Istio vs. Envoy Gateway: Gateway API on Kubernetes

2026-04-30

Comparing Istio and Envoy Gateway as Gateway API implementations: mTLS, egress, Cilium, managed cloud specifics (AKS, GKE, OVH MKS), and real client IP.

Both Istio and Envoy Gateway implement the Kubernetes Gateway API and use Envoy as their data plane. That is roughly where the similarity ends. Istio is a full service mesh that happens to implement Gateway API; Envoy Gateway is a dedicated Gateway API controller with no mesh ambitions. Choosing between them is mostly a question of scope.

This post starts with that comparison — architecture, mTLS, egress control, and resource overhead — then broadens to cover Cilium as a lighter alternative for East-West security, how the choice plays out on managed Kubernetes offerings (AKS, GKE, and OVH MKS including their cloud-native ingress and egress options), and finally how to get the real client IP through a cloud load balancer to your application.

Scope

	Istio	Envoy Gateway
Primary purpose	Service mesh + gateway	Gateway API only
Data plane	Envoy	Envoy
Control plane	istiod	envoy-gateway
East-West traffic (pod-to-pod)	✓ (mTLS via ztunnel or sidecar)	✗
North-South traffic (ingress)	✓	✓
Mesh features (mTLS, observability, traffic policies)	✓	✗
Egress control (outbound to external services)	✓ (with Egress Gateway)	✗

Envoy Gateway does one thing: manage ingress traffic into the cluster. Istio manages both the ingress boundary and the communication between every service inside the cluster.

Architecture

Istio

Istio has two operating modes.

Sidecar mode (classic): an Envoy proxy is injected into every pod as a sidecar container. All traffic in and out of the pod flows through the sidecar, which enforces mTLS, applies traffic policies,

and emits telemetry. The control plane (`istiod`) distributes configuration and certificates to every sidecar.

Ambient mode (current default for new deployments): no sidecars. Instead, a `ztunnel` DaemonSet runs on every node. The ztunnel process intercepts all pod traffic at the node level using iptables or eBPF and wraps it in HBONE (HTTP-Based Overlay Network Environment) — an mTLS tunnel based on HTTP/2 and CONNECT. For L7 policies (JWT validation, header routing between services), an optional **Waypoint Proxy** is deployed per namespace or service account.

{{ bounded_image(src="/img/istio-ambient-ztunnel-flow.drawio.png", alt="Istio Ambient Mode full traffic flow: Client reaches the Istio Gateway (Envoy Pod) on Node 1 via LoadBalancer; the Gateway's outbound traffic is intercepted by the local ztunnel DaemonSet, tunneled over mTLS/HBONE to the ztunnel on Node 2, which forwards it to Backend Pod A, B, and C", max_width=900) }}

The gateway in both modes is a dedicated Envoy pod managed by istiod, exposed via a `LoadBalancer` or `NodePort` service.

Envoy Gateway

Envoy Gateway is a Kubernetes controller that watches `Gateway` and `HTTPRoute` (and related) resources and translates them into Envoy xDS configuration. For each `Gateway` resource, it provisions a dedicated Envoy pod. There are no node-level agents, no sidecars, no mesh.

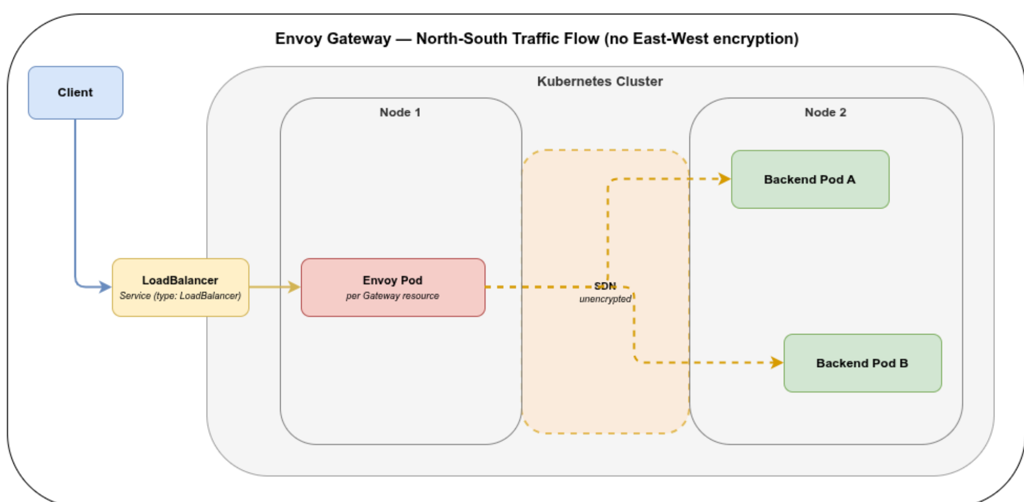

Figure 1: Envoy Gateway traffic flow: Client reaches a LoadBalancer Service, which forwards to an Envoy Pod on Node 1; the Envoy Pod routes to Backend Pod A and Backend Pod B on Node 2 over the unencrypted cluster SDN

The controller itself is small (~50–100 MB). The per-gateway Envoy pods are sized like any other proxy.

Security: the key difference

This is the most important practical distinction.

Envoy Gateway secures the ingress boundary: TLS termination, mTLS to backends if configured. It has no visibility into or control over traffic between pods. Pod-to-pod communication is left entirely to the cluster network — unencrypted by default.

Istio Ambient Mode secures both boundaries:

- **Ingress**: the Istio Gateway (Envoy pod) terminates TLS from clients — same as Envoy Gateway. The difference is what happens next: the Gateway pod's outbound connections to backend pods are also intercepted by the local ztunnel and tunneled over mTLS/HBONE. With Envoy Gateway alone, this backend leg is unencrypted plaintext over the cluster SDN.
- **East-West / pod-to-pod**: ztunnel runs on every node and automatically wraps all pod-to-pod traffic in mTLS. Identity is derived from the Kubernetes Service Account and backed by X.509 certificates issued by Istio's CA (`istiod`). No certificate = no connection. This applies across nodes without any per-pod configuration.

The result is that in an Ambient-mode Istio cluster, a compromised pod cannot silently read traffic from another pod on the same or a different node — all traffic is encrypted and mutually authenticated. With Envoy Gateway alone, a compromised node can observe all pod-to-pod traffic in plaintext.

L4 vs. L7. ztunnel operates at L4 (TCP). It authenticates and encrypts all connections but does not inspect HTTP headers or apply L7 policies between services. For that, a Waypoint Proxy is deployed selectively — only for the services that need it. This is explicitly lighter than the sidecar model, where every pod always carried full L7 proxy overhead.

Protocol-agnostic encryption. Because ztunnel intercepts at L4, it encrypts *all* TCP traffic leaving a node — regardless of the application protocol. A Pod connecting to a PostgreSQL, Redis, or any other non-HTTP service on a different node gets the same mTLS tunnel as an HTTP service, without the application or the database needing to implement TLS themselves. The encryption is transparent to both sides.

Egress traffic

The mTLS coverage described above applies only to **intra-cluster traffic** — connections between pods that both have a ztunnel instance on their node. Egress traffic to services outside the cluster is a different story.

ztunnel cannot establish a HBONE tunnel to an external destination. External services have no ztunnel to terminate the tunnel on the other side. When a pod connects to an external API, database, or any service outside the cluster, ztunnel intercepts the connection and can enforce L4 authorization policies (allow/deny), but the wire traffic to the external destination is **not wrapped in HBONE**. Whatever the application itself sends — plaintext, TLS, or application-level encryption — is what reaches the external service.

Istio Egress Gateway closes this gap. When an Egress Gateway is deployed and configured, outgoing traffic is redirected through it:

```
Pod → ztunnel (Node) ══ HBONE ══▶ Egress Gateway Pod → external service
```

The Egress Gateway is a regular pod inside the mesh, so the pod-to-gateway leg is protected by ztunnel mTLS. At the gateway, policies are enforced, traffic is logged and audited, and TLS origination towards the external service can be configured centrally — without touching individual application deployments. External destinations are registered via Istio `ServiceEntry` resources to make them known to the mesh. The Egress Gateway supports `protocol: HTTP`, `protocol: HTTPS`, and `protocol: TCP` — so plain TCP services such as databases are covered as well.

Envoy Gateway has no egress scope at all. It is an ingress-only controller — outgoing traffic from pods leaves the cluster directly, without any centralized control point.

	Envoy Gateway	Istio Ambient (no Egress GW)	Istio Ambient + Egress GW
Egress control	✗	L4 policy only	✓ full policy + audit
Wire encryption to external	app-level only	app-level only	TLS origination for HTTP, HTTPS, and plain TCP
Centralized egress logging	✗	✗	✓

Gateway API support

Both projects reach the Extended conformance level of the Gateway API specification.

Resource	Istio	Envoy Gateway
`HTTPRoute`	✓	✓
`GRPCRoute`	✓	✓
`TCPRoute`	✓	✓
`TLSRoute`	✓	✓
`ReferenceGrant`	✓	✓
Extension mechanism	`VirtualService`, `DestinationRule` (legacy) + `EnvoyFilter`	`BackendTrafficPolicy`, `SecurityPolicy`, `EnvoyExtensionPolicy`, `EnvoyProxy`

Istio's original CRDs (`VirtualService`, `DestinationRule`) still work and are widely used, but the project is moving toward Gateway API as the primary surface. Envoy Gateway was built around Gateway API from the start and uses the Policy Attachment pattern for extensions — a standard mechanism rather than project-specific CRDs.

Notable Envoy Gateway extensions:

- **`BackendTrafficPolicy`** — rate limiting, circuit breaking, connection limits, timeouts per backend
- **`SecurityPolicy`** — OIDC, JWT validation, Basic Auth, CORS at the gateway level
- **`EnvoyExtensionPolicy`** — Wasm filters, `ext_proc` for external processing
- **`EnvoyProxy`** — direct Envoy bootstrap configuration as an escape hatch

Since both Istio and Envoy Gateway use Envoy as their data plane, the underlying extension mechanisms (Wasm filters, `ext_proc`, Envoy's xDS API) are available in both. The difference is the configuration surface: Envoy Gateway exposes them through the Policy Attachment CRDs above,

while Istio uses `EnvoyFilter` — a lower-level but more flexible escape hatch that patches xDS resources directly.

Resource overhead

Component	Istio (sidecar)	Istio (ambient)	Envoy Gateway
Control plane	~500 MB (istiod)	~500 MB (istiod)	~50–100 MB
Per-node agent	—	~50 MB (ztunnel)	—
Per-pod sidecar	~50 MB	—	—
Gateway pod	~100 MB	~100 MB	~100 MB

Ambient mode removes the per-pod sidecar cost entirely. In a cluster with 100 pods the saving is roughly 5 GB of memory compared to sidecar mode — at the cost of a ztunnel DaemonSet (one per node, not one per pod).

Envoy Gateway is the lightest option when mesh features are not needed: only the controller and the gateway pods themselves are deployed.

Istio compatibility matrix

Istio maintains three actively supported minor versions at any given time. Support ends 6 weeks after the N+2 minor release. The official supported releases page lists current versions:

Istio version	Release	EOL (approx.)	Kubernetes
1.29	2026-02-16	~2026-08	1.26–1.35
1.28	2025-11-05	~2026-05	1.25–1.29
1.27	2025-08-11	2026-04-07	1.24–1.28

As of 2026-05-02

Envoy Gateway compatibility matrix

Envoy Gateway, Gateway API, Envoy Proxy, and Kubernetes are versioned independently. The official matrix lists the supported combinations:

EG version	Envoy Proxy	Gateway API	Kubernetes	EOL
v1.7	distroless-v1.37.0	v1.4.1	v1.32–v1.35	2026-08-05
v1.6	distroless-v1.36.4	v1.4.0	v1.30–v1.33	2026-05-13

As of 2026-05-02

Each Envoy Gateway minor release supports approximately four Kubernetes minor versions and has a support window of roughly six months.

Maturity

	Istio	Envoy Gateway
First release	2017	2022
Latest release	1.29	v1.7
CNCF status	Graduated	Incubating
Gateway API support since	Istio 1.16 (late 2022)	v1.0 GA (2024)
Ambient mode GA	Istio 1.22 (2024)	—
Production-ready	Yes	Yes (since 2024)

As of 2026-05-02

Istio is one of the most deployed pieces of cloud-native infrastructure. Envoy Gateway is younger but reached GA in 2024 and is backed by the Envoy project directly.

When to use which

Use Envoy Gateway when:

- You need a well-supported Gateway API implementation for ingress only.
- You do not need mTLS or traffic policies between your own services.
- You want minimal operational overhead — no mesh control plane, no node agents.
- Your team is not already invested in Istio.

Use Istio (Ambient mode) when:

- You need mTLS between services (zero-trust East-West security).
- You want L7 traffic policies between internal services (retries, circuit breaking, header-based routing) — Envoy Gateway supports these only at the ingress boundary, not for East-West service-to-service traffic.
- You need mesh-level observability (distributed tracing, service-to-service metrics).
- You are running in an environment with strict security requirements for pod-to-pod traffic.
- You need centralized egress control — audit logging, policy enforcement, or TLS origination for outgoing traffic to external services, including plain TCP protocols such as databases.

An option for East-West security without a full service mesh

Cilium is a security-focused CNI built on eBPF that enforces network policies directly in the Linux kernel with minimal overhead. Combined with Envoy Gateway for ingress, you get a capable security stack:

- **WireGuard encryption** — transparent node-to-node encryption for all East-West traffic, no sidecar or mesh control plane required
- **L7 NetworkPolicy** — allow/deny based on HTTP methods, paths, or DNS names via eBPF, without a Waypoint Proxy
- **Mutual authentication** via SPIFFE/SPIRE — service identity without istiod
- **Hubble** — network flow visibility and observability

	Cilium + Envoy Gateway	Istio Ambient + Envoy Gateway
East-West encryption	WireGuard (L3/L4)	mTLS/HBONE (L4)
L7 policies (E-W)	✓ (eBPF)	✓ (Waypoint Proxy)
Service identity	SPIFFE/SPIRE	Kubernetes SA + istiod CA
Egress control	limited	✓ (Egress Gateway)
Control plane overhead	very low	moderate (istiod + ztunnel)

The trade-off: Cilium is lighter and requires no mesh control plane, but Istio Ambient has more mature egress control and a richer traffic management feature set.

i Cilium + Envoy Gateway: no centralized egress

In this combination there is no egress gateway. Cilium can enforce L3/L4 egress NetworkPolicy — including DNS-based rules via `CiliumNetworkPolicy` with `toFQDNs` — to allow or deny outbound connections by destination. But pods connect **directly** to external services; no proxy intercepts the traffic. There is no centralized audit log, no forced TLS origination, and no single control point for egress. If those are requirements, Istio with an Egress Gateway is the right choice.

Managed cloud: AKS and GKE

Both Azure Kubernetes Service and Google Kubernetes Engine ship their own cloud-native Gateway API implementations — alternative GatewayClasses that provision the cloud provider's load balancer directly from `HTTPRoute` resources, without running an Envoy Gateway or Istio pod. They also add infrastructure-level egress controls independent of the service mesh.

	AKS	GKE
Cloud-native Gateway API	Application Gateway for Containers (AGfC, GA)	GKE Gateway API (GA)
L4 ingress (LoadBalancer)	Azure Load Balancer	Google Network Load Balancer
Advanced L7 (rate limiting, OIDC, ext_proc)	Envoy Gateway / Istio	Envoy Gateway / Istio
Default CNI	Azure CNI	VPC-native
Cilium support	Azure CNI + Cilium dataplane (GA)	Dataplane V2 (Cilium-based fork, default)
Managed Istio	Managed Istio add-on (GA)	Cloud Service Mesh (GA)
Cloud-managed egress	Azure NAT Gateway / Azure Firewall	Cloud NAT / Cloud Armor
Egress logging	NSG Flow Logs / Azure Monitor	VPC Flow Logs / Cloud Logging

AKS. Application Gateway for Containers (AGfC) is Azure's Gateway API implementation — GA since 2024. It provisions an Azure Application Gateway (L7 HTTPS load balancer) per `Gateway` resource and handles TLS termination and HTTPRoute-based routing without any in-cluster proxy pod. For features beyond basic routing — rate limiting, OIDC authentication, Wasm filters, `ext_proc` — Envoy Gateway or Istio is still the right tool. The Azure CNI + Cilium dataplane combination is GA and gives you Cilium NetworkPolicy + eBPF; full open-source Cilium features

(ClusterMesh, SPIFFE/SPIRE) require manual installation on top. The Managed Istio add-on (GA since AKS 1.28) handles the istiod lifecycle and supports Ambient mode (preview as of 2026-05).

GKE. The GKE Gateway API is GA and ships multiple GatewayClasses that provision Cloud Load Balancers directly from `HTTPRoute` resources: `gke-l7-regional-external` (regional HTTPS LB), `gke-l7-global-external-managed` (global Anycast HTTPS LB with CDN integration), and `gke-l7-rilb` (internal HTTPS LB). As with AGfC on AKS, advanced extension policies require Envoy Gateway or Istio on top. Dataplane V2 is the default CNI on new clusters — it is Google's Cilium-based eBPF dataplane, but not open-source Cilium; Cilium ClusterMesh between GKE (Dataplane V2) and AKS (open-source Cilium) is not natively compatible. Cloud Service Mesh (formerly Anthos Service Mesh) is Google's managed Istio offering and is GA.

Multi-cluster: AKS ⇔ GKE

Cross-cluster traffic between AKS and GKE runs over a VPN or dedicated interconnect (Azure ExpressRoute + Google Cloud Interconnect, or a site-to-site VPN Gateway on each side). The mesh or CNI layer then handles what happens inside that tunnel.

	Cilium ClusterMesh	Istio multi-cluster	Envoy Gateway
AKS ↔ GKE compatible	✗ (CNI fork mismatch)	✓ (CNI-agnostic)	✗ (per-cluster ingress only)
Cross-cluster mTLS	✓ (same CNI only)	✓	✗
Setup complexity	low (same CNI)	moderate	—

Cilium ClusterMesh requires the same open-source Cilium on all clusters. Because GKE runs Dataplane V2 (Google's fork), ClusterMesh between AKS and GKE does not work natively. Istio multi-cluster — either Primary-Remote or Multi-Primary — is CNI-agnostic and works across providers. With AKS Managed Istio on one side and GKE Cloud Service Mesh on the other, you get cross-cluster mTLS and unified traffic policy without being tied to a specific CNI.

For multi-cloud AKS + GKE with East-West security or egress requirements, Istio is the pragmatic path. For single-cluster deployments where cross-cluster mesh is not needed, the cloud-native egress controls (NAT Gateway / Cloud NAT) combined with Cilium + Envoy Gateway form a viable lightweight stack.

Managed cloud: OVH MKS

OVH Managed Kubernetes Service (MKS) is a more vanilla managed Kubernetes compared to AKS or GKE — there is no cloud-native L7 Gateway API implementation equivalent to Application Gateway for Containers or GKE Gateway API. Ingress is handled by a self-installed controller (Envoy Gateway, Istio, Traefik, …) with the OVHcloud Load Balancer (based on OpenStack Octavia) acting as the L4 frontend, provisioned via the OpenStack Cloud Controller Manager.

	OVH MKS
Cloud-native Gateway API	✗ — L4 LB only; L7 routing via self-installed controller
L4 ingress (LoadBalancer)	OVHcloud Load Balancer (OpenStack Octavia)
Proxy Protocol v2	✓ (via annotation)
Advanced L7 (rate limiting, OIDC, ext_proc)	Envoy Gateway / Istio — self-installed
Default CNI	Cilium (full open-source)
Managed Istio	✗ — self-installed
Cloud-managed egress	✗
Egress logging	✗

Load balancer integration and Proxy Protocol v2. The OVHcloud LB integrates with Kubernetes through the OpenStack CCM. When using the Kubernetes Gateway API, the `spec.infrastructure.annotations` field (GA in Gateway API v1.1) propagates annotations directly onto the provisioned LoadBalancer Service, so the `Gateway` resource is the single point of configuration:

```
spec:
  gatewayClassName: istio
  infrastructure:
    annotations:
      # MKS >= 1.31 (OpenStack CCM / Octavia)
      loadbalancer.openstack.org/proxy-protocol: "v2"
      # MKS < 1.31: service.beta.kubernetes.io/ovh-loadbalancer-proxy-
protocol: "v2"
```

With Proxy Protocol v2 enabled, the real client IP is preserved through the load balancer and available to the in-cluster proxy (Istio Gateway or Envoy Gateway) without relying on `X-Forwarded-For` headers set by the LB.

CNI. OVH MKS ships full open-source Cilium as its default CNI — not a fork. This means Cilium ClusterMesh works natively between OVH clusters. A ClusterMesh between OVH and AKS is theoretically possible if AKS is configured with standalone open-source Cilium instead of the Azure CNI + Cilium dataplane add-on, but that is a non-default setup on AKS.

Egress. There is no managed egress gateway or NAT service equivalent to Azure NAT Gateway or Google Cloud NAT. Egress traffic leaves pods directly — the same gap described for Cilium + Envoy Gateway above applies here. Istio with an Egress Gateway remains the option if centralized egress control is required.

How to get the real client IP to your application

When a client connects through a cloud load balancer and then an in-cluster proxy (Envoy Gateway or Istio), the source IP is NATted at the node level by default — the application sees the proxy's pod IP, not the real client address. Three mechanisms address this, each with different trade-offs and protocol support.

X-Forwarded-For header (HTTP/HTTPS only)

The in-cluster proxy adds an `X-Forwarded-For` header based on the TCP source IP it receives. If node-level SNAT is active, that source IP is the node IP — so XFF ends up wrong. The prerequisite is either `externalTrafficPolicy: Local` (see below) so the proxy sees the real client IP, or a cloud L7 load balancer (AGfC, GKE Gateway API) that sets XFF itself before forwarding to the backend.

Configuring the number of trusted upstream hops:

- **Envoy Gateway**: `ClientTrafficPolicy` → `clientIPDetection.xForwardedFor.numTrustedHops`
- **Istio**: `meshConfig.defaultConfig.gatewayTopology.numTrustedProxies`

XFF only works for HTTP/HTTPS — not for raw TCP connections or other non-HTTP protocols.

externalTrafficPolicy: Local (any protocol)

Setting `externalTrafficPolicy: Local` on the Kubernetes LoadBalancer Service disables node-level SNAT. The cloud LB routes traffic directly to the node running the gateway pod; that pod sees the real client IP as the TCP source IP, for any protocol.

With the Kubernetes Gateway API, this is set via `spec.infrastructure.annotations` on the `Gateway` resource so there is no need to patch the provisioned Service directly:

```
spec:
  infrastructure:
    annotations:
      service.beta.kubernetes.io/external-traffic: "Local"
```

Trade-off: the load balancer health-checks per node rather than per pod. If gateway pods are not spread evenly across nodes, traffic distribution becomes uneven.

Proxy Protocol v2 (any protocol)

The L4 load balancer prepends a binary header to the TCP stream that contains the real client IP and port. Both the LB and the backend must explicitly enable it.

LB side — annotation on the `Gateway` resource's `spec.infrastructure.annotations`:

Platform	Annotation
OVH MKS ≥ 1.31	`loadbalancer.openstack.org/proxy-protocol: "v2"`
OVH MKS < 1.31	`service.beta.kubernetes.io/ovh-loadbalancer-proxy-protocol: "v2"`
AKS — standard public L4 LB	not natively supported — use `externalTrafficPolicy: Local`
AKS — Private Link Service	`service.beta.kubernetes.io/azure-pls-proxy-protocol: "true"`
GKE — standard L4 LB	not natively supported — use `externalTrafficPolicy: Local`
GKE — Private Service Connect	✓ all PSC producer types support PROXY Protocol v2

Both Azure and GCP offer Proxy Protocol v2 support through their private connectivity services — Azure via **Private Link Service (PLS)**, GCP via **Private Service Connect (PSC)**. Both are designed for **private connectivity**: exposing a Kubernetes service to other VNets, projects, or tenants via a Private Endpoint — not for public internet ingress.

The Proxy Protocol v2 header in both cases carries the consumer-side source IP (the Private Endpoint's IP in the consumer's VNet) and a connection identifier (PLS link ID / PSC connection ID) as TLV-encoded metadata — not the original internet client IP.

AKS PLS requires two annotations: `service.beta.kubernetes.io/azure-pls-create: "true"` to create the PLS resource, and `service.beta.kubernetes.io/azure-pls-proxy-protocol: "true"` to enable Proxy Protocol v2 on it.

GKE PSC requires setting up an internal LoadBalancer as the PSC producer backend and a `ServiceAttachment` resource with PROXY Protocol enabled — this cannot be done purely via Service annotations and requires a separate GKE `ServiceAttachment` CR.

Backend side:

- **Envoy Gateway**: `ClientTrafficPolicy` with `enableProxyProtocol: true`
- **Istio**: via `EnvoyFilter` patching the gateway listener

Proxy Protocol v2 works for any protocol and avoids the uneven load-distribution trade-off of `externalTrafficPolicy: Local`, but requires explicit support from the load balancer.

Summary

Mechanism	Protocol	LB support required	Trade-off
`X-Forwarded-For`	HTTP/HTTPS only	✗	Requires correct `numTrustedHops` + no SNAT
`externalTrafficPolicy: Local`	any	✗	Uneven load distribution
Proxy Protocol v2	any	✓	Both LB and backend must be configured

AGfC (AKS) and GKE Gateway API are L7 load balancers that terminate TLS and set XFF themselves — no extra configuration needed for HTTP workloads. For TCP workloads behind these LBs, `externalTrafficPolicy: Local` is the practical option.

Envoy Gateway / Istio behind a standard L4 LB (AKS / GKE): use `externalTrafficPolicy: Local` via `spec.infrastructure.annotations` on the `Gateway` resource.

OVH MKS: Proxy Protocol v2 via the Octavia annotation is the preferred approach — it works for all protocols and avoids the load-distribution trade-off of `externalTrafficPolicy: Local`.