tetrate

The current state and future of the **Istio Service Mesh**

September 2022

Table of contents

Executive Summary

Cloud native is eating the cloud and open source, and service mesh is still thriving as a critical part of the cloud native technology stack. Istio is one of the most popular service mesh today, and has been open source for over five years since 2017.

This book takes you through the historical motivation for the emergence of the service mesh, the evolution of Istio, and the Istio open source ecosystem. This book provides detailed information on

- The rise of service mesh technology is due to the popularity of Kubernetes, microservices, DevOps, and cloud native architectures.
- The emergence of Kubernetes and programmable proxies, which laid a solid foundation for Istio.
- While eBPF can accelerate transparent traffic hijacking in Istio, it cannot replace the sidecar in the service mesh.
- The future of Istio lies in building a zero trust network based on the hybrid cloud.

With the entry of Istio into CNCF and the introduction of the latest Ambient Mesh, we can expect that Istio will be easier to adopt, and its future will be more extensive.

Service mesh technology is on the rise due to the popularity of Kubernetes, microservices, DevOps, and cloud-native architectures.

Background and Introduction

This article reviews the development of Istio open source in the past five years and looks forward to the future direction of Istio. The main points of view in this article are as follows:

- Service mesh technology is on the rise due to the popularity of Kubernetes, microservices, DevOps, and cloud-native architectures.
- The rise of Kubernetes and programmable data proxies is the foundation of Istio.
- While eBPF can accelerate transparent traffic hijacking in Istio, it can not replace sidecars in service meshes.
- The future of Istio is to serve as the foundation forbuild a secure, zero-trust network.

The eve of the birth of Istio

Since 2013, with the explosion of the mobile Internet, enterprises have had higher requirements for the efficiency of application iteration. Application architecture has shifted from monolithic to microservices, and DevOps has also become popular. In the same year, with the open-source of Docker, the problems of application encapsulation and isolation were solved, making it easier to schedule applications in the orchestration system. In 2014, Kubernetes and Spring Boot were open-sourced, and Spring framework development of microservice applications became popular. In the next few years, many RPC middleware open source projects appeared, such as Google's gRPC 1.0, released in 2016. The service framework is in full bloom. To save costs, increase development efficiency, and make applications more flexible, more and more enterprises are migrating to the cloud, but this is not just as simple as moving applications to the cloud. To use cloud computing more efficiently, a set of "cloud native" methods and concepts are also on the horizon.

Istio open source timeline

Let's briefly review the major events of Istio open source:

September 2016

Since Envoy is an important part of Istio, we should start the timeline from Envoy. Before becoming open source, Envoy was used as an edge proxy inside Lyft, and it was verified in large-scale production inside Lyft. In fact, Envoy was open sourced before it was open sourced, and it got the attention of Google engineers. At that time, Google was planning to launch an open source service mesh project, initially planning to use Nginx as a proxy. In 2017, Envoy was donated to CNCF.

May 2017

Istio was open sourced by Google, IBM, and Lyft. The microservices architecture was used from the beginning. The composition of the data plane, control plane, and sidecar pattern was determined.

March 2018

Kubernetes successfully became the first project to graduate from CNCF, becoming increasingly "boring". The basic API has been finalized. In the second edition, CNCF officially wrote the service mesh into the cloud native first definition. The company, Tetrate was founded by the Google Istio team.

July 2018

Istio 1.0 is released, billed as "production ready".

March 2020

Istio 1.5 is released, the architecture returned to a monolithic application, the release cycle was determined, a major version was released every three months, and the API became stable.

From 2020 to the present:

The development of Istio mainly focuses on Day 2 operation, performance optimization, and extensibility. Several open source projects in the Istio ecosystem have begun to emerge, such as Slime, Areaki, and Merbridge.

Why did Istio come after Kubernetes?

The fundamental reasons for the emergence of service meshes are the increase in use of heterogeneous languages, the surge in the number of services, and the shortened life cycle of containers are the fundamental reasons for the emergence of service meshes.

To make it possible for developers to manage traffic between services with minimal cost, Istio needs to solve three problems:

1. Transparently hijack traffic between applications, which means that developers can quickly use Istio's capabilities without modifying applications.

2. Inject proxies into applications and efficiently manage the fleet of distributed sidecar proxies.

3. An efficient and scalable sidecar proxy that can be configured through an API.

The three problems are indispensable for the Istio service mesh, and two of them are directly related to the sidecar proxy, while the injection and management of the proxies is a concern for the service mesh. The choice of this proxy will directly affect the direction and success of the project.

To solve container orchestration, scheduling, and management, Istio relies on Kubernetes. The concerns of a programmable proxy are solved by the Envoy proxy.

From the figure below, we can see the transition of the service deployment architecture from Kubernetes to Istio.

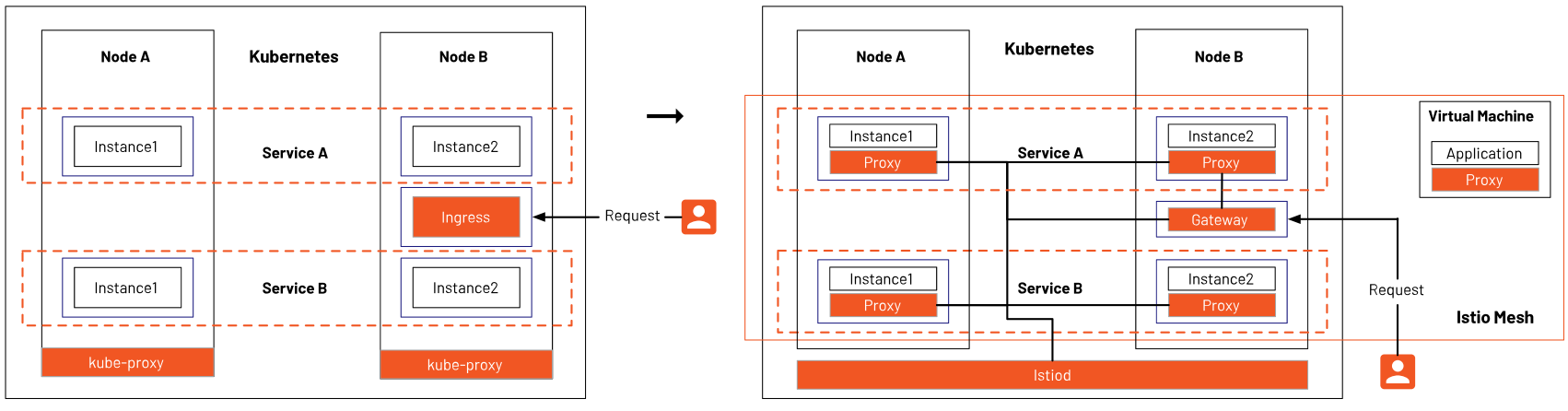

Figure 1. The architectural change from Kubernetes to Istio

From Kubernetes to Istio, in a nutshell, the deployment architecture of the application has the following characteristics:

- Kubernetes manages the life cycle of applications, specifically, application deployment and management (scaling, automatic recovery, and rollouts).
- Automatic sidecar injection using Kubernetes init container and sidecar mode to achieve transparent traffic interception. First, the inbound and outbound traffic of the service is intercepted through the sidecar proxy, and then the behavior of the proxy is managed through the control plane configuration. There's a proxyless mode for Istio, see gRPC proxyless service mesh for details.
- The service mesh decouples traffic management from Kubernetes, and the traffic inside the service mesh does not need the support of the kube-proxy component. Through an abstraction similar to the microservice application layer, the traffic between services is managed to achieve security and observability features.
- The control plane sends proxy configuration to the data plane through the xDS protocol. The proxies that have implemented xDS include Envoy and the open source MOSN project.
- Typically, the north-south traffic in Kubernetes is managed by the Kubernetes Ingress resource. With Istio, this has changed, and the traffic is managed by the Gateway resource. Note that Istio has support for the Ingress resource as well.

Transparent traffic hijacking

If you are using middleware such as gRPC to develop microservices, the interceptor in the SDK will automatically intercept the traffic for you, as shown in the following figure.

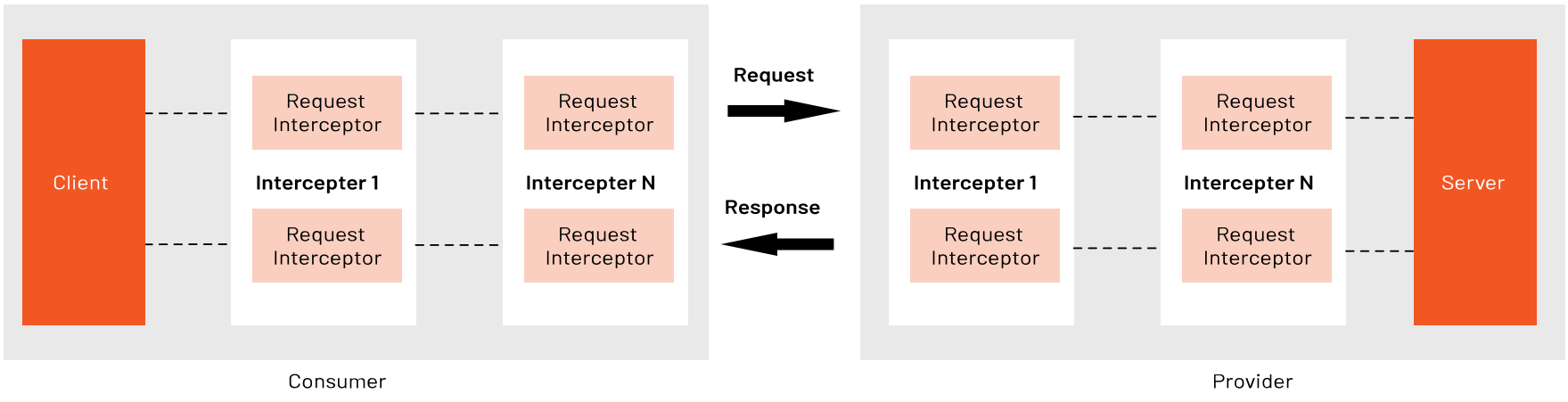

Figure 2. The Interceptor of gRPC

How to make the traffic in the Kubernetes pod go through the proxy? The answer is to inject a proxy into each application pod. Containers in the pod share the network space with the application, which allows us to hijack the inbound and outbound traffic and route it through the sidecar. The traffic is hijacked using iptables as shown in the figure below.

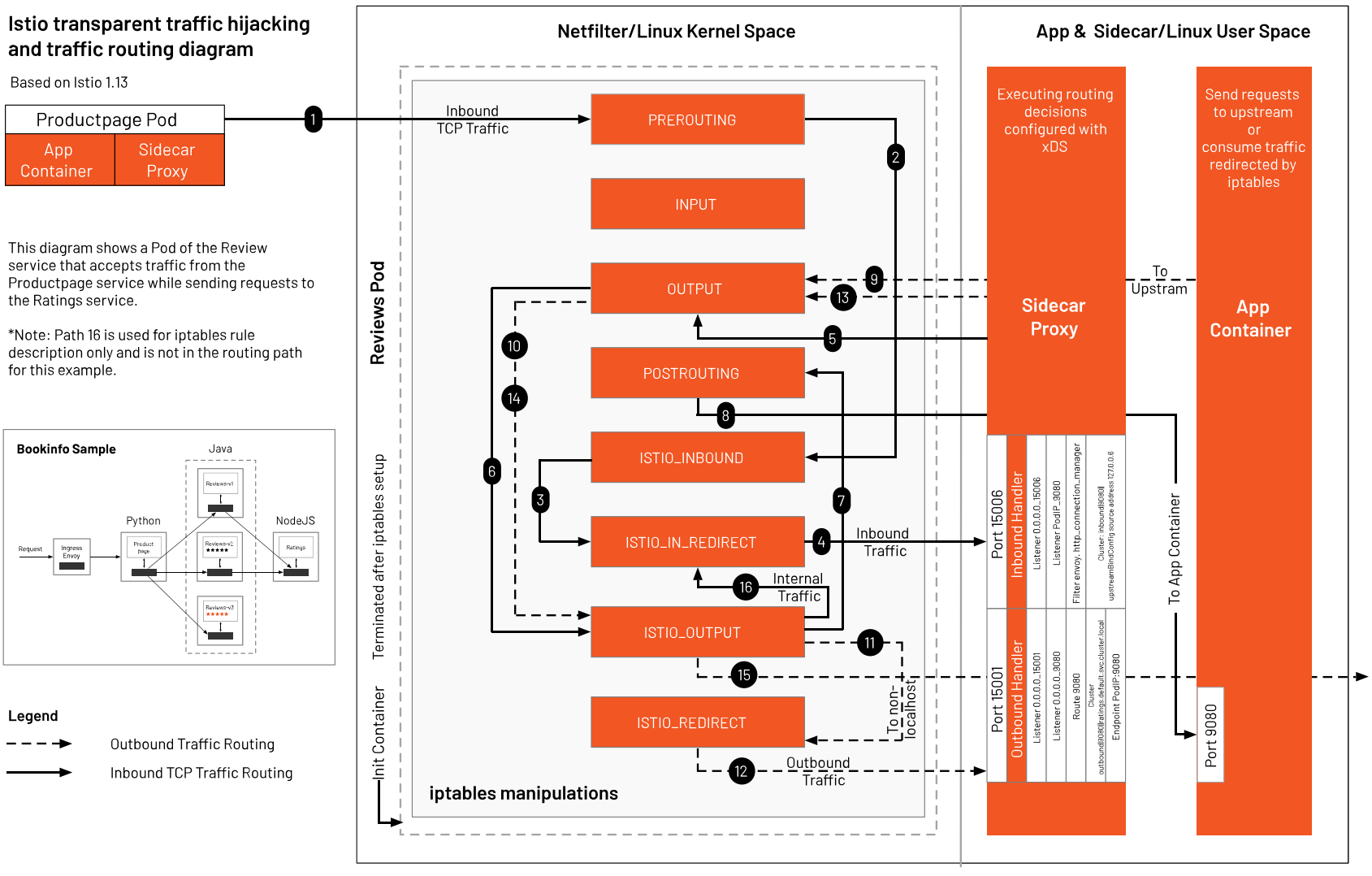

Figure 3. Transparent traffic hijacking in Istio

From the figure, we can see a very complex set of iptables traffic hijacking logic. The advantage of using iptables is that it applies to any Linux operating system. But this also has some side effects:

1. All services in the Istio mesh need to add a network hop when entering and leaving the pod. Although each hop may be only a couple of milliseconds, as the dependencies between services and services in the mesh increases, this latency may increase significantly, which may not be suitable for services that pursue low latency.
2. As the number of services increases, so does the number of injected sidecars. The control plane needs to deliver more Envoy proxy configurations to the data plane, which will cause the data plane to use a lot of system memory and network resources.

How to optimize the service mesh in response to these two problems?

1. Use proxyless mode: remove the sidecar proxy and return to the SDK.
2. Optimize the data plane: reduce the frequency and size of proxy configurations delivered to the data plane.
3. eBPF: used to optimize network hijacking.

This article will later explain these details in the performance optimization section.

Sidecar operation and maintenance management

Istio is built on top of Kubernetes and leverages Kubernetes' container orchestration and lifecycle management to automatically inject sidecars into pods through admission controllers. This happens each time Kubernetes creates pods.

To solve the sidecar resource consumption problem, there were four service mesh deployment modes proposed, as shown in the following figure.

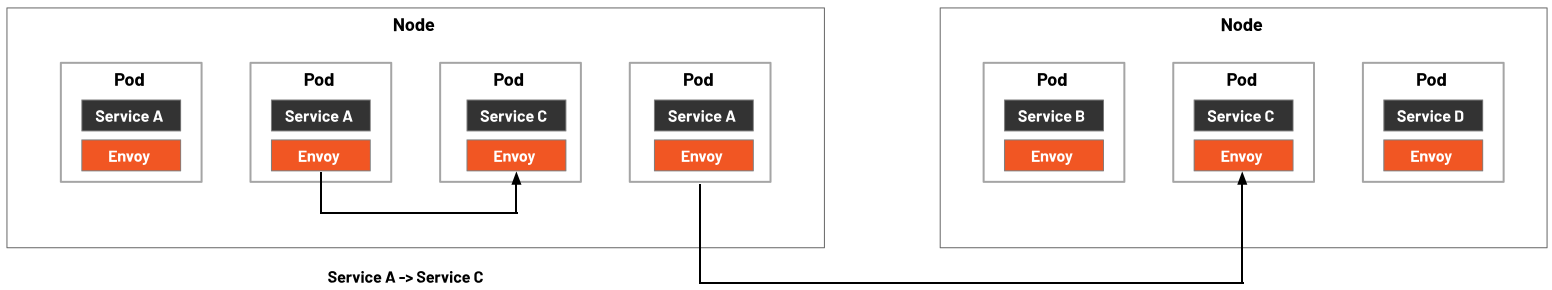

Figure 4. Mode 1: Sidecar per workload

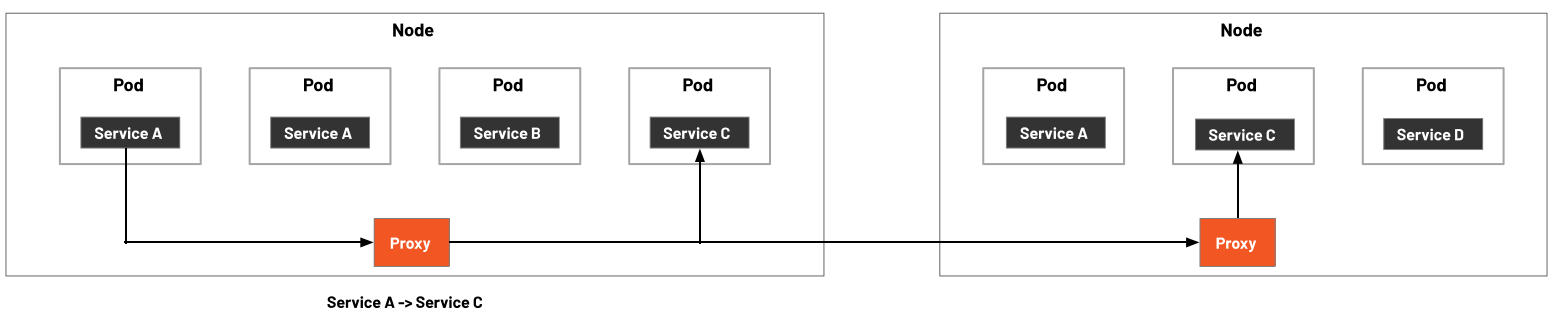

Figure 5. Mode 2: Shared proxy per node

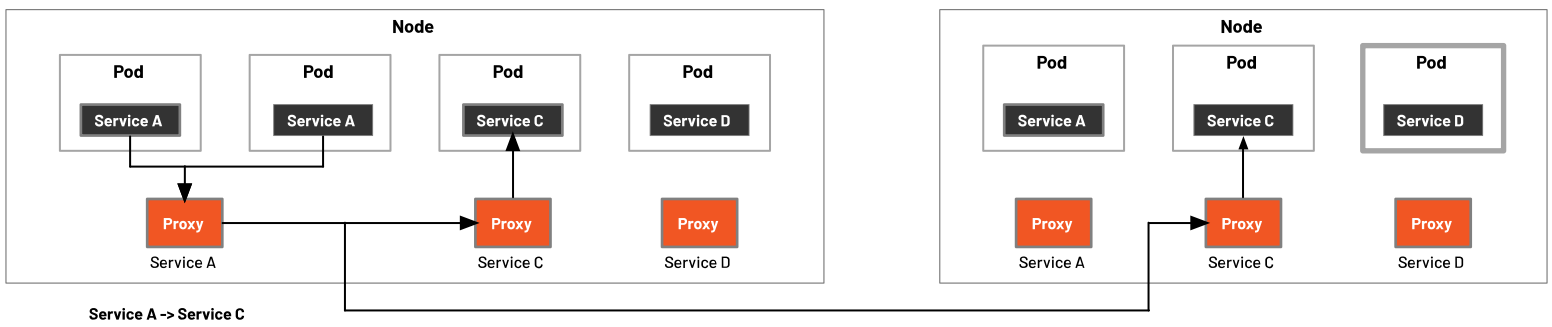

Figure 6. Mode 3: Shared proxy per service account per node

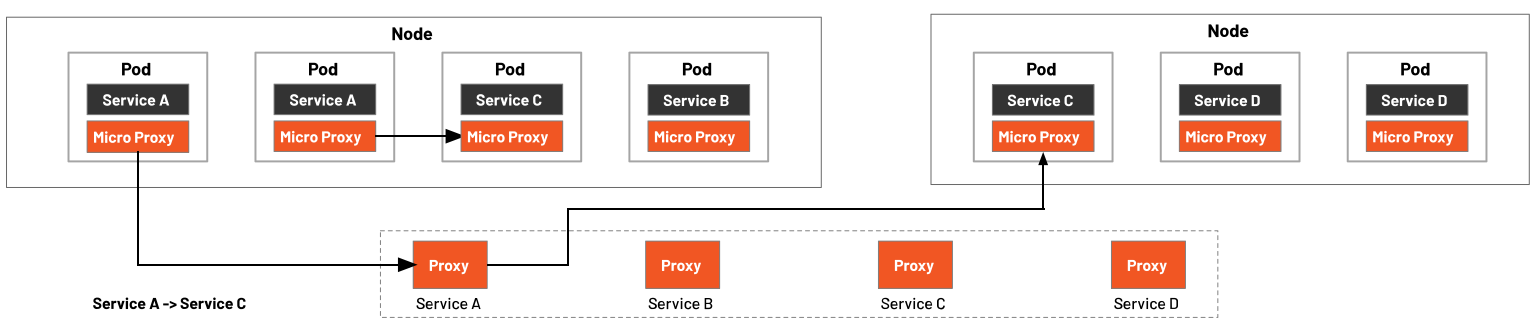

Figure 7. Mode 4: Micro-proxy with remote L7 proxy

The following table compares these four deployment methods in detail. Each of them has advantages and disadvantages. The specific choice depends on the current situation.

Mode	Memory overhead	Security	Fault domain	Operation and maintenance
Sidecar proxy	The overhead is most significant because a proxy is injected per pod.	Since the sidecar must be deployed with the workload, the workload can bypass the sidecar.	Pod-level isolation, if the proxy fails, only the workload in the Pod is affected.	A workload's sidecar can be upgraded independently without affecting other workloads.
Node sharing proxy	There is only one proxy on each node, shared by all workloads on that node, with low overhead.	There are security risks in the management of encrypted content and private keys.	Node-level isolation, if a version conflict, configuration conflict, or extension incompatibility occurs when a shared proxy is upgraded, it may affect all workloads on that node.	There is no need to worry about injecting sidecars.
Service Account / Node Sharing Proxy	All workloads under the service account/ identity use a shared proxy with little overhead.	It doesn’t guarantee authentication and security of connections between workloads and proxies.	The level of isolation between nodes and service accounts, the fault domain is the same as "node sharing proxy".	Same as in "node sharing proxy" mode.
Shared remote proxy with micro-proxy	Because we inject a micro-proxy for each pod, the overhead is relatively large.	L4 routing is decoupled from security concerns as micro-proxy only handles mTLS.	When a Layer 7 policy needs to be applied, the traffic of the workload instance is redirected to the L7 proxy and can be bypassed directly if it is not required. The L7 proxy can run as a shared node proxy, a per-service account proxy, or a remote proxy.	Same as "sidecar proxy" mode.

Programmable proxy

Zhang Xiaohui of Flomesh explained the evolution of proxy software. I will quote some of his views below to illustrate the crucial role of programmable proxies in Istio.

The following figure shows the evolution process of the proxy software from configuration to programmable mode, and the representative proxy software in each stage.

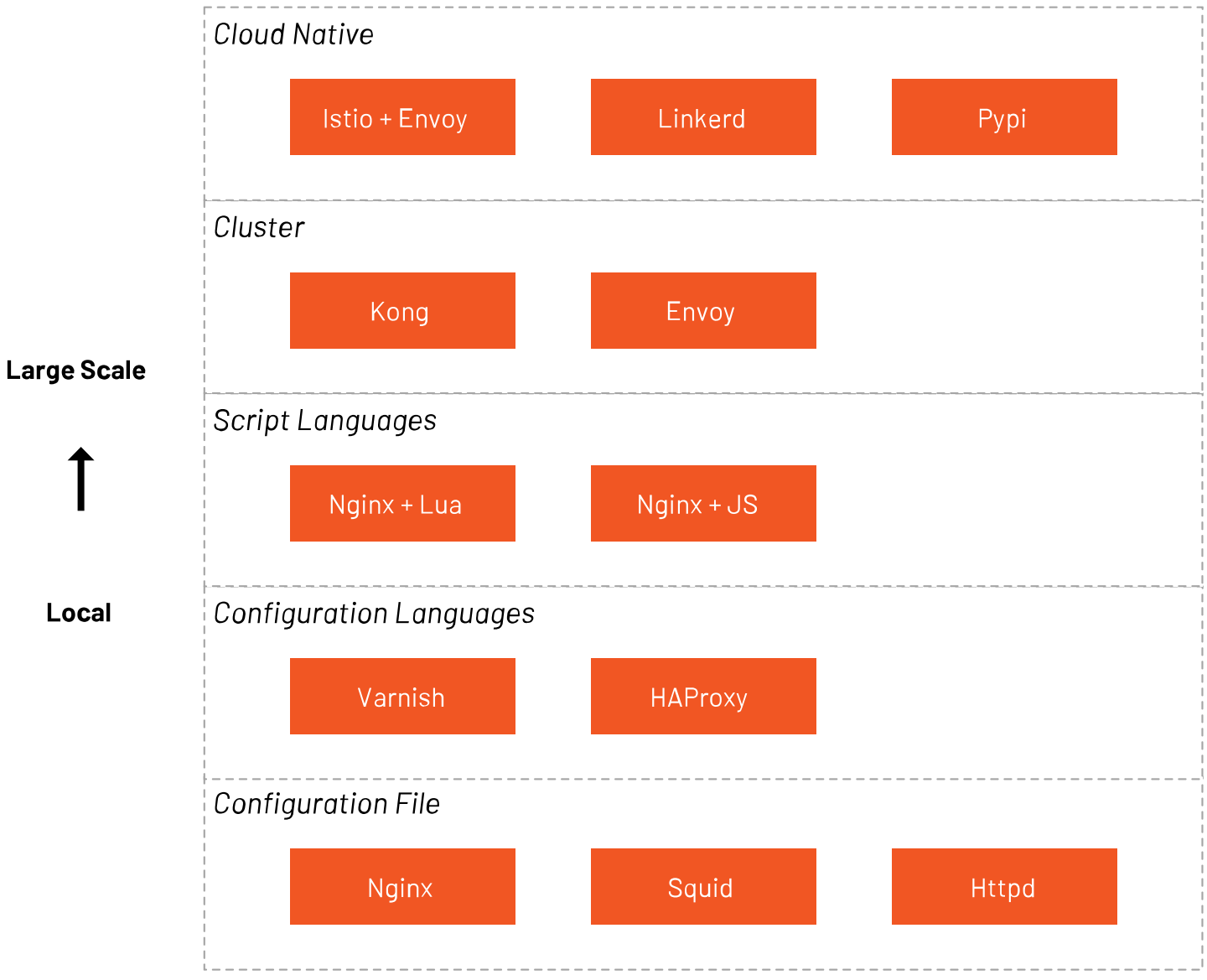

Figure 8. Evolution of proxy

The entire proxy evolution process follows the application as it moves from local and monolithic to large-scale and distributed. I will briefly outline the evolution of proxy software:

- Configuration files era: almost all software has configuration files, and proxy software is inseparable from configuration files because of its relatively complex functions. The proxy at this stage is mainly developed using the C language, including its extension modules, which highlights the proxy's ability. This is the original form of proxies, including Nginx, Apache HTTP Server, Squid, etc.

- Configuration language era: Proxies in this era are more extensible and flexible, and support features such as dynamic data acquisition and matching logic judgment. Varnish and HAProxy are two representative examples.
- Scripting language era: Since the introduction of scripting languages, proxies have become programmable. We can use scripts to add dynamic logic to proxies more easily, increasing development efficiency. The representative software is Nginx and its supported scripting languages.
- Clusters era: With the popularity of cloud computing, large-scale deployment and dynamic configuration of APIs have become necessary capabilities for proxies. With the increase in network traffic, large-scale proxy clusters have emerged. The representative proxies of this era include Envoy, Kong, etc.
- Cloud-native era: Multi-tenancy, elasticity, heterogeneous hybrid cloud, multi-cluster, security, and observability are all higher requirements for proxies in the cloud-native era. This will also be a historical opportunity for service meshes; the proxies will be combined together to form a mesh with representative software such as Istio, Linkerd, and Pypi.

Are these all service meshes?

The table below compares the current popular open source service mesh projects.

	Istio	**Linkerd**	**Consul Connect**	**Traefik Mesh**	**Kuma**	**Open Service Mesh (OSM)**
Current version	1.14	2.11	1.12	1.4	1.5	1.0l
License	Apache License 2.0	Apache License 2.0	Mozilla License	Apache License 2.0	Apache License 2.0	Apache License 2.0
Initiator	Google, IBM, Lyft	Buoyant	HashiCorp	Traefik Labs	Kong	Microsoft
Service proxy	Envoy, which supports proxyless mode for gRPC	Linkerd2-proxy	Default is Envoy , replaceable	Traefik Proxy	Envoy	Envoy
Ingress controller	Envoy, custom Ingress, supports Kubernetes Gateway API	CNCF	View Contribution Guidelines	View Contribution Guidelines	CNCF	CNCF

Governance	It is one of the most popular service mesh projects at present.	The earliest service mesh, the creator of the concept of "Service Mesh", the first service mesh project to enter CNCF, using a lightweight proxy developed with Rust.	Consul service mesh, using Envoy as a sidecar proxy.	A service mesh project launched by Traefik, using Traefik Proxy as a sidecar and supporting SMI (mentioned below).	A service mesh project launched by Kong that uses Envoy as a sidecar proxy, using Kong's own gateway as ingress	An open source service mesh created by Microsoft, using Envoy as a sidecar, compatible with SMI (also proposed by Microsoft).

In addition to the items listed above, there are a few others that appear to be mesh but are not:

- Envoy: Envoy is a cloud-native proxy, frequently used as a sidecar in other Envoy-based service meshes and for building API Gateways.
- Service Mesh Performance (SMP): Metrics that capture details of infrastructure capacity, service mesh configuration, and workload metadata to standardize service mesh values and describe the performance of any deployment.
- Service Mesh Interface (SMI): It is not a service mesh but a set of service mesh implementation standards. Similar to OAM, SPIFFE, CNI, CSI, etc., it defines interface standards, and the specific implementation varies. Currently, Traefik Mesh and Open Service Mesh claim to support this specification.
- Network Service Mesh: It's worth mentioning this project because it's often mistaken for a service mesh. In fact, it is oriented towards a three-layer network, and it can be used to connect multi-cloud/hybrid clouds without changing the CNI plug-in. It's not a "service mesh" as we define it, but a powerful complement to a service mesh (albeit a somewhat confusing name with a service mesh in it).

Looking at the so-called "service mesh" projects mentioned above, we can see that most service mesh projects started as proxies first, and then the control plane was implemented later. Istio, Consul Connect, Open Service Mesh, and Kuma use Envoy as a sidecar proxy. Only Linkerd and Traefik Mesh have created their proxies. All service mesh projects support the sidecar pattern. Apart from Istio, Linkerd, and Consul Connect, which have been used in production, other service mesh projects didn't have significant production usage.

Performance optimization for Istio

After the architecture stabilized in Istio 1.5, the community's main focus was on optimizing performance. In the following sections, we'll look at the different optimization methods that were considered by Istio.

Proxyless mode

Proxyless mode is an experimental feature proposed in Istio 1.11. It envisions a service mesh without a sidecar proxy based on gRPC and Istio. Using this pattern, we can add gRPC services directly to Istio without injecting an Envoy proxy into the pod. The figure below shows a comparison of sidecar mode and proxyless mode.

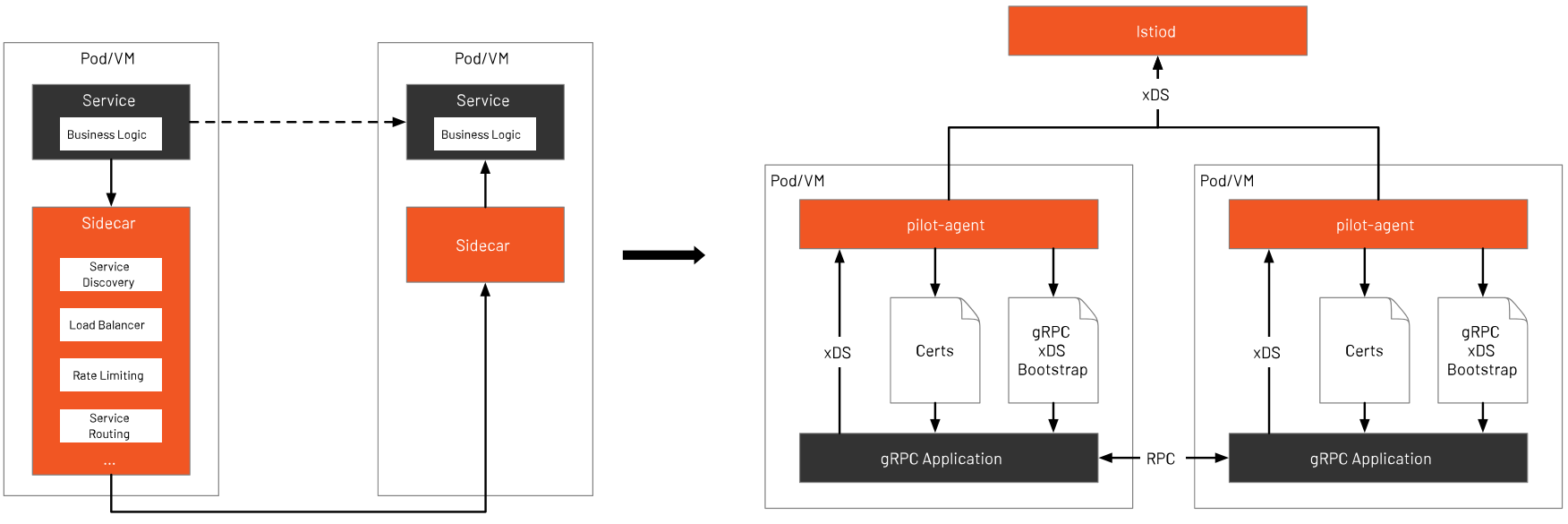

Figure 9. Sidecar vs. Proxyless

As we see from the above figure, although proxyless mode does not use a proxy for data plane communication, it still needs an agent for initialization and communication with the control plane. First, the pilot agent generates a bootstrap file at startup, in the same way that it generates bootstrap files in the sidecar mode. This tells the gRPC library how to connect to Istiod, where to find certificates for data plane communication, and what metadata to send to the control plane. Next, the pilot agent acts as an xDS proxy, connecting and authenticating with Istiod. Finally, the pilot agent obtains and rotates the certificate used in the data plane communication. This behavior pattern is the same as the sidecar mode.

The essence of a service mesh is not a sidecar model, nor a configuration center, or transparent traffic interception, but a standardized inter-service communication standard.

Some say that the proxyless model has returned to the old way of developing microservices based on an SDK, and the advantages of service meshes have been lost. Can it still be called service mesh? This is also a compromise on performance—if you mainly use gRPC to develop microservices, you only need to maintain gRPC versions in different languages; that is, you can manage microservices through the control plane.

Envoy xDS has become the de facto standard for communication between services in the mesh.

Optimizing traffic hijacking with eBPF

Earlier, we referred to a diagram that shows the different iptables rules such as PREROUTING, ISTIO_INBOUND, ISTIO_IN_REDIRECT, OUTPUT, ISTIO_OUTPUT, and so on. The traffic is routed based on these routes before it reaches the application.

Suppose now that service A wants to call service B running in another pod on a different host. The figure below shows the request path through the network stack.

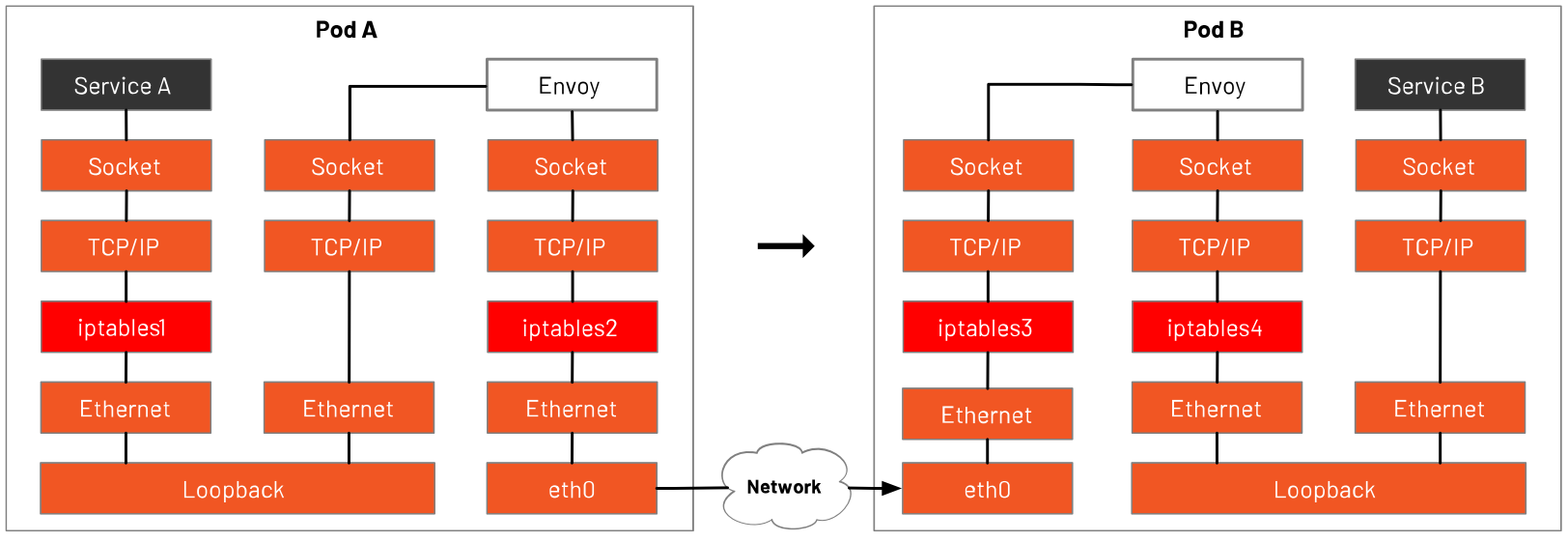

Figure 10. Service request path between pods on different hosts (iptables mode)

From the figure, we can see that there are four iptables passes in the whole calling process. Among them, the outbound (`iptables2`) from Envoy in Pod A and the inbound (`iptables3`) from `eth0` in Pod B are unavoidable. So can the remaining two, `iptables1` and `iptables4` be optimized?

Would it be possible to shorten the network path by letting the two sockets communicate directly? This requires programming through eBPF such that:

- Service A's traffic is sent directly to Envoy's inbound socket.
- After Envoy in Pod B receives the inbound traffic, it has determined that the traffic is to be sent to the local service and directly connects the outbound socket to Service B.

The transparent traffic interception network path using eBPF mode is shown in the following figure.

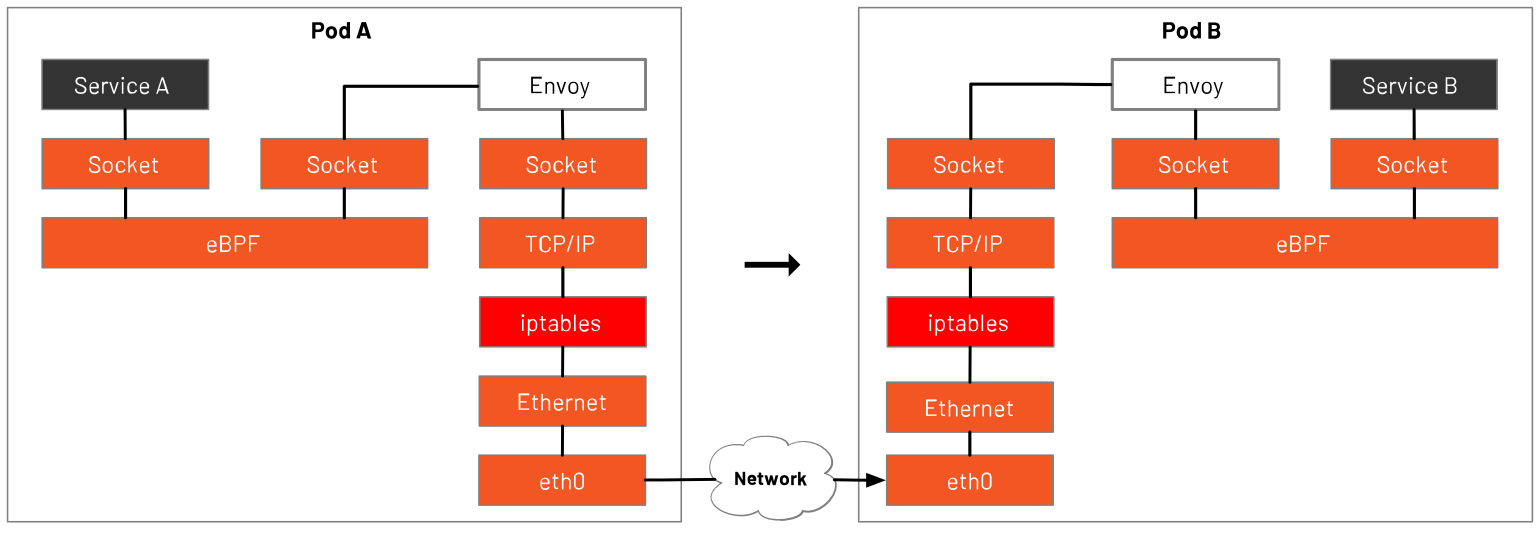

Figure 11. Service request path between pods on different hosts (eBPF mode)

The network path is shorter if services A and B are on the same node.

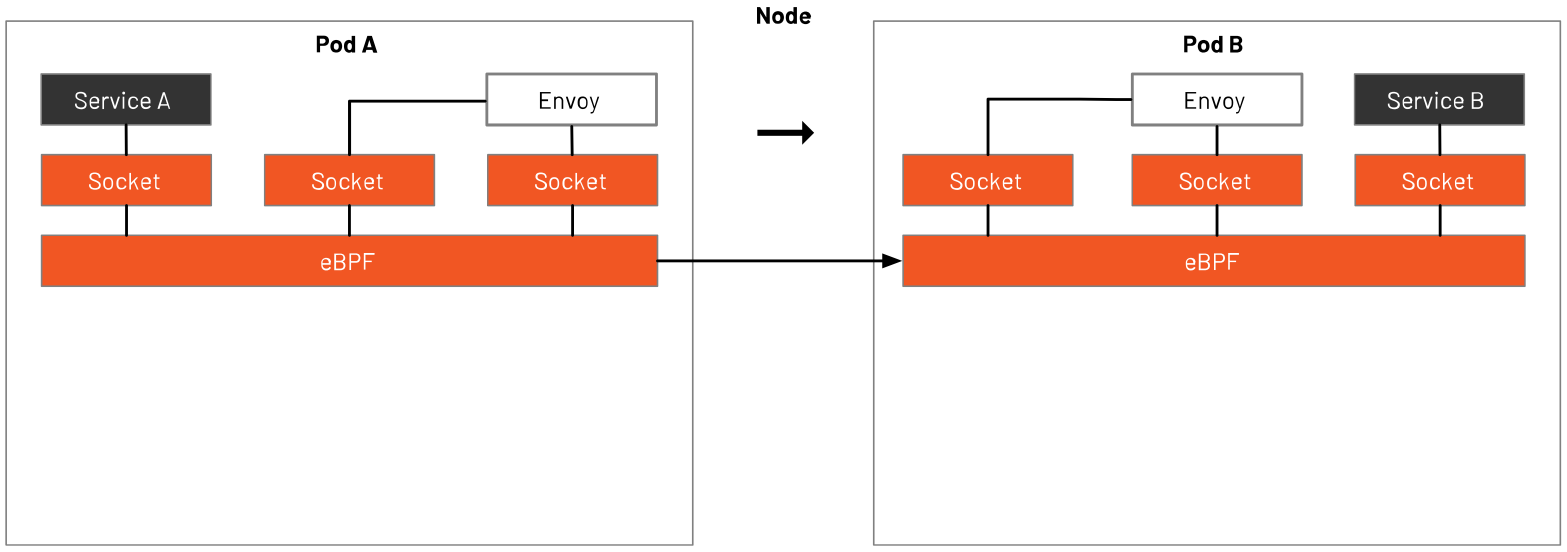

Figure 12. Network request path between pods on the same host (eBPF mode)

Access between services on the same node completely bypasses the TCP/IP stack and becomes direct access between sockets.

What is eBPF?

Modifying the Linux kernel code is difficult, and it takes a long time for new features to be released into the kernel. eBPF is a framework that allows users to load and run custom programs within the operating system's kernel. That is, with eBPF, you can extend and change the behavior of the kernel without directly modifying the kernel.

After the eBPF program is loaded into the kernel, it must pass the verifier verification before it can run. The verifier can prevent the eBPF program from accessing beyond its authority, ensuring the kernel's security.

eBPF programs are attached to kernel events and are triggered on entry or exit from a kernel function. In kernel space, eBPF programs must be written in a language that supports a compiler that generates eBPF byte code. Currently, you can write eBPF programs in C and Rust. Note that the eBPF program has compatibility issues with certain Linux versions.

Since the eBPF program can directly monitor and operate the Linux kernel, it has a view of the lowest level of the system. It can play a role in traffic management, observability, and security.

The open source project Merbridge uses eBPF to shorten the path of transparent traffic hijacking and optimize the performance of the service mesh. For details on the Merbridge implementation, you can refer to this Istio blog post.

The eBPF functions used by Merbridge require a Linux kernel version of at least 5.7.

At first glance, eBPF seems to implement the functions of Istio at a lower level and has a greater tendency to replace sidecar. But eBPF also has many limitations that make it impossible to replace service meshes and sidecars in the foreseeable future. Removing the sidecar in favor of a proxy-per-host model would result in the following

1. The blast radius of a proxy failure is expanded to the entire node.
2. It complicates the security problem because too many certificates are stored on a single node. If the node is compromised, all certificates and keys are compromised as well.
3. On the host, traffic contention between pods.

Moreover, eBPF is mainly responsible for Layer 3/4 traffic and can run together with CNI, but it is not suitable to use eBPF for Layer 7 traffic.

It doesn't seem like eBPF technology will be able to replace service meshes and sidecars anytime soon.

Control plane performance optimization

Earlier, we looked at the data plane optimizations. In this section, we'll look at how to optimize performance on the control plane side. You can think of a service mesh as a show, where the control plane is the director, and the data plane is all the actors. The director is not involved in the show but directs the actors. If the show's plot is simple and the duration is very short, then each actor will be allocated very few scenes, and rehearsal will be very easy.

If it is a large-scale show, the number of actors is large, and the plot is very complicated. To rehearse the show well, one director may not be enough. They can't direct so many actors, so we need multiple assistant directors (expanding the number of control plane instances). We also need to prepare lines and scripts for the actors. If actors can perform a series of lines and scenes in one shot (reducing the interruption of the data plane and pushing updates in batches), does that make the rehearsal more efficient?

From the above analogy, we can tease out the different aspects of the control plane that can be optimized:

- Reduce the size of the configuration that needs to be pushed.
- Batch push proxy configuration.
- Scale out the control plane by adding more instances.

Reduce the configuration that needs to be pushed out

The most straightforward way to optimize control plane performance is to reduce the scope and size of the proxy configurations to be pushed to the data plane. Assuming there is a workload A, it is possible to significantly reduce both the size of the configuration and the scope of workloads to be pushed by only pushing the proxy configuration related to A (i.e., the services that A depends on) as opposed to the configuration of all services in the mesh. The Sidecar resource can help us control which configuration gets sent. The following is an example of a sidecar configuration:

istio-sidecar-resource.yaml Raw

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: default
  namespace: us-west-1
spec:
  workloadSelector:
    labels:
      app: app-a
  egress:
  - hosts:
    - "us-west-1/*"
```

We can use the workloadSelector field to limit the scope of workloads that the sidecar configuration applies to. The egress field is used to determine the scope of services the workload should be aware of. The control plane will configure the selected workloads to only receive configuration on how to reach services in the cn-bj namespace, instead of pushing configuration for all services in the mesh. The configuration size being pushed by the control plane inside of the service mesh reduces its memory and network usage.

Batch push the proxy configurations

The process of pushing the proxy configuration from the control plane to the data plane is complex. The following figure shows the process.

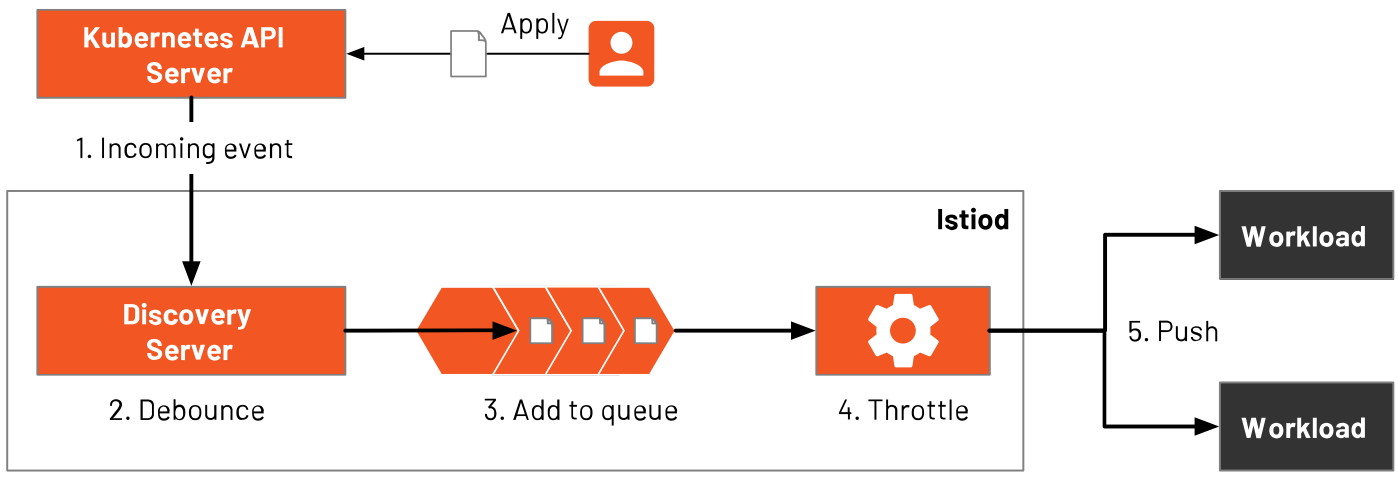

Figure 13.Istiod pushing proxy configuration to the data plane

After an administrator configures the Istio mesh, the process of pushing proxy configuration is as follows:

1. The event that the administrator updates the configuration will trigger the configuration synchronization of the data plane.
2. Istio's DiscoveryServer components listen to these events and add them to the queue where the events get merged for a certain period of time. This process is called debouncing, and it prevents too frequent updates to the data plane configuration.
3. After the debouncing period, the events are pushed to the queue.
4. To expedite the push progress, Istiod limits the number of simultaneous push requests.
5. Events are translated into Envoy configuration that gets pushed to the workloads.

From the above process, we can see that the key to optimizing configuration push is the debounce period in step 2 and the limit on the simultaneous pushes in step 4. There are several environmental variables that can help us tweak the control plane pushes:

- PILOT_DEBOUNCE_AFTER: The time after which the event will be added to the push queue.
- PILOT_DEBOUNCE_MAX: This defines the maximum amount of time an event can debounce.
- PILOT_ENABLE_EDS_DEBOUNCE: Specifies whether endpoint updates meet debounce rules or have priority and fall into the push queue immediately.
- PILOT_PUSH_THROTTLE: Controls how many push requests are processed at once.

Please refer to the Istio documentation for the default values and specific configuration of these environment variables.

When setting these values, you can follow these principles:

- If control plane resources are idle, to speed up the propagation of configuration updates, you can:
  - Shorten the debouncing period and increase the number of pushes.
  - Increase the number of push requests processed simultaneously.
- If the control plane is saturated, to reduce performance bottlenecks, you can:
  - Lengthen the debouncing cycle to reduce the number of pushes.
  - Increase the number of push requests processed simultaneously.

The optimal solution will depend on your scenarios. Make sure you refer to observability tools when making the optimizations.

Scaling the control plane

If configuring the debounce batch processing and using the Sidecar resource doesn't optimize the performance of the control plane, the last option is to scale out the control plane. This includes increasing the CPU and memory resources of a single Istiod instance as well as increasing the number of instances. Whether to scale up or out depends on the situation:

- When the resource usage of an Istiod is saturated, it is recommended that you increase the CPU/memory of the Istiod. This is usually because there are too many resources in the service mesh (Istio's custom resources, such as VirtualService, DestinationRule, etc.) that need to be processed.
- Then increase the number of instances of Istiod so that the number of workloads to be managed by a single instance can be spread out.

Data plane performance optimization

Apache SkyWalking can serve as an observability tool for Istio and can also help us analyze the performance of services in dynamic debugging and troubleshooting. The newly released Apache SkyWalking Rover component uses eBPF technology to identify Istio's key performance issues accurately.

We can increase Envoy's throughput and optimize Istio's performance with the following approaches:

- Disabling Zipkin tracing or reducing the sampling rate.
- Simplify the access log format.
- Disable Envoy's Access Log Service (ALS).

For data that shows the impact of the above optimizations on Envoy's throughput, see Pinpoint Service Mesh Critical Performance Impact by using eBPF.

Envoy: the service mesh's leading actor

We know that the service mesh is composed of the data plane and the control plane. From the list of service mesh open source projects we mentioned earlier, we can see that most of the projects are based on Envoy while running their own control plane. If we continue with the previous analogy of Istio being a show, we can say that Envoy is the main actor and has a leading role.

The xDS protocol, invented by Envoy, has become a generic API for service meshes.

The diagram shows the architecture of Envoy.

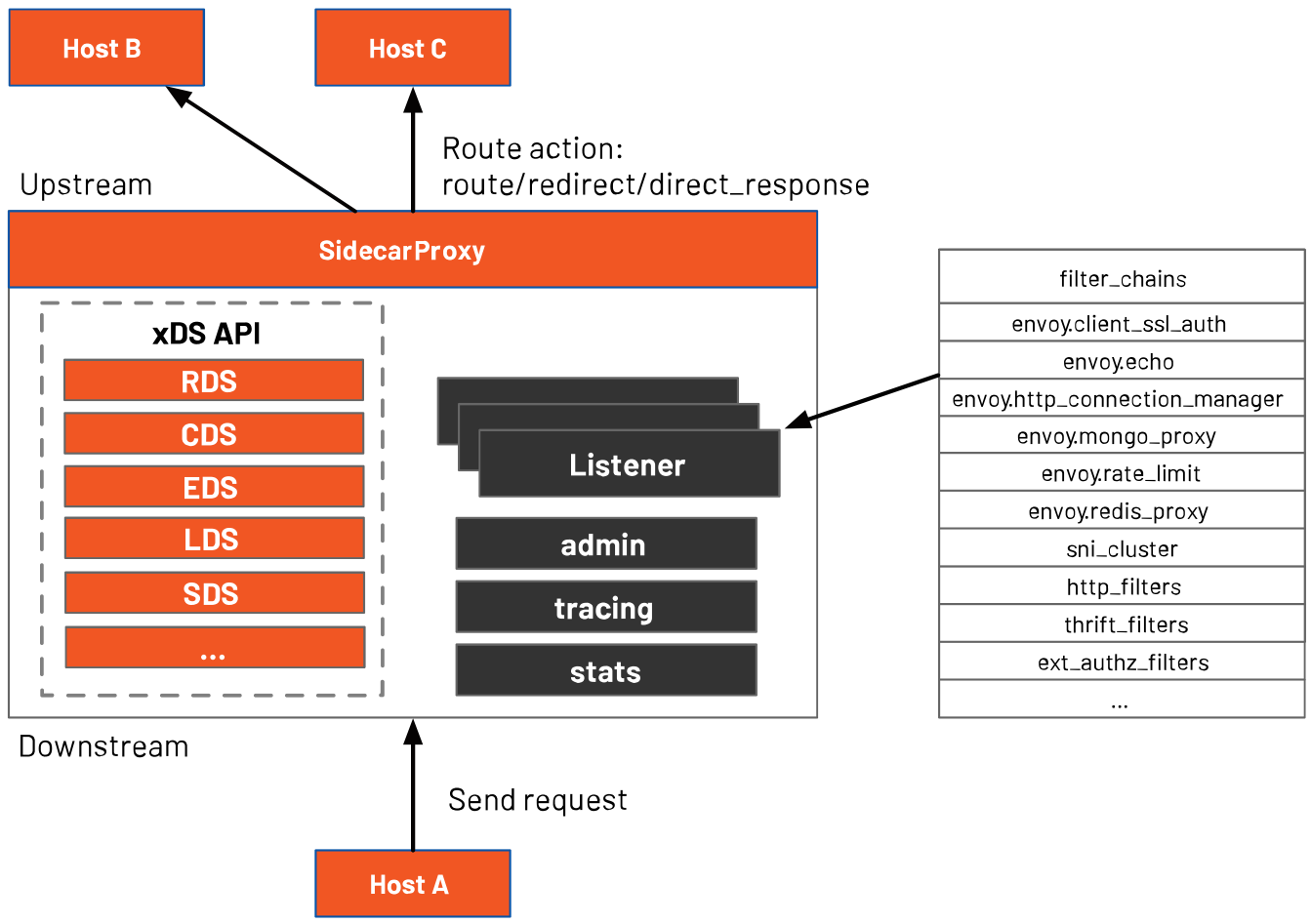

Figure 14. Envoy Architecture Diagram

The xDS API is what sets Envoy apart from other proxies. Its code and parsing mechanism are intricate and difficult to expand. The following is a detailed overview of the different components in Istio mesh. From the figure, we can see that pilot-agent is the process that launches and manages the lifecycle of the Envoy proxy.

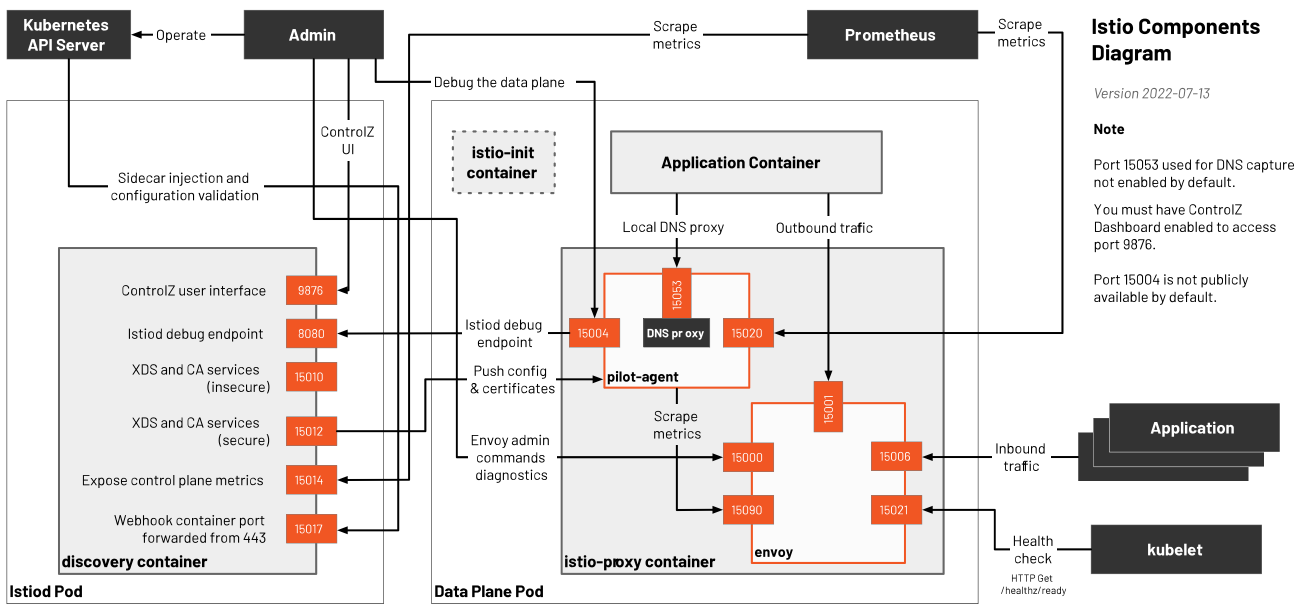

Figure 15. Istio components

The role of the pilot-agent process is as follows:

- It's the parent process in the container and it's responsible for the lifecycle management of Envoy.
- It receives pushes from the control plane and configures the proxy and certificates.
- It collects Envoy statistics and aggregates sidecar statistics for Prometheus to collect.
- Includes a built-in local DNS proxy for resolving internal domain names of the cluster that cannot be resolved by Kubernetes DNS.
- It performs health checks for Envoy and the DNS proxy.

Based on the above roles of the pilot-agent, we can see that it is mainly used for interacting with Istiod and being an intermediary between the control plane and the Envoy proxy. So will Envoy "act and guide," no longer cooperate with Istio, and build its own control plane?

In a Sidecar container, the pilot-agent is like Envoy's "Sidecar".

Envoy Gateway unified service mesh gateway

In addition to the Kubernetes services resource, the ingress resource is the one that manages external access to services running inside the cluster. Using Ingress, we can expose services in the cluster and route traffic to services via HTTP Hosts and URI paths. Compared to exposing services directly using the service resource, using the ingress resource can reduce the network access point (PEP) of the cluster and reduce the risk of the cluster being attacked by the network and you only need one load balancer. The following figure shows the process of using Ingress to access services in the cluster.

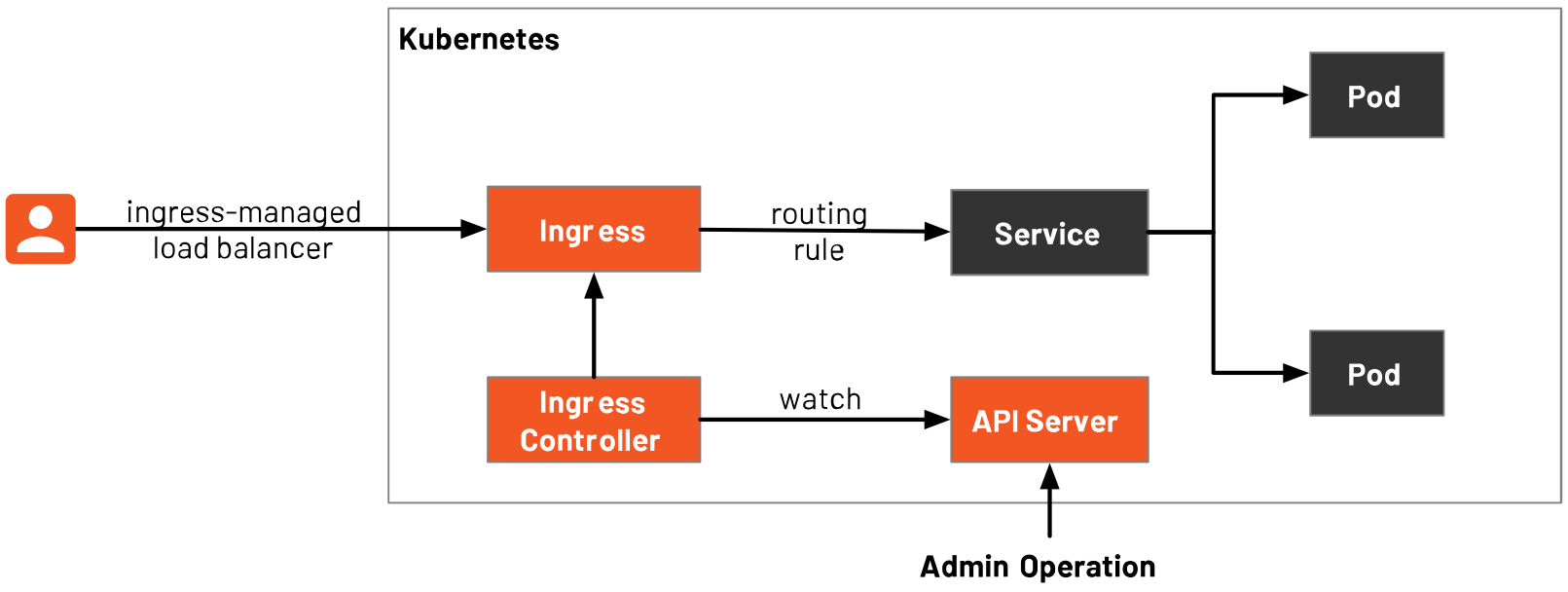

Figure 16. Kubernetes Ingress traffic access flow chart

Before Kubernetes, API Gateway was widely used as edge routing. When referring to Istio, Istio's custom Gateway resources are added, which makes accessing resources in the Istio service mesh one more option, as shown in the following figure.

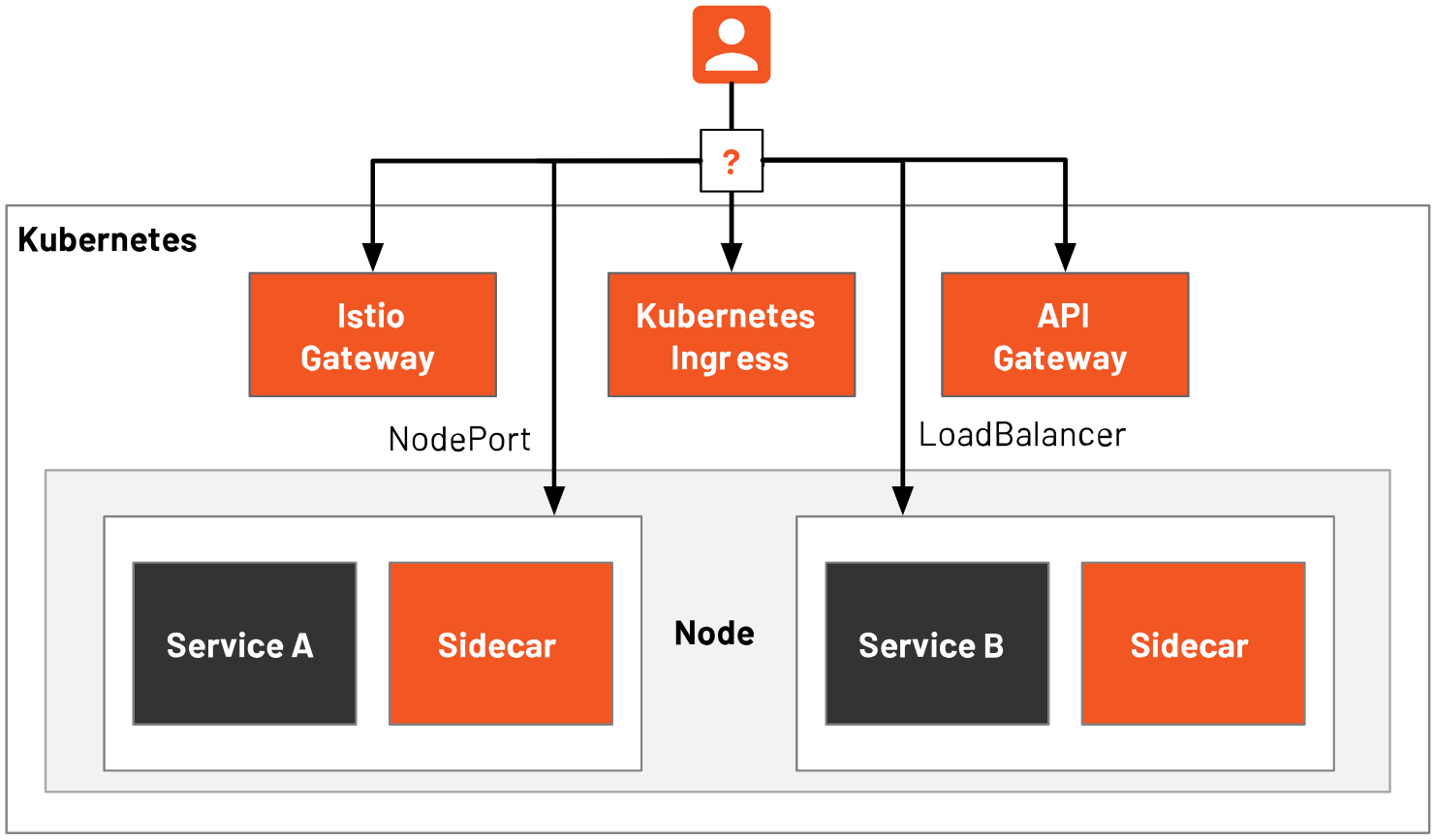

Figure 17. Ways to access services in the Istio meshz

Which option do we choose when exposing services in a single Istio service mesh? Do we pick NodePort, LoadBalancer, Istio Gateway, Kubernetes Ingress, or an API Gateway? How do clients access services within the mesh if it is a multi-cluster service mesh? In addition to working as a sidecar proxy in Istio, Kuma, and Consul Connect, Envoy Proxy can also be used standalone as an ingress gateway: Contour, Emissary, Hango and Gloo.

Because the Envoy community does not offer a control plane implementation, these projects use Envoy to implement service meshes and API gateways, which results in a great deal of functional overlap, proprietary features, or a lack of community diversity.

In order to change the status quo, the Envoy community started the Envoy Gateway project. The project aims to combine the experience of existing Envoy-based API Gateway related projects. Some Envoy-specific extensions to the Kubernetes Gateway API lower the barrier to entry for Envoy users to use gateways. Because the Envoy Gateway still issues configuration to the Envoy proxy through xDS, you can also use it to manage gateways that support xDS, such as the Istio Gateway.

The gateways we have seen now are basically used as ingress gateways in a single cluster and can do nothing in multi-cluster and multi-mesh deployments. To deal with multiple clusters, we need to add another layer of gateways on top of Istio and a global control plane to route traffic between multiple clusters, as shown in the figure below.

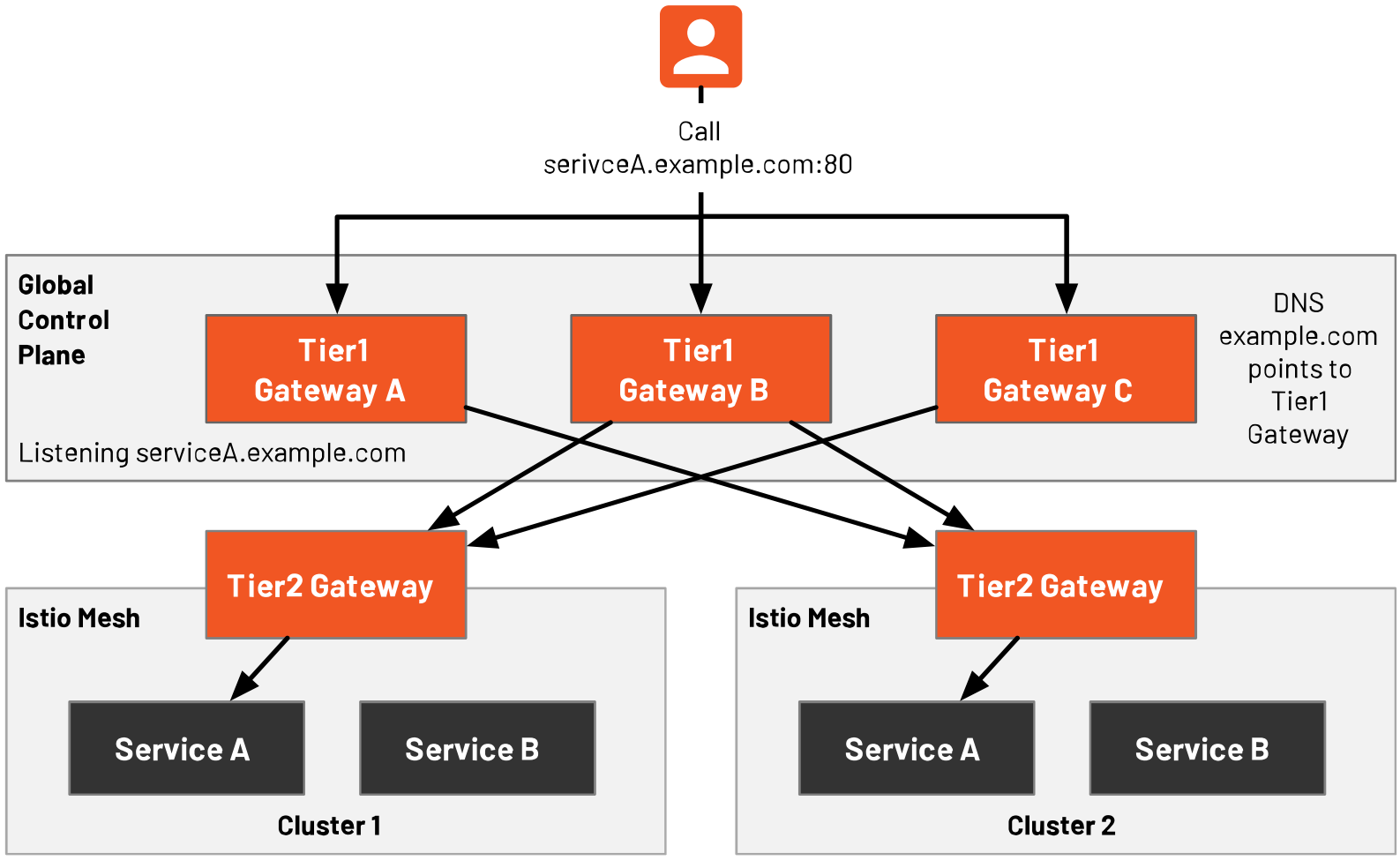

Figure 18. Two-tier gateway with multi-cluster and multi-mesh

A brief introduction to two-tier gateways

The Tier-1 gateway (from now on referred to as T1) is located at the application edge and it's used in a multi-cluster environment. The same application is hosted on different clusters at the same time, and the T1 gateway routes the application's request traffic between these clusters.

The Tier-2 gateway (from now on referred to as T2) is located at the edge of a cluster and is used to route traffic to services within that cluster managed by the service mesh.

To manage multi-cluster service meshes, we add a layer on top called a global control plane. The global control plane and APIs work together and in addition to the Istio control planes in individual clusters. A single point of failure is prevented by clustering the T1 gateways. To learn more about two-tier gateways, refer to designing traffic flow via Tier-1 and Tier-2 ingress gateways.

This is an example of how a T1 gateway cloud can be configured:

```yaml
apiVersion: gateway.tsb.tetrate.io/v2
kind: Tier1Gateway
metadata:
  name: service1-tier1
  group: demo-gw-group
  organization: demo-org
  tenant: demo-tenant
  workspace: demo-ws
spec:
  workloadSelector:
    namespace: t1
    labels:
      app: gateway-t1
      istio: ingressgateway
  externalServers:
  - name: service1
    hostname: servicea.example.com
    port: 80
    tls: {}
    clusters:
    - name: cluster1
      weight: 75
    - name: cluster2
      weight: 25
```

This configuration exposes the servicea.example.com service through the T1 gateway and forwards 75% of the traffic to cluster1 and 25% of the traffic to cluster2.

In addition, in order to deal with the traffic, services, and security configurations in multiple clusters, Tetrate's flagship product, Tetrate Service Bridge, a series of group APIs have also been added to the TSB documentation for details.

Istio open source ecosystem

Istio has been open source for more than five years. Many open source projects have emerged in the past two years, among which the more representative ones built on top of Istio are:

- Slime (NetEase open source)
- Tencent's open source initiatives Aeraki
- Istio's official support for Wasm plugins

Their presence makes Istio more intelligent and expands the scope of Istio.

Slime

Slime is an Istio-based smart mesh manager open-sourced by the NetEase Shufan microservices team. Slime has been implemented based on Kubernetes Operator and can be used as the CRD manager of Istio. It can define dynamic service governance policies without any customization of Istio, so as to achieve the purpose of automatically and conveniently using the high-level functions of Istio and Envoy.

In the control plane performance optimization section, we mentioned optimizing the performance of Istio by "reducing the configuration that needs to be pushed". However, Istio cannot automatically identify and cannot rely on the proxy configuration that needs to be pushed to each sidecar to optimize it.

Slime provides a lazyload controller, which can help us implement lazy loading of the configuration. Users do not need to configure the SidecarScope manually. Istio can load service configuration and service discovery information on demand.

The following figure shows the flow chart of updating the data plane configuration with Slime as the management plane of Istio.

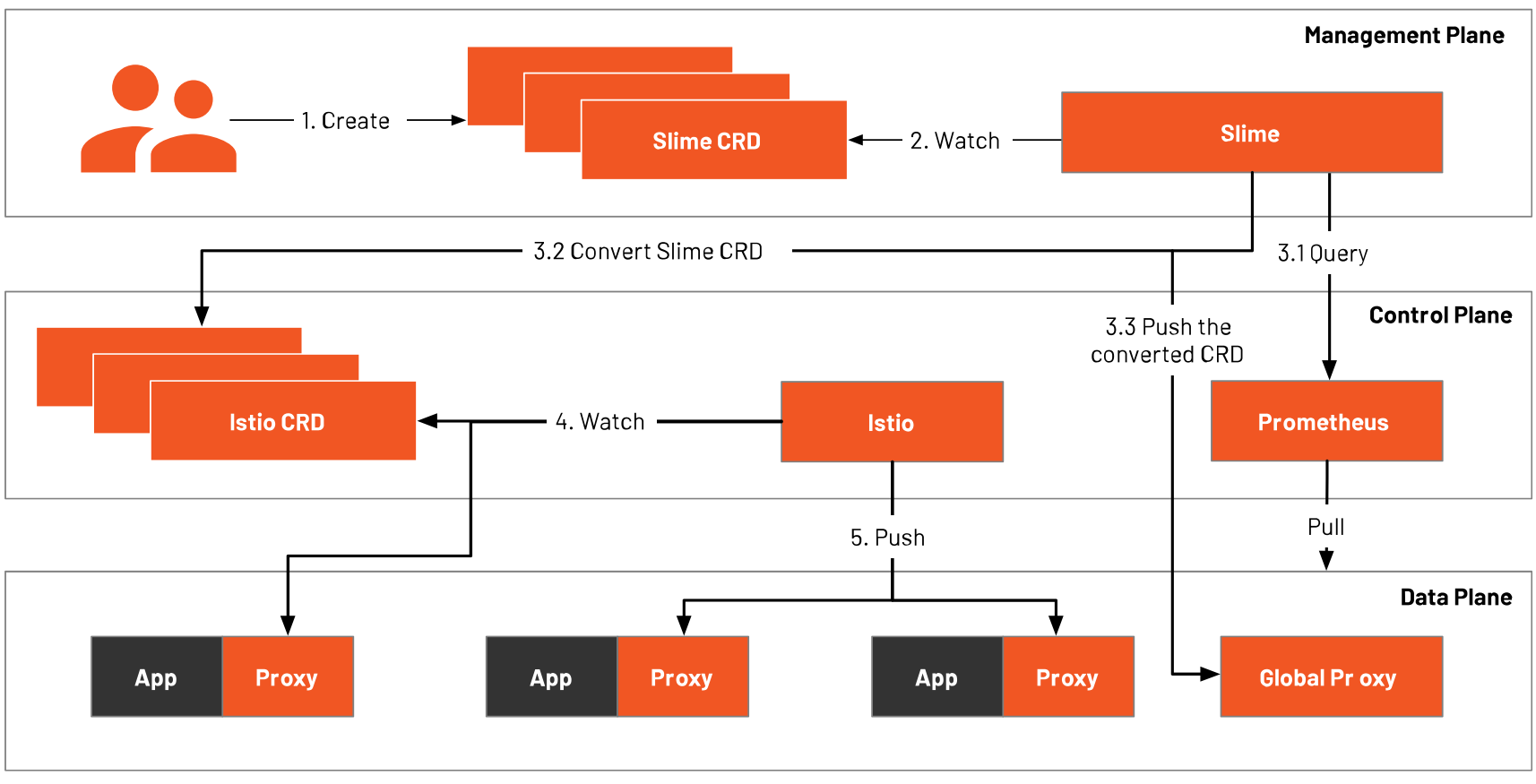

Figure 19. Updating Istio data plane configuration using Slime

The Global Proxy is built using Envoy, deploying one in each namespace that needs to initiate configuration lazy loading or just one in the entire mesh. All calls that lack service discovery information (you can also manually configure the service invocation relationship) are diverted by the sneaky route to the Global Proxy, where after its initial forwarding, Slime senses the invoked party's information and discovers the mapping between the service name and the requested resource.

The specific steps for updating the data plane configuration are as follows:

- The Slime Operator completes the initialization of Slime components in Kubernetes according to the administrator's configuration.
- Developers create Slime CRD resources and apply them to the Kubernetes cluster.
- Slime queries the monitoring data of related services saved in Prometheus, converts Slime CRD to Istio CRD in combination with the configuration of the adaptive part in Slime CRD, and pushes it to Global Proxy at the same time.
- Istio monitors the creation of Istio CRDs.
- Istio sends the Sidecar Proxy's configuration info to maintain the service invocation relationship and solve the problem of missing service information.

Aeraki

Aeraki Mesh is a service mesh project open sourced by Tencent Cloud in March 2021. It expands support for seven-layer protocols based on Istio and focuses on solving the service governance of non-HTTP protocols in Istio. It entered the CNCF sandbox in June 2022. The following figure shows the architectural diagram of Aeraki.

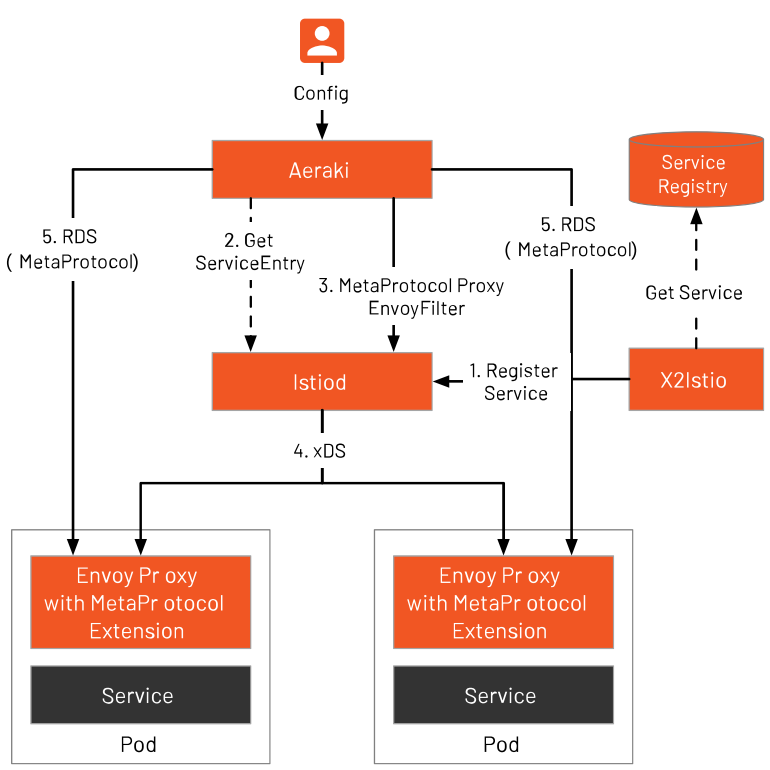

Figure 20. Aeraki architecture

The process of using Aeraki to serve non-HTTP in an Istio mesh is as follows:

- Aeraki's X2Istio component connects to the service registry, obtains the registration information of non-HTTP services, and generates a ServiceEntry to register with Istio.
- Aeraki, as the management plane on top of Istio, obtains the ServiceEntry configuration from Istio.

- Aeraki judges the protocol type of the service (e.g. tcp-metaprotocol-dubbo) through the port command, then generates the MetaProtocol Proxy Filter (compatible with EnvoyFilter) configuration and, at the same time modifies the RDS address to point it to Aeraki.
- Istio uses the xDS protocol to deliver the configuration (LDS, CDS, EDS, etc.) to the data plane.
- Aeraki generates routing rules based on the information in the service registry and user settings and sends them to the data plane through RDS.

The key to the whole process of accessing non-HTTP services in Istio is the MetaProtocol Proxy. Istio supports HTTP/HTTP2, TCP and gRPC protocols by default, and experimentally supports Mongo, MySQL and Redis protocols. To use Istio to route traffic for other protocols not only requires a lot of work to modify the Istio control plane and extend Envoy, but also a lot of duplication because different protocols share common control logic. The Envoy MetaProtocol Proxy is a general seven-layer protocol proxy implemented based on Envoy. The MetaProtocol Proxy is an extension based on the Envoy code. It implements basic capabilities such as service discovery, load balancing, RDS dynamic routing, traffic mirroring, fault injection, local/global traffic limiting, etc. for the seven-layer protocol, which greatly reduces the difficulty of third-party protocol development for Envoy.

The following figure shows the architecture diagram of MetaProtocol Proxy.

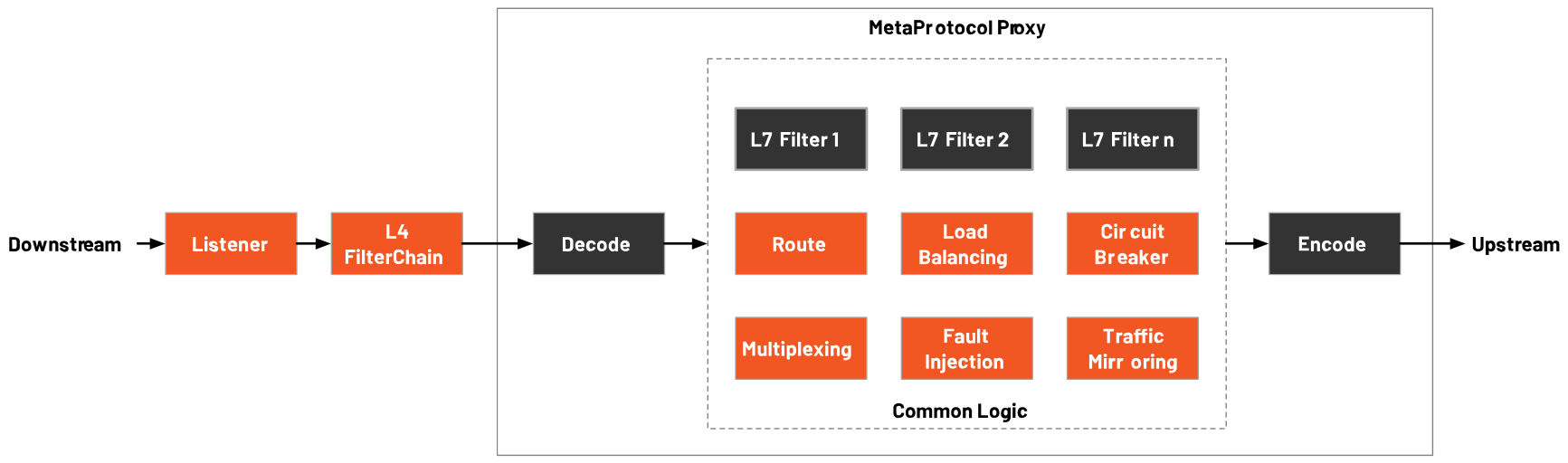

Figure 21. MetaProtocol Proxy architecture

When we want to extend Istio to support other seven-layer protocols such as Kafka, Dubbo, and Thrift, we only need to implement the codec interfaces (Decode and Encode) in the above figure, and then we can quickly develop a third-party protocol plug-in based on MetaProtocol. Because MetaProtocol Proxy is an extension of Envoy, you can still develop filters for it in different languages and use EnvoyFilter resources to deliver configuration to the data plane.

WasmPlugin API

WasmPlugin is an API introduced in Istio 1.12. As a proxy extension mechanism, we can use it to add custom and third-party Wasm modules to the data plane. The diagram below shows how a user can use the WasmPlugin in Istio.

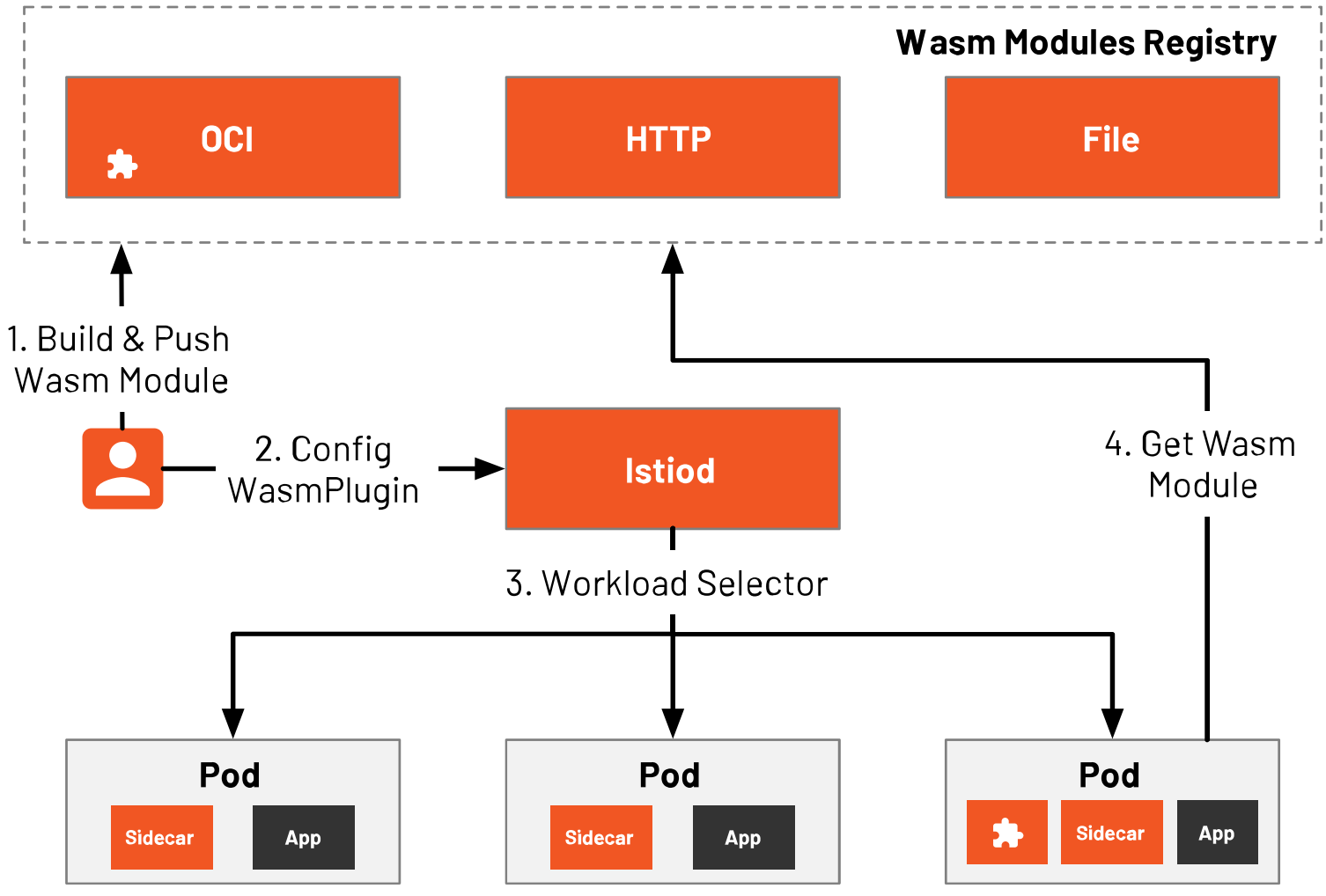

Figure 22. Using WasmPlugin in Istio mesh

Specific steps are as follows:

1. Users use the Proxy-Wasm SDK (currently available in AssemblyScript, C++, Rust, Zig, and Go) to develop extensions and build them into OCI images (such as Docker images) to upload to the mirror repository.
2. The user writes the WasmPlugin configuration and applies it to Istio.
3. The WasmPlugin configuration for the workload in the configuration is selected by the Istio control plane, and the Wasm module is injected into the specified Pod.
4. The pilot-agent in the sidecar loads the Wasm modules from remote or local files and run them in Envoy.

Who should use Istio?

Well, having said that, what does this have to do with you? Istio's relationship with you depends on your role.

- If you are the platform leader, after applying the service mesh, you may enhance the observability of your platform and have a unified platform to manage microservices. You will be the direct beneficiary and the main implementer of the service mesh.
- If you are an application developer, you will also benefit from a service mesh because you can be more dedicated to the business logic and not worry about other non-functional issues such as retry policies, TLS, etc..

The following diagram shows the adoption path for service meshes.

Whether to adopt a service mesh depends on your technology development stage, whether the application implements containerization and microservices, the need for multi-language, whether mTLS is required, and the acceptance of performance loss.

Service mesh positioning in the cloud native technology stack

The development of technology is changing with each passing day. In the past two years, some new technologies have appeared, which seem to challenge the status of the service mesh. Some people claim that it can directly replace the existing service mesh of the classic sidecar model. We should not be confused by the noise of the outside world, correcting the positioning of service mesh in the cloud native technology stack.

Blindly promoting a technology and ignoring its applicable scenarios is hooliganism.

The diagram below shows the cloud-native technology stack.

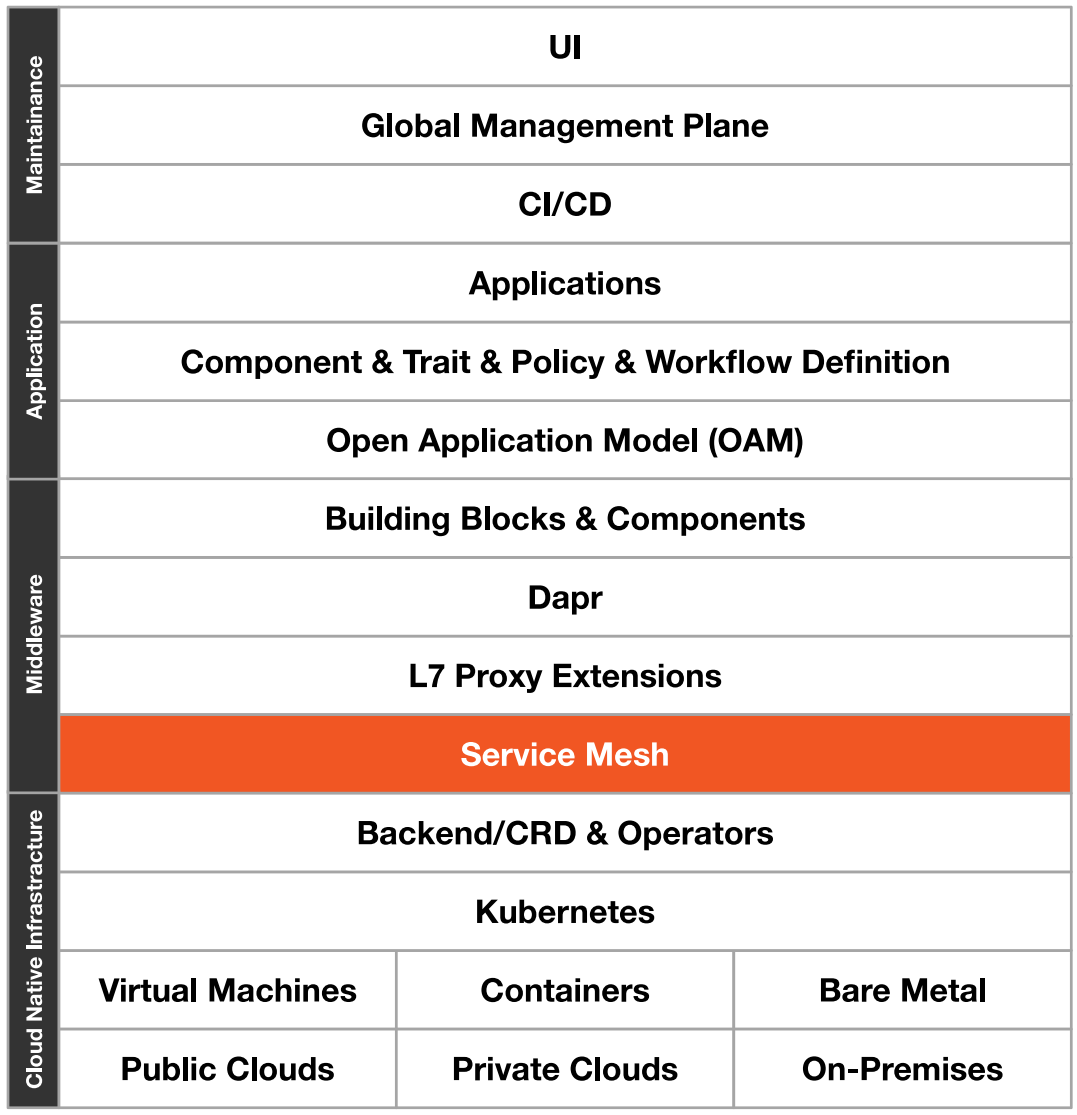

Figure 23. Cloud native technology stack

We can see that the "cloud infrastructure", "middleware", and "application" layers in the cloud native technology stack diagram all enumerate some iconic open source projects that build standards in their fields:

- In the field of cloud infrastructure, Kubernetes unifies the standards for container orchestration and application life cycle management, and the Operator mode lays the standards for extending the Kubernetes API and third-party application access.
- In the field of middleware, the service mesh assumes some or all of the responsibilities of the seven-layer network, observability, and security in the cloud native technology stack. It runs in the lower layer of the application and is almost imperceptible; Dapr (distributed application runtime) defines the capability model of cloud-native middleware. Developers can integrate Dapr's multi-language SDK into their applications and programs for the distributed capabilities provided by Dapr, without caring about the applications running on them. environment and docking back-end infrastructure. Because the Dapr runtime (Sidecar mode deployment, which contains various building blocks) is running in the same Pod as the application, it automatically connects us with the backend components;
- In the application field, OAM aims to establish an application model standard, an application through components, characteristics, policies, and workflows.

The diagram below shows how Istio is positioned for seven-tier mesh management in a cloud-native deployment.

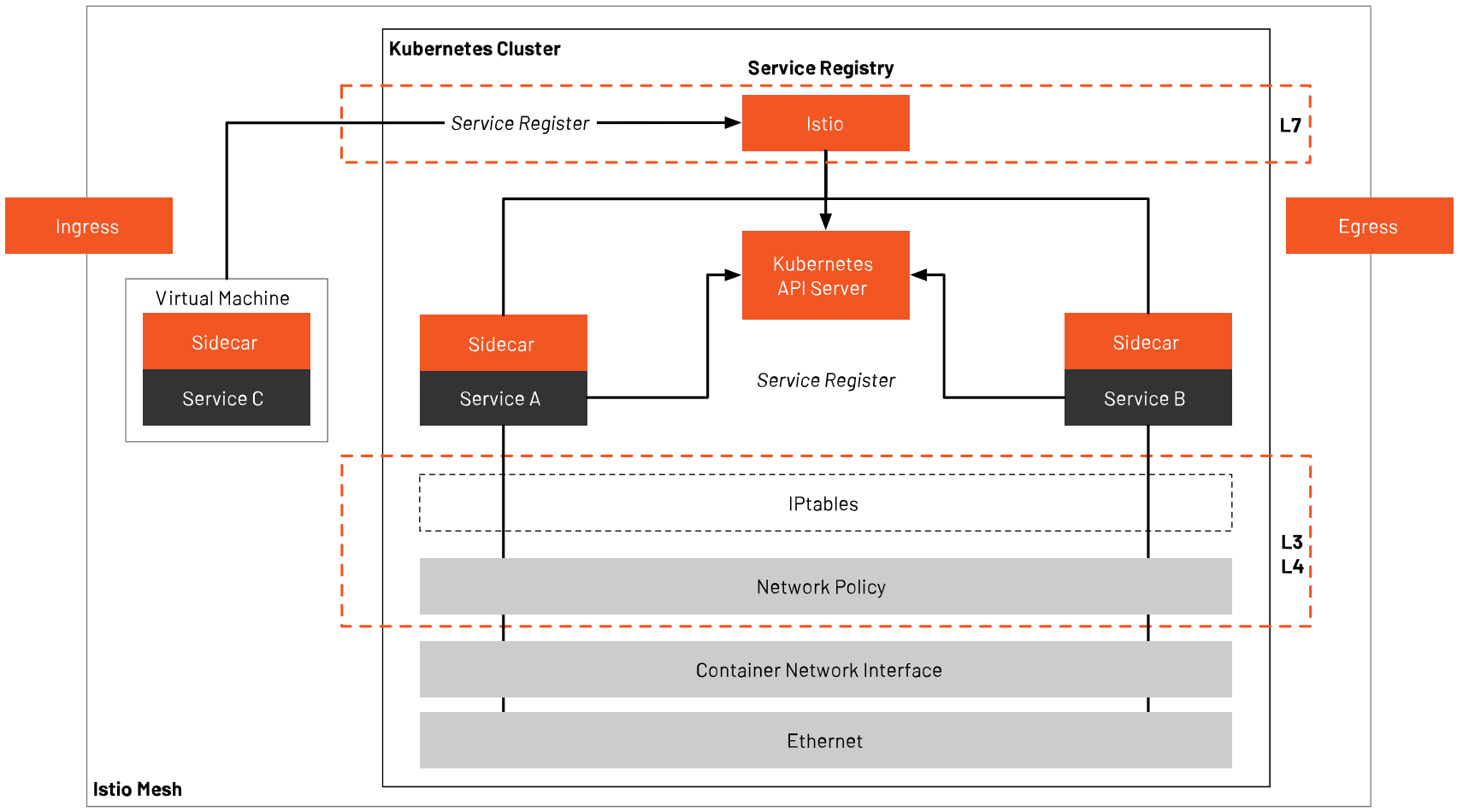

Figure 24. Istio is positioned in a layer-7 network in a cloud-native architecture

What is the relationship between Dapr and Istio?

Similarities between Istio and Dapr:

- Both Istio and Dapr can use the sidecar mode deployment model.
- Both belong to middleware and can also manage communication between services.

Differences between Istio and Dapr:

- Different goals: Istio's goal is to build a zero-trust network and define inter-service communication standards, while Dapr's goal is to build a standard API for middleware capabilities.
- Different architectures: Istio = Envoy + transparent traffic hijacking + control plane; Dapr = multilingual SDK + standardized API + distributed capability components.
- However, the application of Istio is almost imperceptible to developers and mainly requires the implementation of the infrastructure operation and maintenance team, while the application of Dapr requires developers to independently choose to integrate the Dapr SDK.

The future of service mesh

In the above article, I introduced the development context and open source ecosystem of Istio. Next, I will introduce the future trends of Istio service mesh:

- Zero trust network
- Hybrid cloud

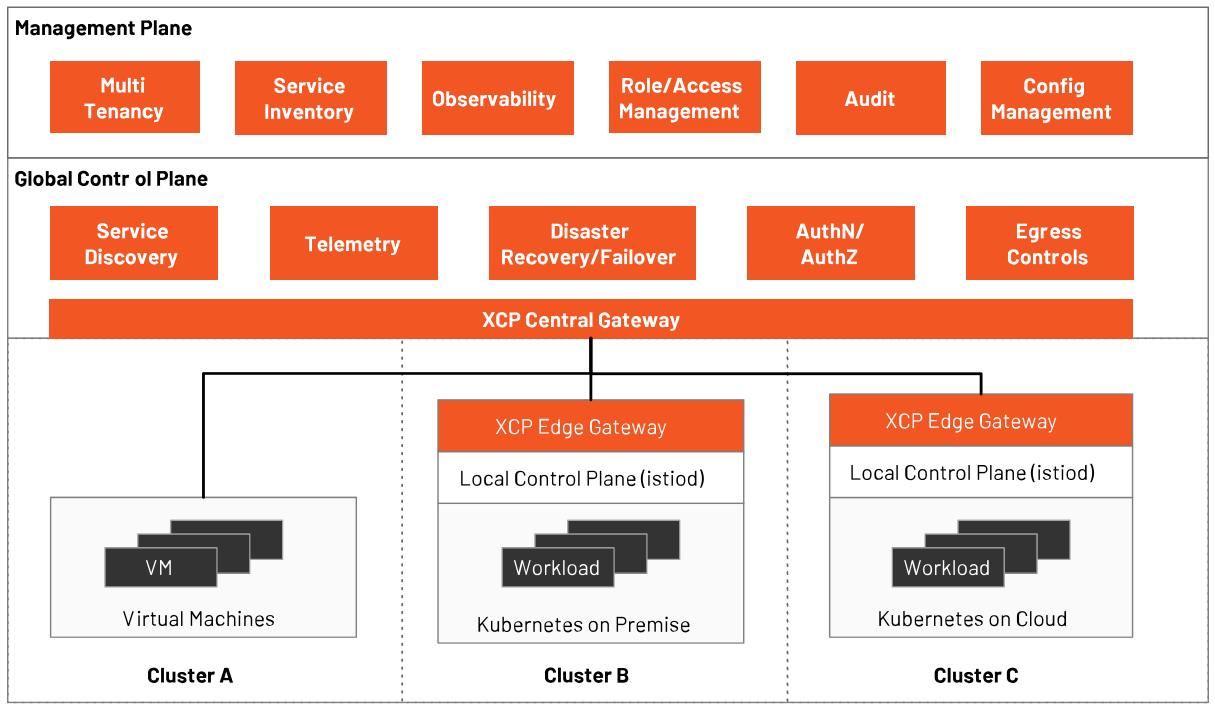

Figure 25. Tetrate Service Bridge architecture

The future of service meshes lies in being the infrastructure for zero-trust networks and hybrid clouds.

This is also the direction of Tetrate, the enterprise-level service mesh provider of the author's company. We are committed to building an application-aware network suitable for any environment and any load and providing a zero-trust hybrid cloud platform. Shown below is the architecture diagram of Tetrate's flagship product, Tetrate Service Bridge.

Tetrate was founded by the founders of the Istio project, and TSB is based on open source Istio, Envoy, and Apache SkyWalking. We also actively contributed to the upstream community and participated in the creation of the Envoy Gateway project to simplify the use of Envoy gateways (XCP in the figure above is a gateway built with Envoy).

Zero trust

Zero Trust is an important topic at IstioCon 2022. Istio is becoming an important part of zero trust, the most important of which is identity-oriented control rather than network-oriented control.

What is Zero Trust?

Zero Trust is a security philosophy, not a best practice that all security teams follow. The concept of zero trust was proposed to bring a more secure network to the cloud-native world. Zero trust is a theoretical state where all consumers within a network not only have no authority but also have no awareness of the surrounding network. The main challenges of zero trust are the increasingly granular authorization and time limit for user authorization.

Authentication

Istio 1.14 adds support for SPIRE. SPIRE (SPIFFE Runtime Environment, CNCF Incubation Project) is an implementation of SPIFFE (Secure Production Identity Framework For Everyone, CNCF Incubation Project). In Kubernetes, we use ServiceAccount to provide identity information for workloads in Pods, and its core is based on Token (using Secret resource storage) to represent workload identity. A token is a resource in a Kubernetes cluster. How to unify the identities of multiple clusters and workloads running in non-Kubernetes environments (such as virtual machines)? That's what SPIFFE is trying to solve.

The purpose of SPIFFE is to establish an open and unified workload identity standard based on the concept of zero trust, which helps to establish a fully identifiable data center network with zero trust. The core of SPIFFE is to define a short-lived encrypted identity document—SVID (SPFFE Verifiable Identity Document)—through a simple API, which is used as an identity document (based on an X.509 certificate or JWT token) for workload authentication. SPIRE can automatically rotate SVID certificates and keys according to administrator-defined policies, dynamically provide workload identities, and Istio can dynamically consume these workload identities through SPIRE.

The Kubernetes-based SPIRE architecture diagram is shown below.

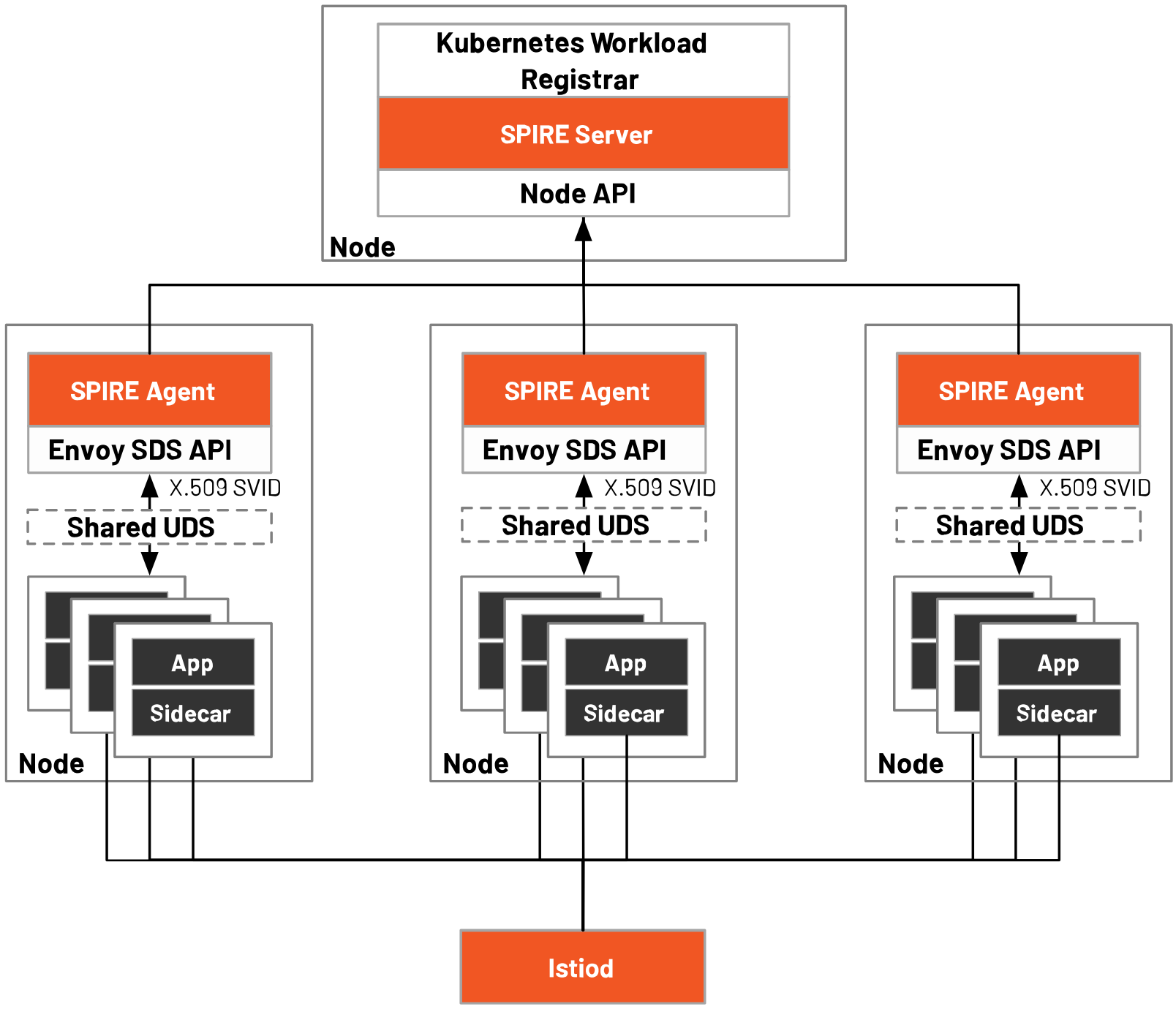

Figure 26. SPIRE deployed in Kubernetes

Istio originally used the Citadel service in Istiod to be responsible for certificate management in the service mesh, and issued the certificate to the data plane through the xDS (to be precise, SDS API) protocol. With SPIRE, the work of certificate management is handed over to SPIRE Server. SPIRE also supports the Envoy SDS API. After we enable SPIRE in Istio, the traffic entering the workload pod will be authenticated once after being transparently intercepted into the sidecar. The purpose of authentication is to compare the identity of the workload with the environment information it runs on (node, Pod's ServiceAccount and Namespace, etc.) to prevent identity forgery. Please refer to How to Integrate SPIRE in Istio to learn how to use SPIRE for authentication in Istio.

We can deploy SPIRE in Kubernetes using the Kubernetes Workload Registrar, which automatically registers the workload in Kubernetes for us and generates an SVID. The registration machine is a Server-Agent architecture, which deploys a SPIRE Agent on each node, and the Agent communicates with the workload through a shared UNIX Domain Socket. The following diagram shows the process of using SPIRE for authentication in Istio.

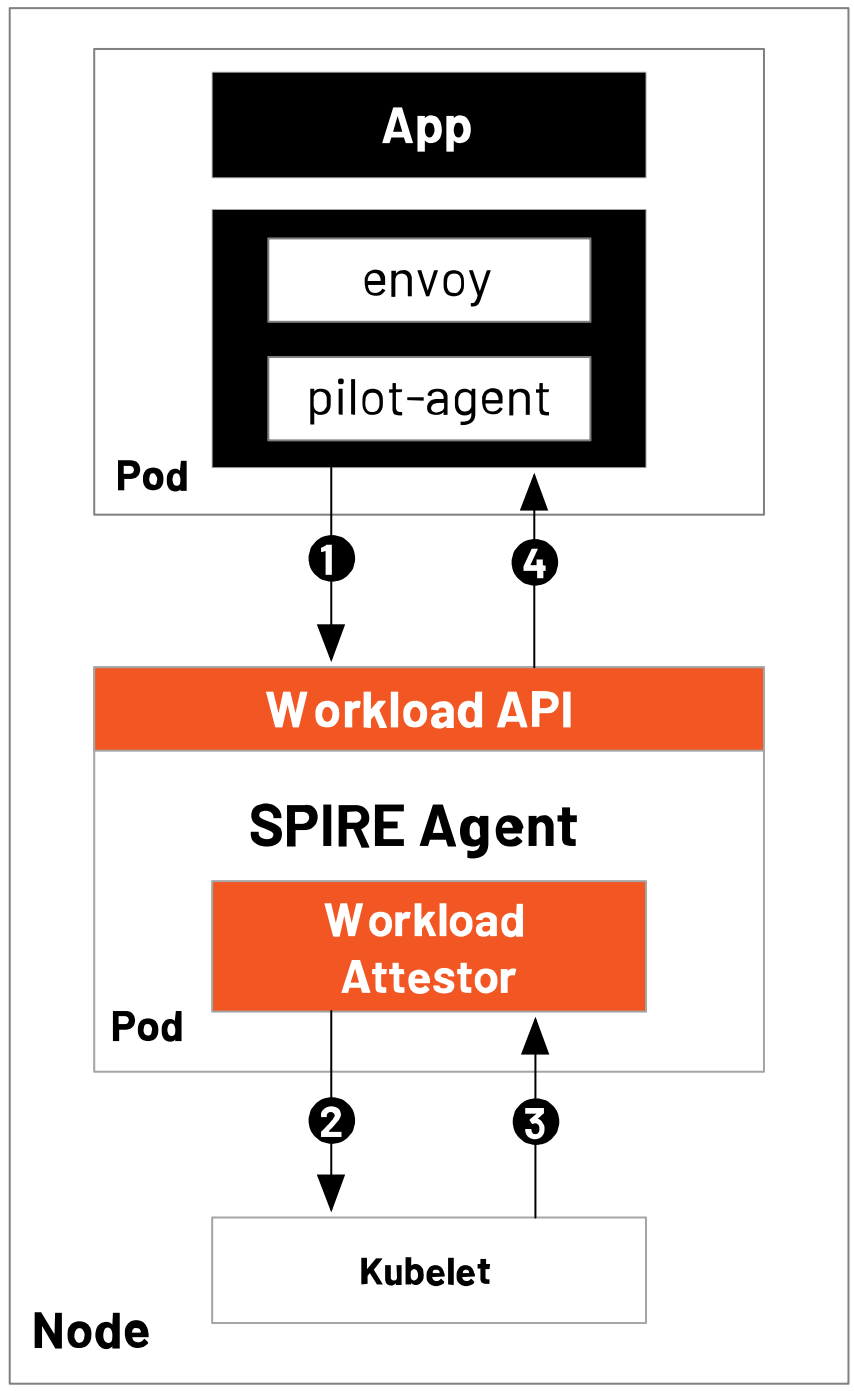

Figure 27. SPIRE-based workload authentication process in Istio

The steps to using SPIRE for workload authentication in Istio are as follows:

1. To obtain the SIVD, the SPIRE Agent is referred to as a pilot-agent via shared UDS.
2. The SPIRE Agent asks Kubernetes (to be precise, the kubelet on the node) for load information.
3. The kubelet returns the information queried from the API server to the workload validator.
4. The validator compares the result returned by the kubelet with the identity information shared by the sidecar. If it is the same, it returns the correct SVID cache to the workload. If it is different, the authentication fails.

Please refer to the SPIRE documentation for the detailed process of registering and authenticating workloads.

NGAC

When each workload has an accurate identity, how can the permissions of these identities be restricted? RBAC is used by default in Kubernetes for access control. As the name suggests, this access control is based on roles. Although it is relatively simple to use, there is a role explosion problem for large-scale clusters—that is, there are too many roles, and the types are not static, making it difficult to track and audit role permission models. In addition, the access rights of roles in RBAC are fixed, and there is no provision for short-term use rights, nor does it take into account attributes such as location, time, or equipment. Enterprises using RBAC have difficulty meeting complex access control requirements to meet the regulatory requirements that other organizations demand.

NGAC, or Next Generation Access Control, takes the approach of modeling access decision data as a graph. NGAC enables a systematic, policy-consistent approach to access control, granting or denying user management capabilities with a high level of granularity. NGAC was developed by NIST (National Institute of Standards and Technology) and is currently used for rights management in Tetrate Service Bridge. For more information on why you should choose NGAC over ABAC and RBAC, please refer to the blog post Why you should choose NGAC as your permission control model.

Hybrid cloud

In practical applications, we may deploy multiple Kubernetes clusters in various environments for reasons such as load balancing; isolation of development and production environments; decoupling of data processing and data storage; cross-cloud backup and disaster recovery; and avoiding vendor lock-in. The Kubernetes community provides a "cluster federation" function that can help us create a multi-cluster architecture, such as the common Kubernetes multi-cluster architecture shown in the figure below, in which the host cluster serves as the control plane and has two member clusters, namely West and East.

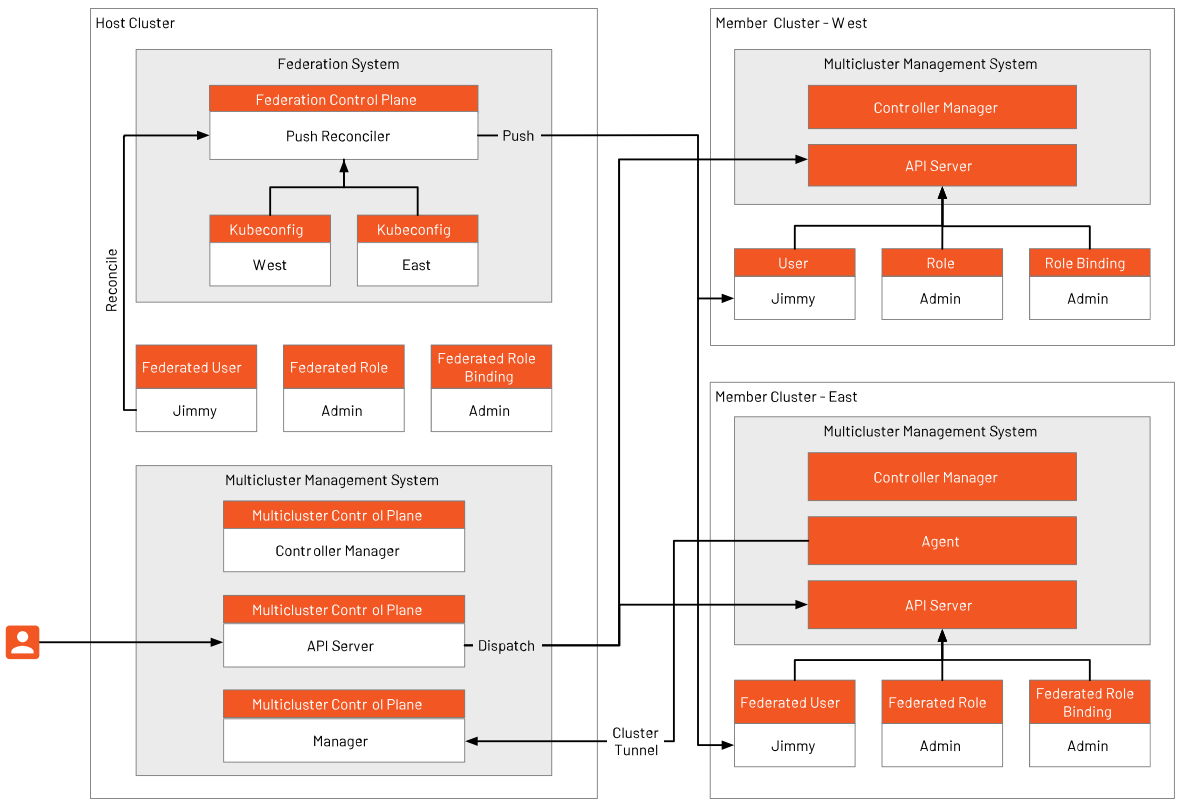

Figure 28. Kubernetes cluster federation architecture

Cluster federation requires that the networks between the host cluster and member clusters can communicate with each other but does not require network connectivity between member clusters. The host cluster serves as the API entry, and all resource requests from the outside world to the host cluster will be forwarded to the member clusters. The control plane of the cluster federation is deployed in the host cluster, and the "Push Reconciler" in it will propagate the identities, roles, and role bindings in the federation to all member clusters. Cluster federation simply "connects" multiple clusters together, replicating workloads among multiple clusters, and the traffic between member clusters cannot be scheduled, nor can true multi-tenancy be achieved.

Cluster federation is not enough to realize hybrid clouds. In order to realize hybrid clouds in the true sense, it is necessary to achieve interconnection between clusters and realize multi-tenancy at the same time. TSB builds a general control plane for multi-cluster management on top of Istio and then adds a management plane to manage multi-clusters, providing functions such as multi-tenancy, management configuration, and observability. Below is a diagram of the multi-tenancy and API of the Istio management plane.

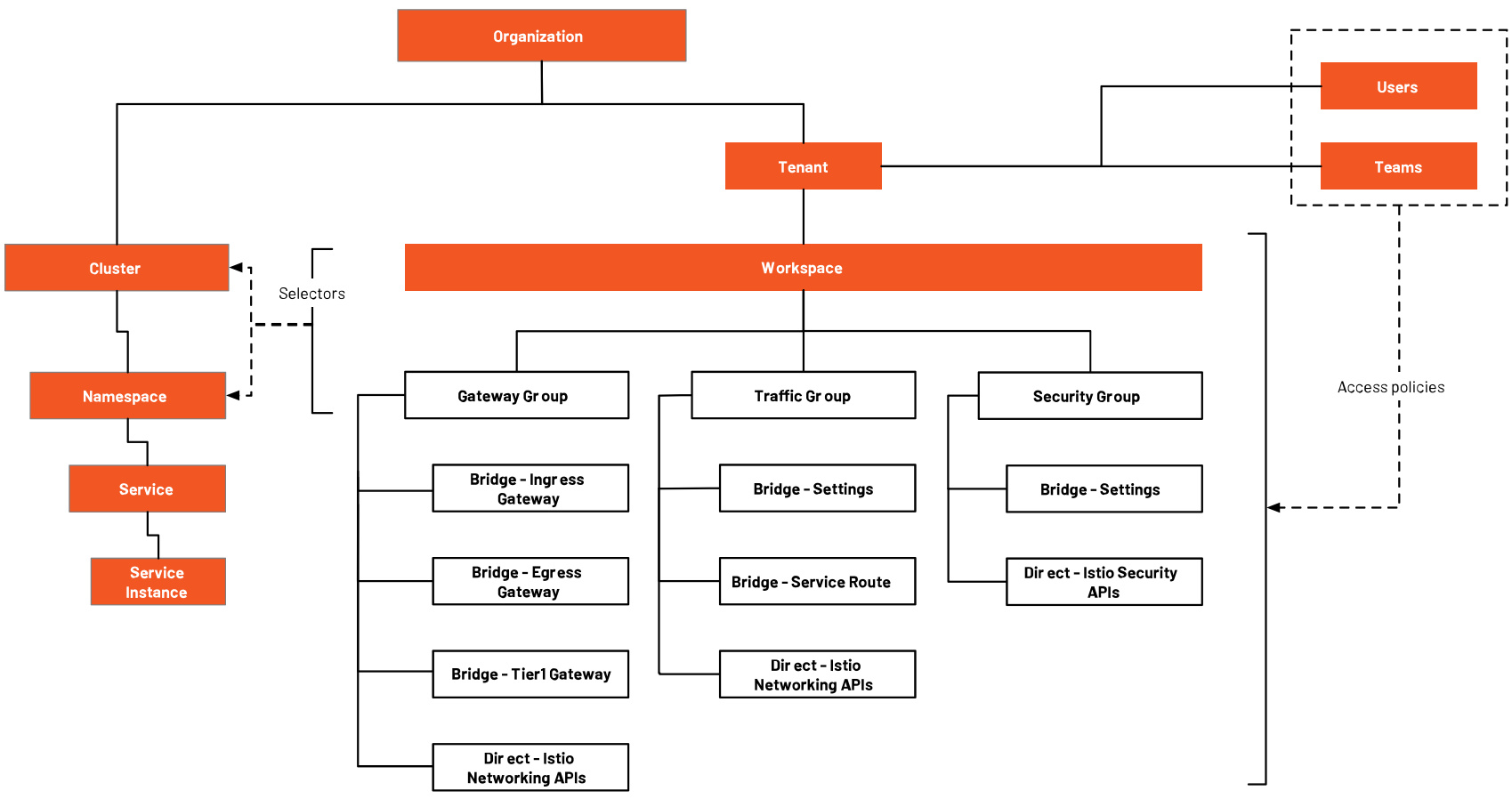

Figure 29. TSB's management plane built on top of Istio

In order to manage the hybrid cloud, TSB built a management plane based on Istio, created tenant and workspace resources, and applied the gateway group, traffic group, and security group to the workloads in the corresponding cluster through selectors. For the detailed architecture of TSB, please refer to the TSB documentation.

Thanks for reading.

About Tetrate

Tetrate enables safe and fast modernization journeys for enterprises. Built atop Envoy and Istio, its flagship product, Tetrate Service Bridge, spans traditional and modern workloads so customers can get consistent baked-in observability, runtime security, and traffic management — for all their workloads, on any environment. In addition to the technology, Tetrate brings a world-class team that leads the open Envoy and Istio projects, providing best practices and playbooks that enterprises can use to modernize their people and processes

Location: Tetrate, 691 S Milpitas Blvd, Suite 217, Milpitas, CA 95035, USA

www.tetrate.io | info@tetrate.io