
```

Copy code

```
{ "access_token": "ACCESS_TOKEN", "token_type": "Bearer", "expires_in": 3600, "scope": "SCOPE" }
```

3. **Service A Uses the Token to Access Service B**:
   - Service A includes the access token in the Authorization header of its requests to Service B.

```
1 http
```

Copy code

```
GET /resource Host: resource-server.com Authorization: Bearer ACCESS_TOKEN
```

4. **Service B Validates the Token**:
   - Service B validates the access token. If valid, it allows the request to access the requested resource.

### Summary

- **OAuth 2.0** is suitable for service-to-service communication due to its focus on authorization and token-based access control.
- **Client Credentials Grant** is the recommended OAuth 2.0 flow for service-to-service scenarios.
- **OIDC** is more suited for user authentication scenarios and is not necessary for pure service-to-service interactions.

Using OAuth 2.0 with the Client Credentials Grant will provide a secure and efficient mechanism for service-to-service authentication and authorization.

OpenID Connect (OIDC) can be used for service-to-service authentication. Although OIDC is commonly associated with user authentication, it can also facilitate secure communication between services without direct user interaction. This is typically done through the use of OAuth 2.0 client credentials flow, which is part of the broader OAuth 2.0 framework that OIDC builds upon.

OIDC VS OAth2

OpenID Connect (OIDC) is an open standard that builds on top of OAuth 2.0, a framework for authorizing access to web services. OAuth 2.0 allows a user to grant permission to a third-party application (called a client) to access a web service (called a resource server) on their behalf, without sharing their credentials. However, OAuth 2.0 does not provide a way for the client to obtain information about the user's identity or attributes, such as their name, email, or role. This is where OpenID Connect comes in. OpenID Connect extends OAuth 2.0 by adding an identity layer that enables the client to request and receive an identity token (called an ID token) that contains information about the user (called claims). The ID token is issued by an identity provider (called an OP), which is a web service that authenticates the user and manages their identity data.

How can you use OpenID Connect for web service authentication?

## API Gateway vs Service Mesh

**API As a Product - is about packaging the API as a product that other developers, partners, or teams will consume**

Around 2017, the concept of service mesh emerged, creating confusion in the industry about its relationship with the API gateway pattern. This confusion was partly due to the lack of guidance from existing API management (APIM) vendors, who didn't adequately address how service mesh complemented existing use cases. Additionally, major cloud vendors like Google, Amazon, and Microsoft quickly promoted service mesh, leading to misconceptions about its role versus actual technological implementations. Initially, service mesh was widely discussed but poorly understood and implemented. Over time, as implementations improved and user experiences were shared, the industry gained a clearer understanding of service

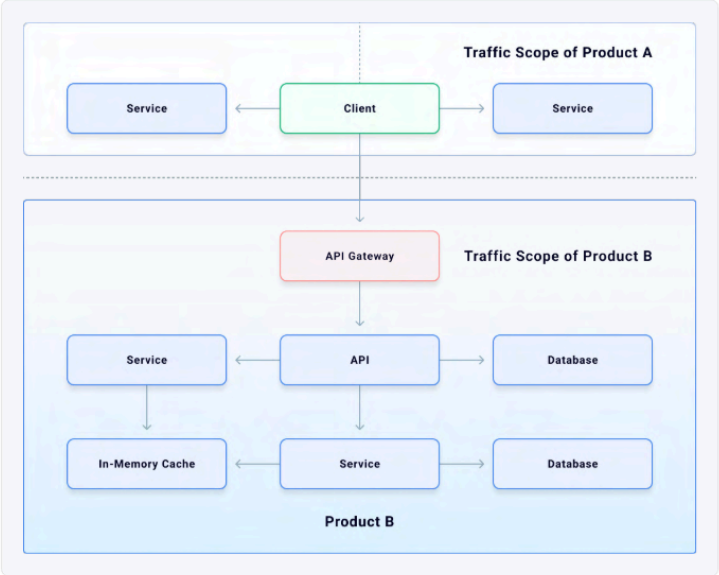
mesh and its role alongside API gateways. The common distinction that API gateways handle north-south traffic and service meshes handle east-west traffic is overly simplistic and highlights a misunderstanding of both patterns.

An API gateway can either have a built-in control plane to manage and configure the data plane or bundle both the data plane and control plane into the same process. While having a separate control plane is generally preferable, some API gateway implementations have successfully operated with a combined DP+CP model due to the manageable number of nodes and the ability to update via existing CI/CD pipelines. Nevertheless, the API management landscape has largely adopted the separation of the data plane from the control plane to enhance resiliency, security, and manageability.

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.

The client applications 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.

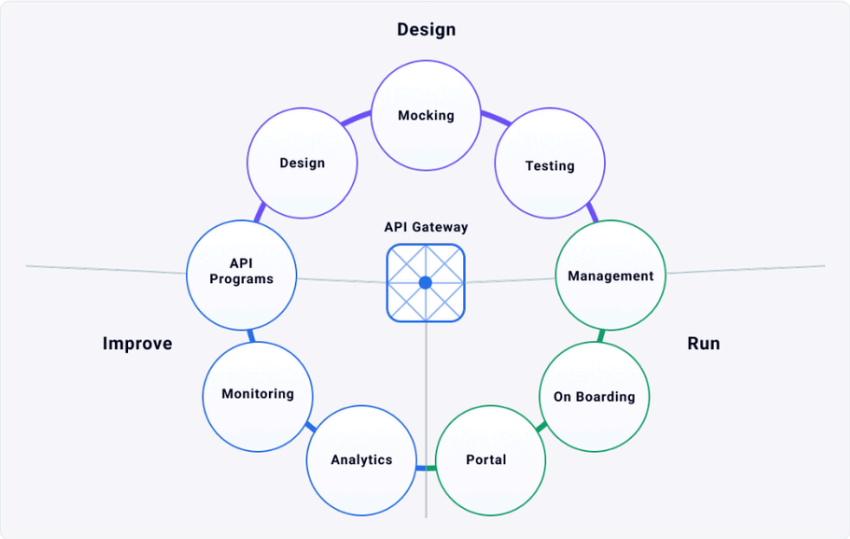
**When offering APIs as a product, an API gateway will encapsulate common requirements that govern and manage requests originating from the client to the API services.**




**Service Connectivity - 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

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, tested, and mocked. Once running, the APIs will have to be monitored and observed 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.




## Service Mesh

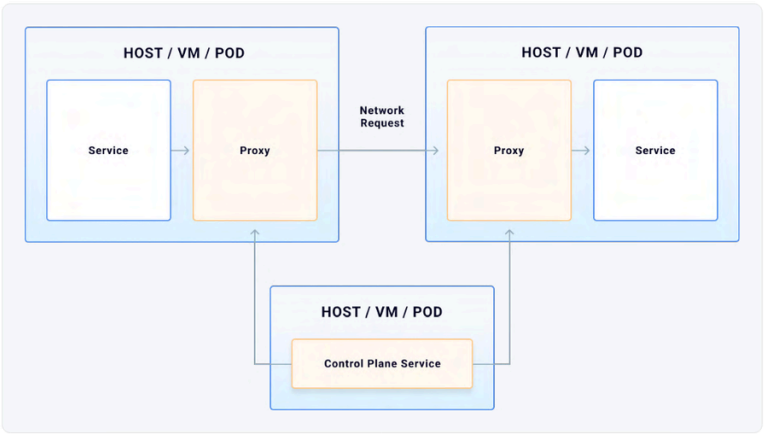
Every time a service makes a network request to another service (e.g., a monolith accessing a database or a microservice calling another microservice), it's important to ensure these requests are secure and observable. The service mesh pattern enhances these aspects and can be applied to any architecture (monolithic or microservice) and any platform (VMs, containers, Kubernetes).

Service mesh doesn't introduce new use cases but improves the implementation of existing ones like security, observability, and error handling. Previously, application teams managed these concerns within their applications, often duplicating effort and creating fragmentation and security risks.

With a service mesh, network management for inbound and outbound requests is outsourced to an out-of-process proxy. This proxy handles every request, making it portable and agnostic, supporting any service in any language or framework. The proxy, acting as the data plane, runs alongside each service instance to provide features like end-to-end mTLS encryption and observability, abstracting these complexities away from the application teams.

The data plane proxy in a service mesh runs alongside every replica of every service, making it a decentralized deployment pattern, unlike the centralized API gateway pattern. To minimize latency, the data plane proxy is deployed on the same machine (VM, host, pod) as the service it supports. Despite the additional network hops, the benefits of using the proxy—such as avoiding fragmentation in network management—often outweigh the latency costs.

The proxy acts both as a proxy for outgoing requests and as a reverse proxy for incoming requests, leading to multiple proxies running across the system. To manage and configure these proxies, a control plane is used. The control plane serves as the source of truth for configurations and dynamically propagates them to the proxies. Importantly, the control plane is not on the execution path of service-to-service requests, ensuring it does not add latency to the network interactions.

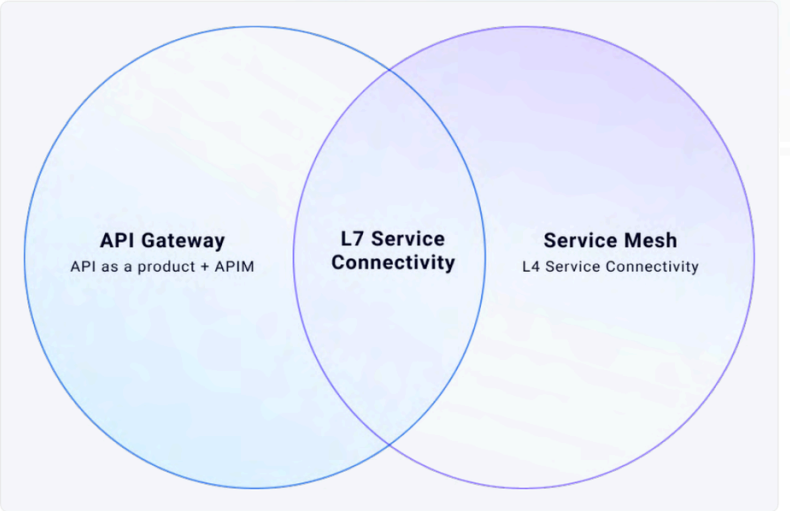


Service connectivity within an organization often involves multiple protocols. Therefore, an ideal service mesh implementation supports not just HTTP but all TCP traffic, whether it's north-south or east-west. This broad support allows service mesh to cater to a wider range of services and implement both L4 and L7 traffic policies.

In contrast, API gateways have historically focused primarily on L7 policies.

## API Gateway vs. Service Mesh

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




| | API Gateway | Service Mesh |
|---|---|---|
| • **Resilience** | With either or both technologies in place, your application can recover quickly from difficulties or failures encountered in your cloud-native application. | With either or both technologies in place, your application can recover quickly from difficulties or failures encountered in your cloud-native application. |
| • **Traffic management** | Without a service mesh or API gateway, the traffic from API calls made by clients would be difficult to manage. This will eventually delay the request processing and response time.<br>• Client-side discovery: The client is responsible for requesting and selecting available network services in the API gateway and service mesh. | Without a service mesh or API gateway, the traffic from API calls made by clients would be difficult to manage. This will eventually delay the request processing and response time.<br>• Client-side discovery: The client is responsible for requesting and selecting available network services in the API gateway and service mesh. |

| | | |
|---|---|---|
| • Client-side discovery | The client is responsible for requesting and selecting available network services in the API gateway and service mesh. | The client is responsible for requesting and selecting available network services in the API gateway and service mesh. |
| • **Service discovery** | Both technologies facilitate how applications and microservices can automatically locate and communicate with each other. | Both technologies facilitate how applications and microservices can automatically locate and communicate with each other. |
| • **System observability** | Both technologies can manage services that clients can access. They also keep logs of clients that have accessed specific services. This helps to track the health of each API call made across to the microservice. | Both technologies can manage services that clients can access. They also keep logs of clients that have accessed specific services. This helps to track the health of each API call made across to the microservice. |
| • **Capabilities** | • API gateways serve as an edge microservice and perform tasks helpful to your microservice's business logic, like request transformation, complex routing, or payload handling | • service mesh only addresses a subset of inter-service communication problems. |
| **External vs. internal communication** | • The API gateway operates at the application level<br>• An API gateway stands between the user and internal application logic<br>• API gateways focus on business logic | • ervice mesh operates at the infrastructure level<br>• service mesh is between the internal microservices<br>• service mesh deals with service-to-service communication. |
| • **Monitoring and observability** | • API gateways can help you track the overall health of an application by measuring the metrics to identify flawed APIs | • service mesh metrics assist teams in identifying issues with the various microservices and components that make up an application's back end rather than the entire program<br>• Service mesh helps determine the cause of specific application performance issues. |
| • **Tooling and support** | • API gateways work with almost every application or architecture and can work with monolithic and microservice applications<br>• API gateways have automated security policies and features that are easy to start; | • Service mesh is typically designed only for specific environments, such as Kubernetes.<br>• service meshes often have complex configurations and processes with a steep learning curve. |

Microservices: Service Mesh vs API Gateway

## API Gateway Comparison

### Kong API Gateway vs Spring Cloud Gateway

| | **Kong API Gateway** | **Spring Cloud Gateway** |
|---|---|---|
| Summary: | 1. You need a high-performance, scalable gateway capable of handling a diverse range of protocols and high traffic volumes. | 1. Your development team is heavily invested in the Spring ecosystem and prefers using Java for customizations.<br>2. You are looking for an easy-to-use, lightweight API gateway |

| | | |
|---|---|---|
| | 2. You require advanced features and plugins out-of-the-box, including those for security, rate limiting, and monitoring.<br>3. Your organization can benefit from enterprise support and additional features provided by Kong's enterprise edition.<br>4. You are looking for a robust solution with service mesh integration capabilities. | solution that integrates well with Spring Cloud.<br>3. You need a solution that leverages existing Spring Cloud features like service discovery and configuration management.<br>4. Your traffic and performance requirements can be met within the capabilities of Spring Cloud Gateway. |
| Performance and Scalability | 1. Kong is built on NGINX and Lua, known for high performance and low latency.<br>2. Designed to handle high throughput and scale horizontally with ease. | 1. Performance might not match Kong's NGINX-based architecture for very high throughput scenarios.<br>2. May require more resources for high traffic loads.<br>3. While it scales well within Spring Cloud environments, scaling horizontally in large clusters might be more complex compared to Kong. |
| Plugin Ecosystem | 1. Rich set of plugins for authentication, rate limiting, logging, monitoring, and more.<br>2. Custom plugins can be written in Lua, making it highly extensible. | **Tight Integration with Spring Ecosystem:**<br>• Seamlessly integrates with other Spring projects (e.g., Spring Boot, Spring Security).<br>• Ideal for teams already using the Spring framework for their microservices. |
| Feature Set | | 1. Leverages Spring Cloud capabilities for service discovery, configuration, and resilience.<br>2. Integrated with Spring Boot Actuator for monitoring and health checks. |
| Multi-protocol Support | 1. Supports HTTP, HTTPS, TCP, and gRPC.<br>2. Flexibility to handle different types of traffic and services. | |
| Service Mesh Integration | 1. Integrates seamlessly with service meshes like Istio.<br>2. Can act as an Ingress controller for Kubernetes environments. | |
| Enterprise Features: | **Enterprise Features:**<br>• Offers an enterprise edition with additional features like advanced security, analytics, and developer portals.<br>• Support for role-based access control (RBAC), audit logs, and other enterprise needs. | |
| Open Source Community and Commercial Support: | 1. Large open-source community and extensive documentation.<br>2. Availability of commercial support and professional services. | |

| | | |
|---|---|---|
| Cost | While the open-source version is free, the enterprise version comes with licensing costs. | Completely open-source with no enterprise edition or additional costs. |

### Open Policy Agent (OPA)

**PEP - "Policy Enforcement Point":**

The PEP intercepts a user's access to a resource, and either grants or denies access to the resource requested by the user. PEPs don't make the decisions; they enforce them. PEPs can potentially also adjust requests or their results before passing them through (aka data-filtering). Examples of PEPs are in-code control flow (i.e. "if"), middleware, reverse proxies, and API gateways

**PDP - "Policy Decision Point":**

The PDP evaluates authorization requests against relevant policies and a data snapshot aggregated from the distributed data layer and makes an authorization decision.

**PAP - "Policy Administration Point":**

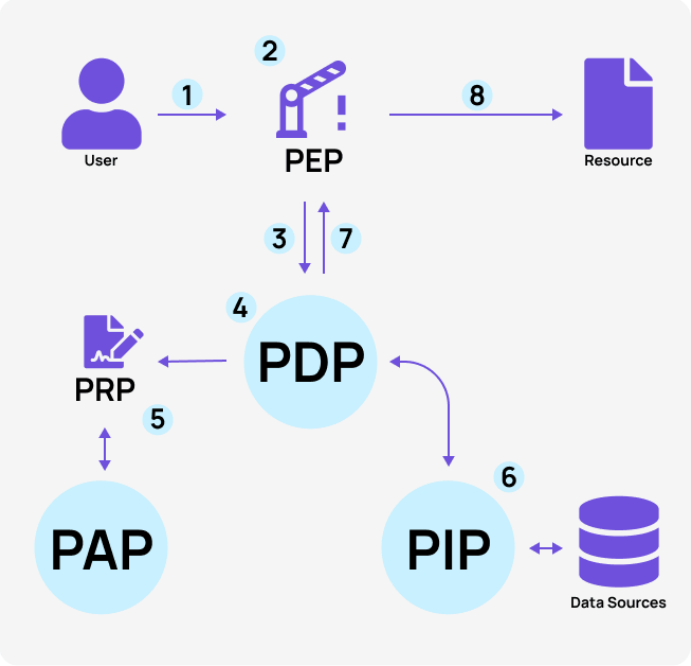
The PAP is in charge of managing all relevant policies to be used by the PDP.

**PIP - "Policy Information Point":**

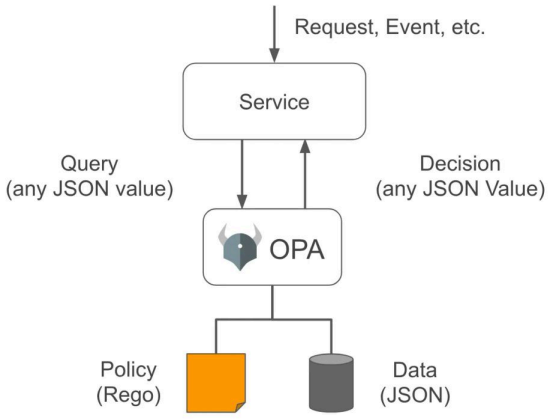
The PIP is any data source (Internal or external) that contains attributes that are relevant for making policy decisions.

**PRP - "Policy Retrieval Point":**

The PAP is the main source of policies - it stores all relevant policies to be used by the PDP, and is managed by the PAP (In OPA it's a bundle-server, and in OPAL this is often a Git repository or an HTTP bundle server).




OPA decouples policy decision-making from policy enforcement. When your software needs to make policy decisions it **queries** OPA and supplies structured data (e.g., JSON) as input. OPA accepts arbitrary structured data as input.




### Rego

OPA policies are expressed in a high-level declarative language called Rego. Rego (pronounced “ray-go”) is purpose-built for expressing policies over complex hierarchical data structures. For detailed information on Rego see the Policy Languagedocumentation.

### OPA vs Istio Authz

This table summarizes the different types of policies and where do we need OPA to implement:

| Group | Field | Istio Authz | OPA |
|---|---|---|---|
| HTTP | any | ✅ | ✅ |
| Kubernetes | source/ip | ✅ | ✅ |
| Kubernetes | source/principal | ✅ | ✅ |
| Kubernetes | destination/ip | ✅ | ✅ |
| Kubernetes | destination/principal | ✅ | ✅ |
| JWT | is the token valid | ✅ | ✅ |
| JWT | sub, iss, aud, azp | ✅ | ✅ |
| JWT | all other fields | ❌ | ✅ |
| HTTP Request Body | | ❌ | ✅ |
| Context | Data not present in the request | ❌ | ✅ |

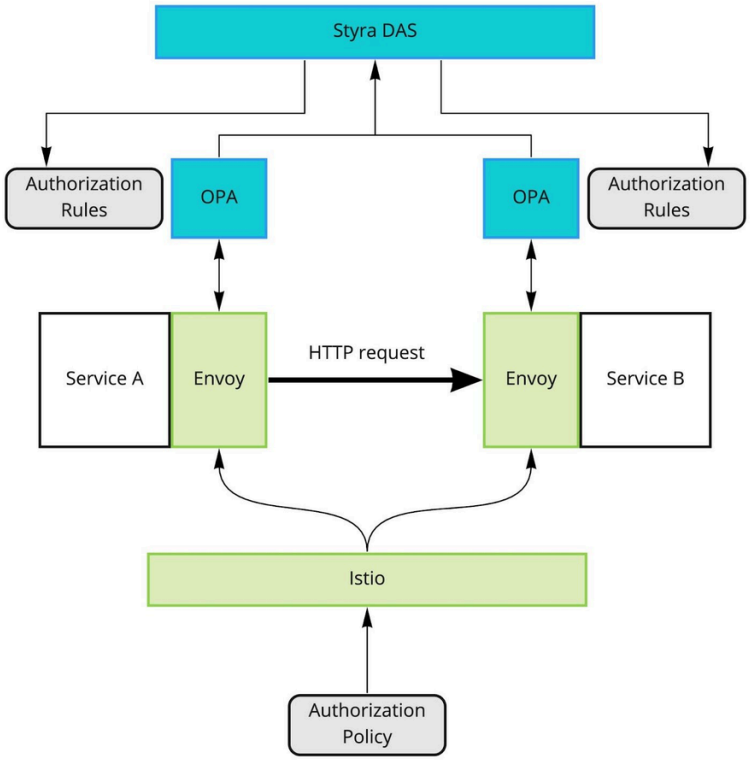
Authorize Better: Istio Traffic Policies with OPA & Styra DAS

## API Gateway, OPA and Service Mesh

Distributed OPA

Control plan - Managing a fleet of OPAs

Authorize Better: Istio Traffic Policies with OPA & Styra DAS


Styra DAS
Authorization Rules
OPA
OPA
Authorization Rules
Service A
Envoy
HTTP request
Envoy
Service B
Istio
Authorization Policy