CLOUD NATIVE
BELGIUM

Gateway API in the Enterprise

From the path of a packet to AI agents — north-south traffic, where the data plane really lives, and the security boundaries that hold it together.

Philippe Bogaerts · BruCON

Cloud Native Belgium · 18 June 2026 · 19.45

meetup.com/cloudnative-belgium

Who am I

Philippe Bogaerts

About

- Cloud Consulting Systems Engineer @Fortinet
- Agentic AI Researcher / Trainer
- BruCON Co-Founder and **part of a fantastic team of co-organizers that deserve all credit !!**
- Focus: https://www.brucon.org/training-details/building-securing-ai-systems-v2026

Find me

- Email: philippe.bogaerts@radarsec.com
- LinkedIn: https://www.linkedin.com/in/philippebogaerts/
- GitHub: https://github.com/xxradar
- Lab: https://github.com/k8ssecurity/gateway_api_tutorial

What we'll cover

A packet's journey, then the controls that make it safe and multi-tenant

1

Path of the packet

How north-south traffic flows — and where Gateway API slots into the classic edge

2

Where the data plane lives

In-cluster proxy vs. cloud/appliance — your physical F5 can be the data plane

3

The landscape

Popular Gateway API implementations and the new AI / MCP gateways

4

Security & multi-tenancy

ReferenceGrant, RBAC, and controller-based namespace selection

5

TLS & network policy

Terminate vs. passthrough, and where L3/L4 policy fits around L7

6

Gateway API vs. service mesh

North-south vs. east-west, and L7 visibility with Cilium/eBPF

01

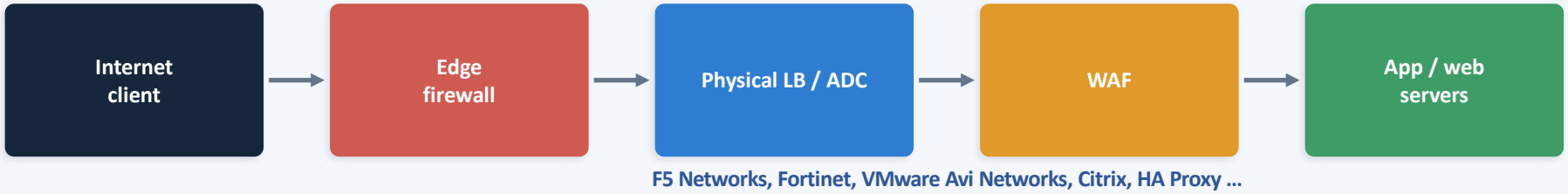
Path of the packet

Where Gateway API fits in the classic enterprise edge

- The traditional north-south chain: firewall → load balancer → app tier
- Gateway API is the vendor-neutral config layer for that edge
- Two tiers: a controller that configures, a proxy that forwards

The traditional path: on-prem, non-Kubernetes

Physical / virtual appliances and a static server pool — the pre-cloud-native baseline



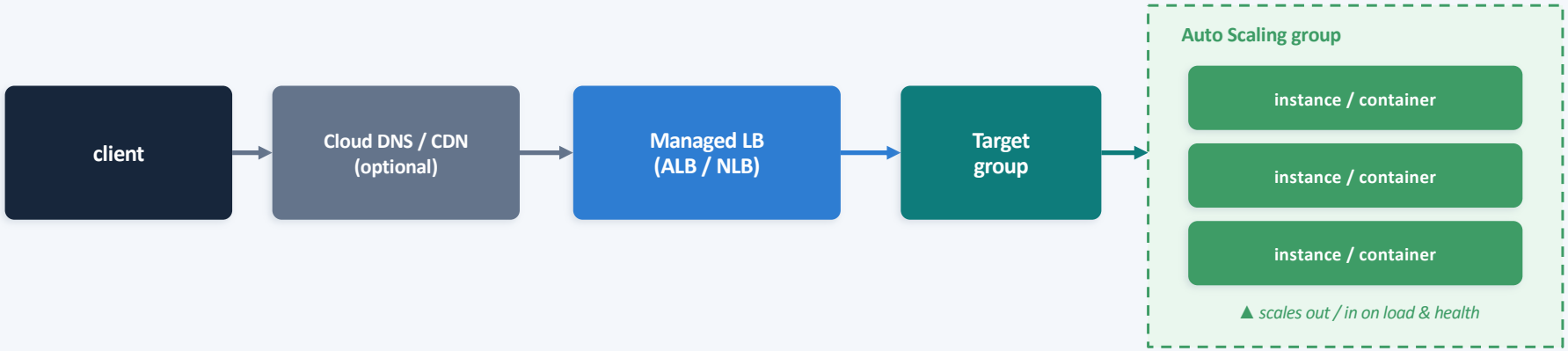
TLS termination · health checks · L7 routing · VIPs

The mental model you already have — and its limits

- On-premises: the LB / ADC is a box you rack (or a VM) — fixed capacity, sized for peak. It owns the VIP, terminates TLS, health-checks the pool, routes by host/path.
- Backends are a static server pool; scaling is manual — provision a server, edit the pool by hand.
- The shape is right, but it's rigid and vendor-specific (iRules, content switching, ticket-driven). Cloud makes this edge managed & elastic (next); Kubernetes makes it portable.

Toward cloud-native: managed LB + autoscaling

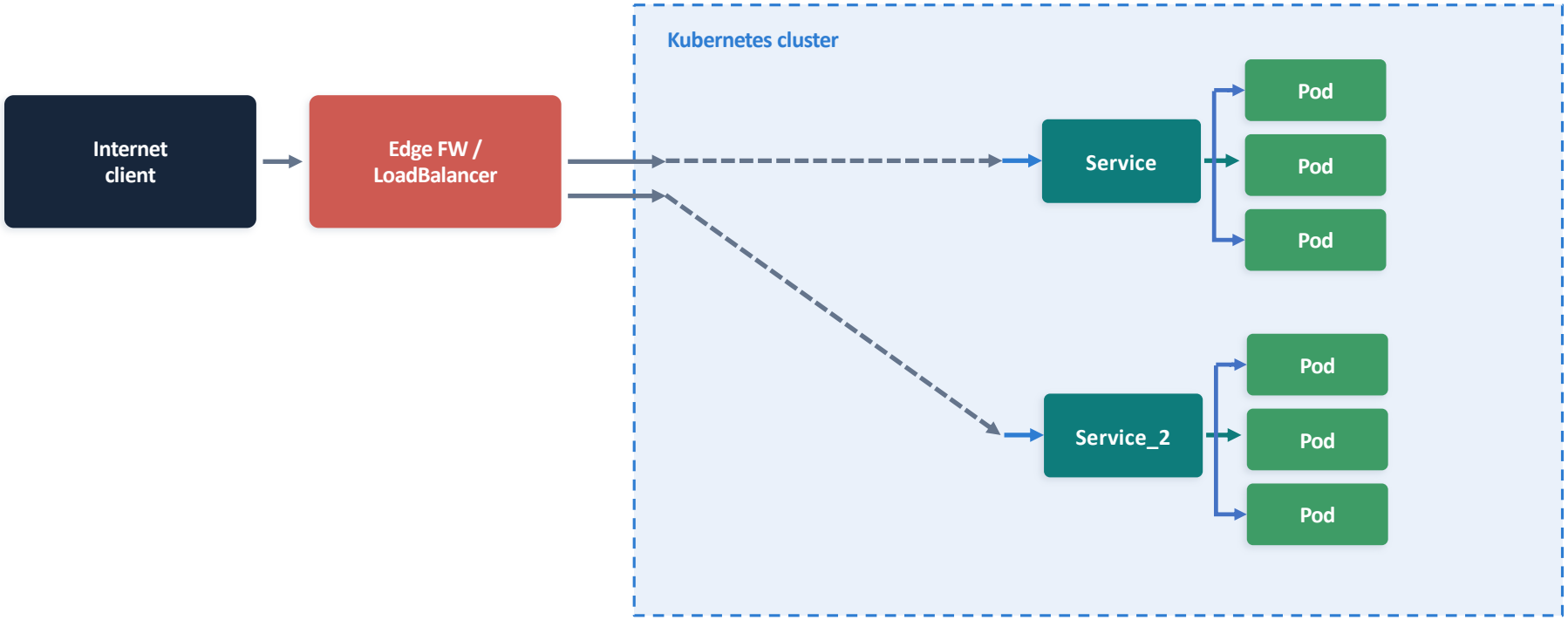
Public cloud keeps the same chain — but the edge becomes a managed service and the pool gets elastic



What changed vs. on-prem

- The LB is a managed service (ALB/NLB) — nothing to rack, pay-per-use, HA by default. Still the policy point: TLS, health checks, host/path routing.
- The backend is an Auto Scaling group behind a target group — elastic, scales on load/health instead of a hand-edited static pool.
- Config moves to the cloud API / IaC (Terraform) — better than iRules, but still cloud-specific. Kubernetes + Gateway API makes it portable (next).

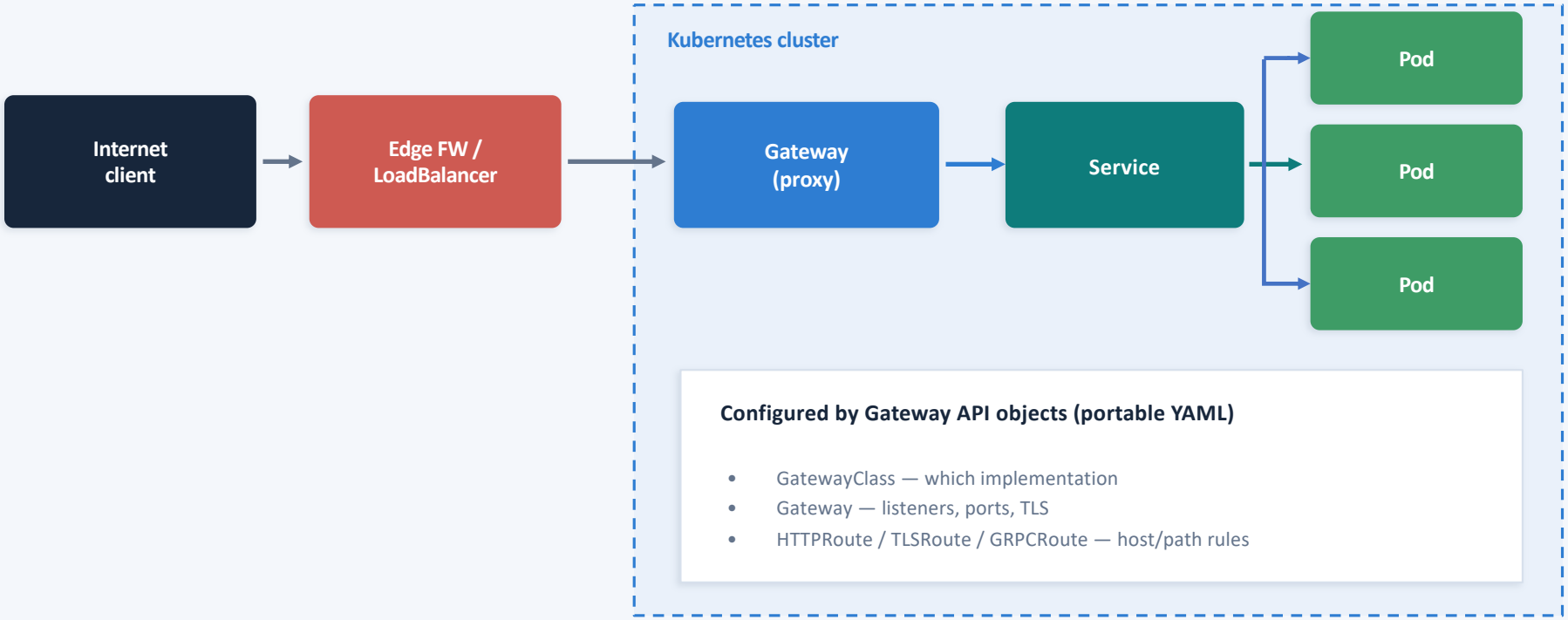
Toward Kubernetes: LB + Pods



L4 LoadBalancing

Where the Gateway API fits

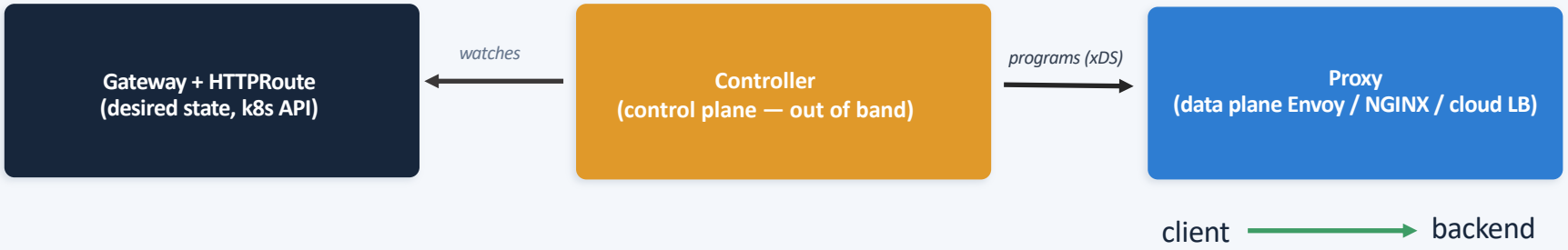
On-prem → cloud → Kubernetes: the managed, elastic edge now lives in the cluster — configured by portable YAML



Same job as the ADC — VIP, TLS, L7 routing — but declared once and portable across vendors and clouds.

Two tiers: controller vs. data plane

The Gateway YAML is desired state — a controller programs the proxy; the proxy moves packets



user traffic only ever touches the proxy — never the controller

Which controller? The GatewayClass decides.

- A Gateway -> GatewayClass -> a controller (e.g. gateway.envoyproxy.io/gatewayclass-controller, agentgateway.dev/agentgateway).
- Only the controller programs its proxy
- The proxy never reads your YAML; it only receives controller-rendered config.

From Ingress to Gateway API

Advanced Ingress meant unvalidated vendor annotations — now: a typed core plus typed extensions

The Ingress way

- One object mixes infra + TLS + routing
- Everything advanced = vendor annotations
- Stringly-typed: a typo is silently ignored
- No schema validation; meaning differs per controller
- Not portable across implementations

1 · Standardized typed filters

Portable on any conformant impl — header modify · URL rewrite · redirect · request mirror · weighted / canary split

2 · Structured vendor extensions

Typed CRDs via Policy Attachment / filters.ExtensionRef — validated & explicit, but NOT portable — JWT / OIDC auth · rate limiting · CORS · Lua / Wasm transforms

3 · Roadmap — graduating to standard

Popular extensions becoming first-class, e.g. GEP-1494 HTTP auth (ext-authz)

JWT is mid-journey (being standardized via GEP-1494). Lua / Wasm deliberately stays vendor — arbitrary code can't be portable.

02

Where does the data plane live?

Same API, two very different packet paths

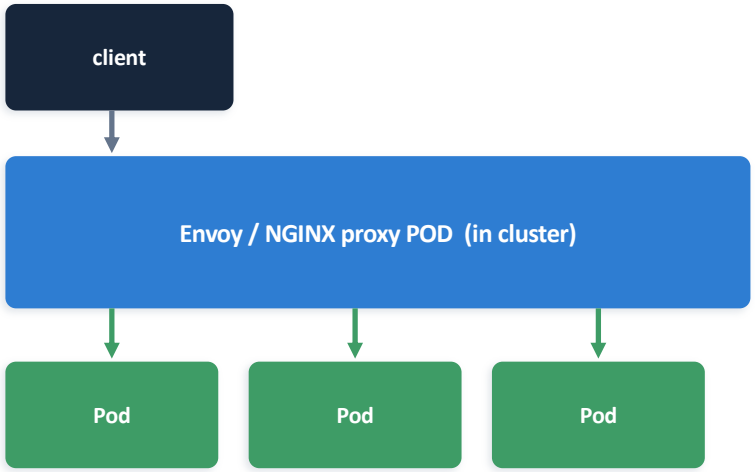
- In-cluster proxy pods (Envoy, NGINX) — the proxy runs on your nodes
- Out-of-cluster managed data plane — a cloud LB or a physical appliance
- The controller just programs it; the packet path changes completely

Gateway: In-cluster vs. out-of-cluster data plane

Identical Gateway/HTTPRoute YAML, the proxy may not even run in your cluster.

In-cluster proxy model

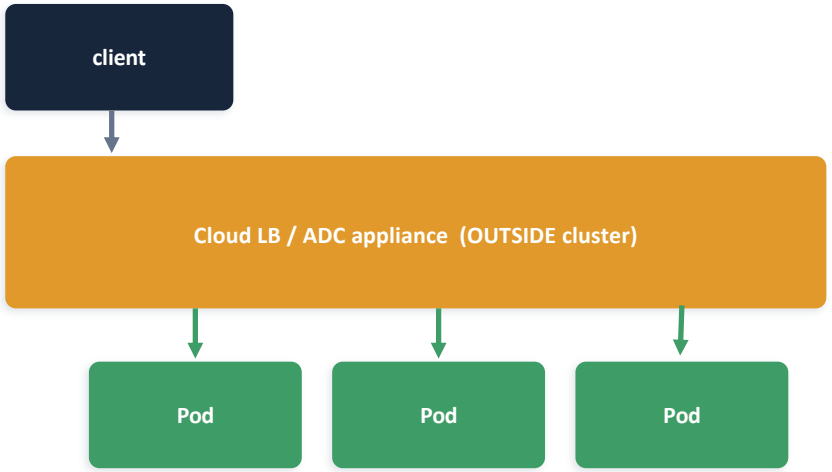
Envoy Gateway · NGINX GF · Cilium · Istio · Traefik · kgateway



L7 hop happens on your nodes — you scale & secure it

Out-of-cluster managed data plane

GKE · AWS ALB/NLB · Azure AppGW · F5 · NetScaler · Avi - Fortinet

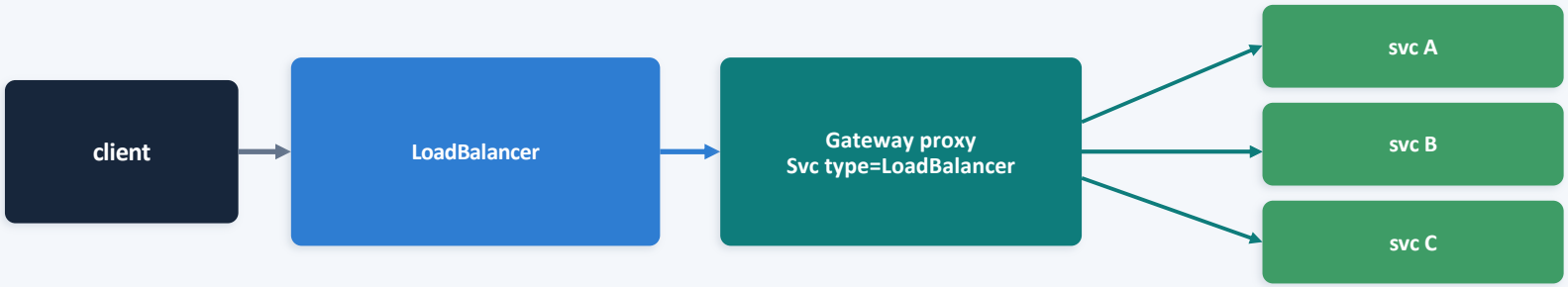


hits pods directly (NEGs / IP target groups) — no in-cluster L7 hop

A controller still runs in-cluster in BOTH cases -> it only configures the proxy.

Exposing the Gateway still needs a LoadBalancer

An in-cluster gateway sits behind a Service type=LoadBalancer -> so you need an LB implementation



One external entry → the gateway. L7 host/path fan-out to every service happens inside, as declarative Gateway API so the external LB config stays tiny.

CNI / pod-IP targeting

- LB sends straight to gateway pod IPs, cloud NEGs, AWS IP-mode target groups, Cilium. **No extra hop, client IP preserved.**
- *…or the out-of-cluster model, where the cloud LB IS the data plane.*

NodePort bridge

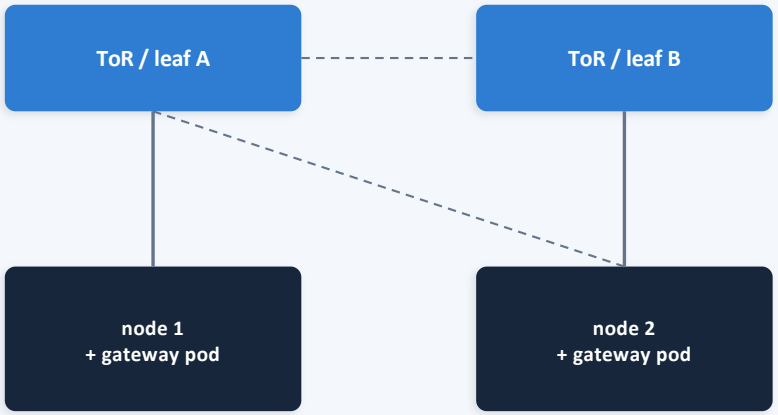
- A 'dumb' external LB / appliance targets every node on a NodePort
- **Extra hop · SNAT / client-IP loss** (use externalTrafficPolicy: Local)
- With an eBPF data plane (Cilium kube-proxy replacement), DSR + Maglev preserve the client source IP and skip the extra hop without externalTrafficPolicy

Same requirement as Ingress and it disappears in the out-of-cluster managed model: there the controller programs the external LB directly, no in-cluster LoadBalancer Service needed.

Bare-metal redundancy: BGP, ECMP & anycast

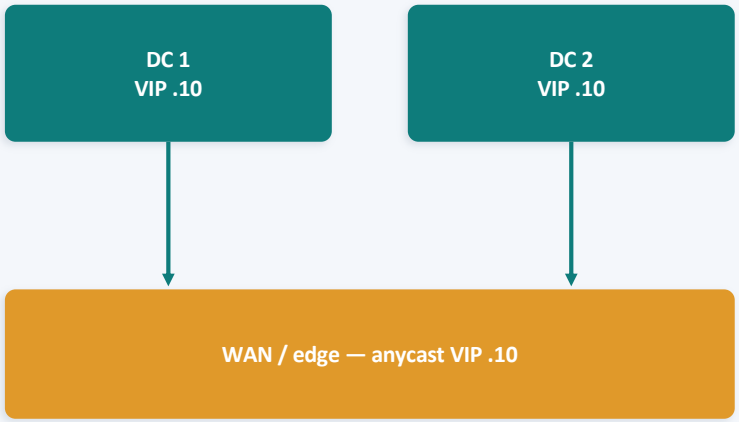
On-prem, the CNI / LB peers with your fabric and advertises the gateway VIP — no cloud LB, no NodePort

Dual ToR · ECMP



each node eBGP-peers BOTH ToRs · BFD · ECMP — VIP 203.0.113.10 advertised, fabric balances across nodes

Dual DC · anycast



same VIP from both DCs · BGP health-based withdrawal = DC failover

What's advertised

- LoadBalancer VIP (the gateway's Service) -> ECMP across nodes
- Optionally Pod CIDRs → fabric routes direct to pods (no NodePort/SNAT)
- BFD → sub-second failover

Who speaks BGP

- Cilium BGP CiliumBGPClusterConfig + CiliumBGPAdvertisement (v2)
- Calico BGPPeer / BGPConfiguration (also routes pod IPs)
- MetalLB BGP mode (BGPAdvertisement)

Gateway API stays unaware -> L7 config; BGP owns VIP reachability, ECMP & DC failover.

Cloud LB or physical LB as the data plane

Controller in-cluster, data plane outside: *the pattern that fits existing enterprise edges*

In-cluster

- Envoy Gateway · NGINX GF
- Cilium · Istio · Traefik · kgateway
- F5 BIG-IP Next → runs as pods!

Proxy IS in the packet path

VS

Public cloud (managed)

- GKE Gateway → Google Cloud LB
- AWS LBC → ALB / NLB (GA, Mar 2026)
- Azure App Gateway for Containers

Data plane = cloud fabric, not pods

ADC / appliance vendors

- F5 BIG-IP
- Citrix NetScaler
- Avi / NSX ALB (Service Engines)
- Fortinet FortiADC – FortiWeb
- HA Proxy

Often Ingress/CRD-driven today -> Roadmap

03

The landscape

Who implements the Gateway API — and the new AI gateways

- Gateway API is GA (v1.0, Oct 2023); current release v1.5 (Feb 2026)
- The field consolidated around an Envoy-native core
- A new category: AI / LLM / MCP gateways for agent traffic

Popular Gateway API implementations

GA and on a 4-month release train — v1.5 is current (2026). Conformance list is the source of truth.

Envoy Gateway

data plane: Envoy

The standard-bearer — pure Gateway API on Envoy

Istio

data plane: Envoy (+ambient)

Only impl conformant for BOTH Gateway & Mesh

Cilium

data plane: eBPF + Envoy

CNCF Graduated, sidecarless, L7 via eBPF

NGINX Gateway Fabric

data plane: NGINX

The canonical non-Envoy option (F5/NGINX)

Traefik

data plane: Traefik (Go)

Most GitHub stars; non-Envoy data plane

kgateway

data plane: Envoy

Rising CNCF project, strong AI/MCP focus

Takeaway: the ecosystem consolidated on Envoy (Envoy Gateway, Istio, Cilium-L7, kgateway, Contour, Gloo) — with NGINX, Traefik & HAProxy as the main non-Envoy data planes. Also conformant: HAProxy Ingress, Kong (via Operator), GKE.

The new category: AI / LLM / MCP gateways

Same gateway instincts, applied to agent traffic — the 2026 differentiator is MCP/A2A + guardrails

agentgateway

Rust · Gateway API

LLM + MCP + A2A native; CNCF/LF (AAIF)

Envoy AI Gateway

Envoy · Gateway API

GenAI routing + MCP; Envoy-native

LiteLLM

Python (standalone)

OSS default; 100+ providers, OpenAI-compat

Portkey

SaaS

Semantic caching + MCP GA; managed leader

Kong AI Gateway

Kong/NGINX

AI + MCP on a mature API platform

Cloudflare AI Gateway

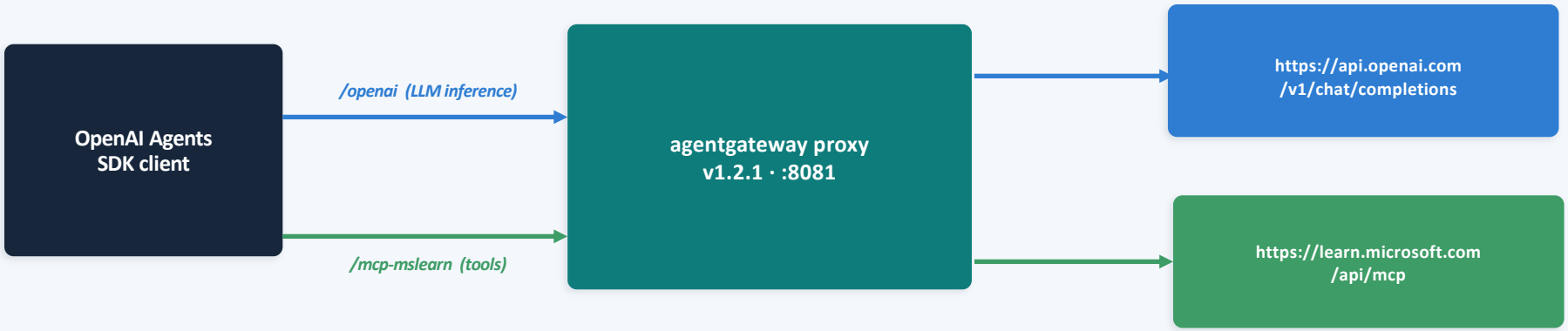
Edge SaaS

Zero-infra: caching, rate limit, guardrails

What an AI gateway adds beyond fan-out: provider routing & failover · token / cost metering · rate limiting · prompt guardrails · semantic caching · MCP & A2A routing.

Hands-on: agentgateway in the lab

One gateway for an agent's BOTH legs — https://github.com/k8ssecurity/gateway_api_tutorial



Gateway holds the key

OpenAI key in a Secret; the gateway injects it. The agent uses a dummy key — never holds the credential.

Tool-level authz (CEL)

An AgentgatewayPolicy allow-lists which MCP tools the agent may call — deny-by-default at the gateway.

Same Gateway API model

Gateway + HTTPRoute + AgentgatewayBackend, on a Rust data plane. One API — now for agents.

agentgateway: overlaps & extends the Gateway API

Standard where it can be (it's conformant), extended where agent traffic needs it

Overlaps — standard Gateway API

conformant to v1.5.0 — passes the upstream suite

- GatewayClass → controllerName agentgateway.dev/agentgateway
- Gateway → listeners & ports (your :8081 proxy)
- HTTPRoute → /mcp-mslearn and /openai paths
- Same control-plane ↔ data-plane (xDS) split
- The identical YAML you'd write for Envoy

Extends — typed CRDs for agents

its own API group: agentgateway.dev

- AgentgatewayBackend as a backendRef — MCP targets (StreamableHTTP / SSE) + AI providers (openai, anthropic…), with auth (secretRef / passthrough) & TLS
- AgentgatewayPolicy — CEL tool-level authz (mcp.tool.name), guardrails, rate limits
- Protocol awareness beyond HTTP — MCP sessions & tool listing, A2A, LLM provider routing / token metering / chat-completions rewrite

How it extends: the same Tier-2 mechanism from the Ingress slide — typed CRDs as a backendRef (group agentgateway.dev) and via Policy Attachment. Validated and explicit, not annotations — and the standard parts stay portable.

04

Security & multi-tenancy

Who may publish what, and to whom — enforced by the platform

- The role model: infra provider / cluster operator / app developer
- ReferenceGrant — cross-namespace consent, and how RBAC complements it
- Controller-based namespace selection — internal vs. external gateways

Built-in role separation = trust boundaries

Three resources, three owners -> least privilege is the default, not an add-on

Infrastructure provider	Cluster operator	Application developer
`GatewayClass`	`Gateway`	`HTTPRoute`
Defines which implementation exists. Cloud/vendor/platform team.	Owns listeners, ports, TLS, IPs, and which namespaces may attach (allowedRoutes). Platform/netsec team.	Owns host/path rules and backends within what the operator allows. App team.

Why it matters: the API splits ownership so an app team can publish routes without holding the keys to the gateway, TLS, or other teams' services. Every cross-boundary action needs explicit permission

Built-in role separation = trust boundaries

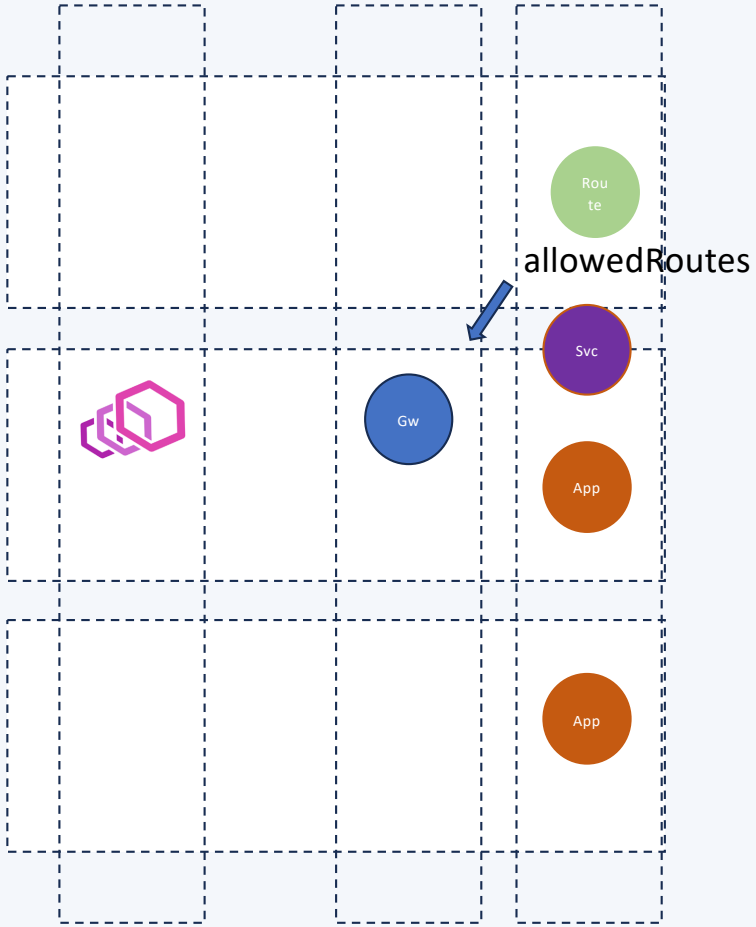
Three resources, three owners -> least privilege is the default, not an add-on

Infrastructure provider	Cluster operator	Application developer
`GatewayClass`	`Gateway`	`HTTPRoute`
Defines which implementation exists. Cloud/vendor/platform team.	Owns listeners, ports, TLS, IPs, and which namespaces may attach (allowedRoutes). Platform/netsec team.	Owns host/path rules and backends within what the operator allows. App team.

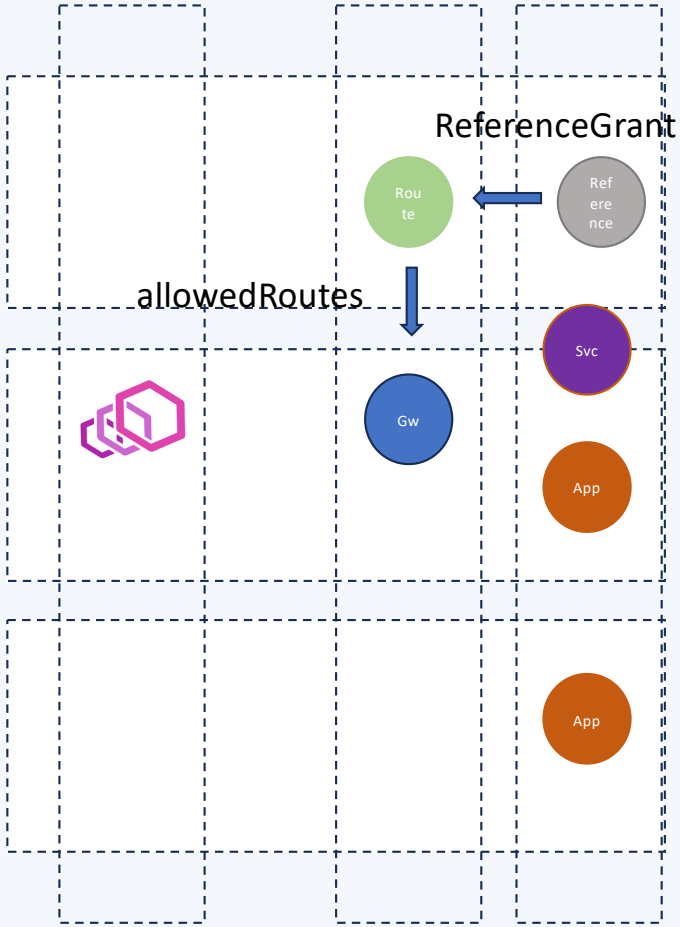
Why it matters: the API splits ownership so an app team can publish routes without holding the keys to the gateway, TLS, or other teams' services. Every cross-boundary action needs explicit permission

All in one namespace Pattern

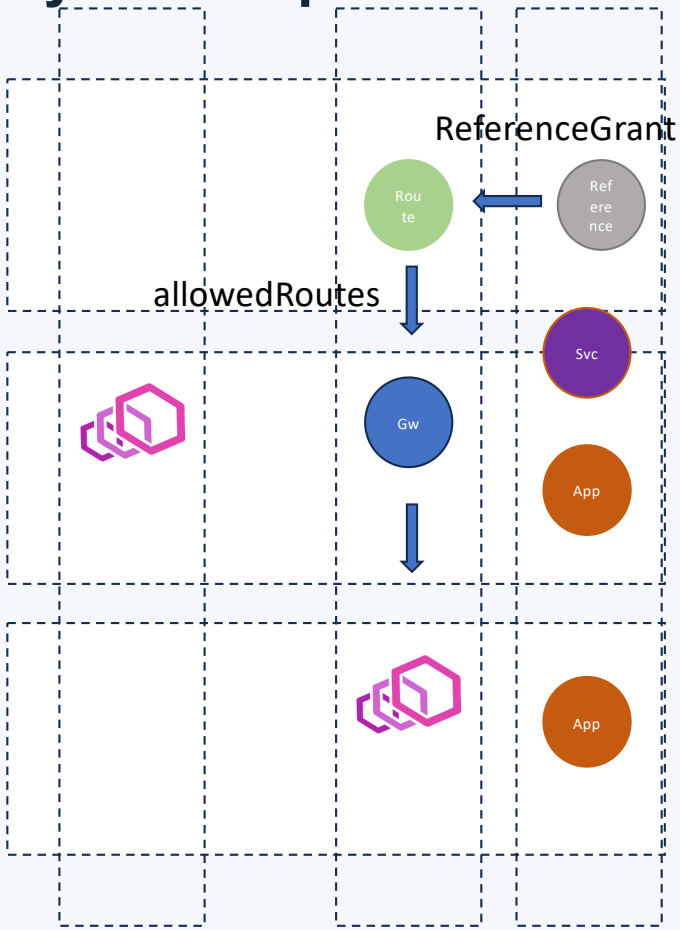
Separation of concerns Pattern



Cross-namespace



Multiple gateways / data planes

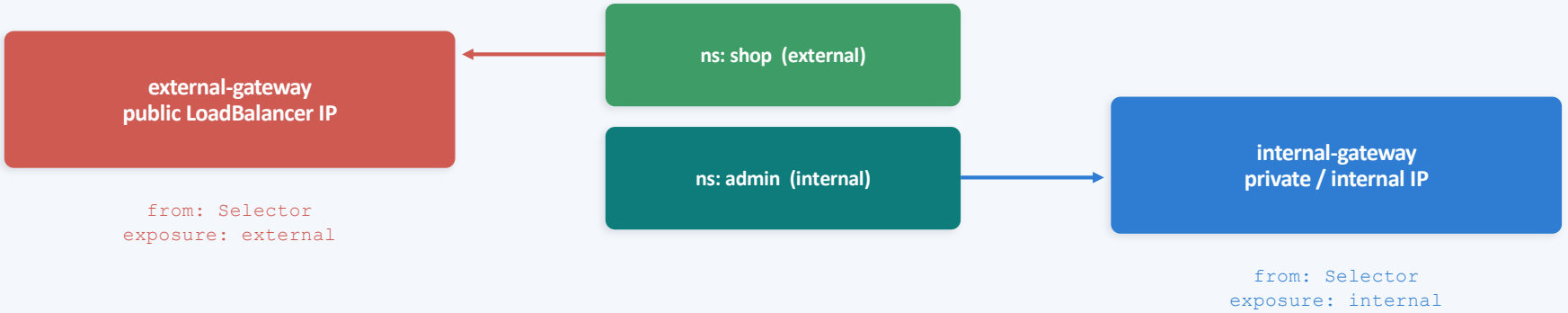


Controller-based namespace selection

allowedRoutes decides which namespaces may attach to a Gateway — set by the operator

`from: Same`	`from: All`	`from: Selector`
only the Gateway's own namespace (API default)	any namespace (convenient, permissive)	only namespaces matching a label (least privilege)

Pattern: two gateways, namespaces self-select by label



Two-way handshake: the namespace label gates attachment, but the route still names its Gateway in parentRefs. The label gates; it doesn't auto-assign. 'Internal vs external' reachability comes from each Gateway's address, not the selector.

05

TLS & network policy

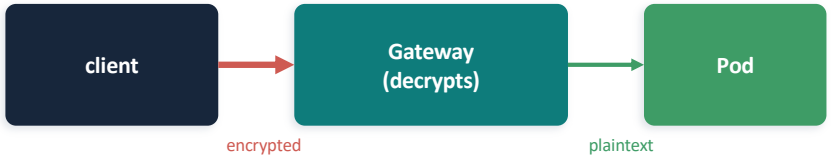
Encryption at the edge, and L3/L4 guardrails around the L7 proxy

- TLS: terminate at the gateway vs. passthrough to the backend
- Cross-namespace certificate refs also need a ReferenceGrant
- NetworkPolicy is the L3/L4 layer the Gateway API does not cover

TLS: terminate vs. passthrough

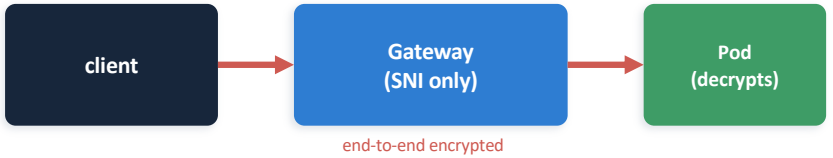
Where decryption happens decides what the gateway can see — and route on

Terminate (HTTPS listener, mode: Terminate)



+ gateway sees host/path → L7 routing, WAF, header rules
− re-encrypt for true end-to-end (backend TLS)

Passthrough (TLSRoute, mode: Passthrough)



+ end-to-end encryption; routes by SNI only
− gateway can't see/route on L7 or run a WAF

Certificates are references too

- The Gateway's tls.certificateRefs point at Secrets. A cert Secret in another namespace needs a ReferenceGrant — same consent model as backends.
- Backend/mTLS (encrypt gateway→pod) is a separate layer — via BackendTLSPolicy or a service mesh — for regulated 'encrypt everywhere' requirements.

Where network policy fits

The Gateway API governs L7 routing — NetworkPolicy is the L3/L4 layer it doesn't cover

L7 — Gateway API	host/path routing, TLS, header rules — WHO gets routed where
L4 — NetworkPolicy	which pods may talk to which, on which ports — default-deny, then allow
L3 — Pod network / CNI	IP reachability (Cilium/eBPF, Calico) — the fabric underneath

Natural fits

- Default-deny per namespace
- Only the gateway may reach backends
- Block pod-to-pod across tenants
- Restrict egress to the internet

Why both: ReferenceGrant says a route MAY target a service; NetworkPolicy says the packets are actually ALLOWED on the wire. Without policy, anything in the cluster can still reach a backend pod directly, bypassing the gateway. Cilium adds L3/L4 AND L7-aware policy (and visibility) in one eBPF dataplane.

06

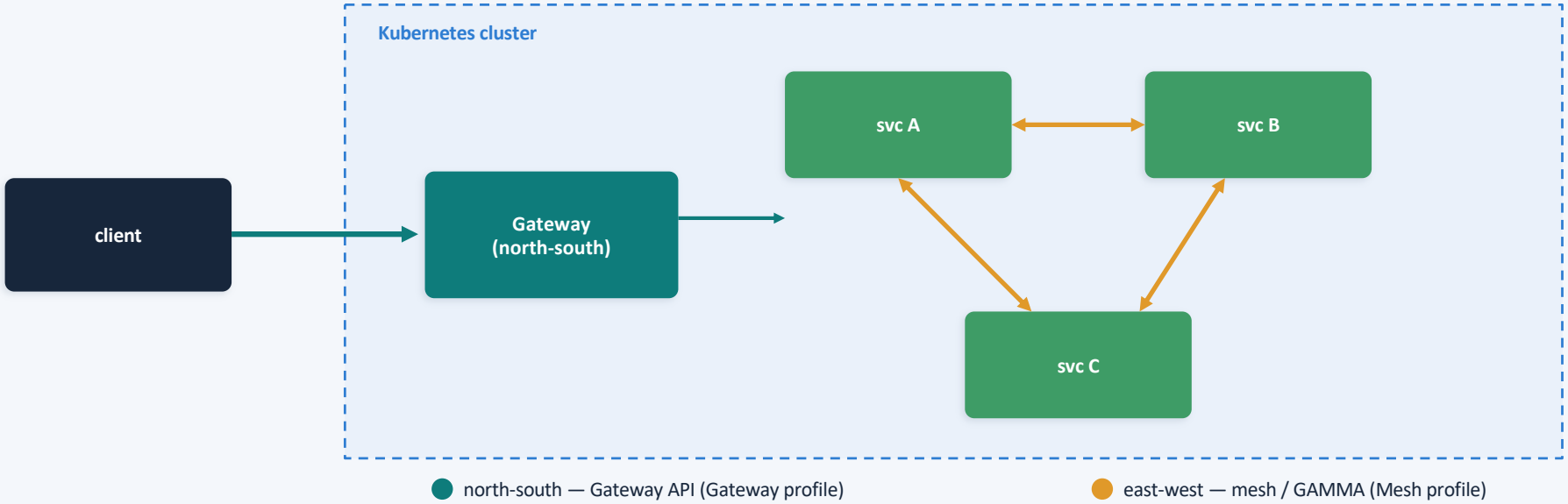
Gateway API vs. service mesh

North-south and east-west are different jobs

- Gateway API = north-south (into the cluster)
- Service mesh / GAMMA = east-west (service-to-service)
- Cilium shows L7 with eBPF — where the two worlds meet

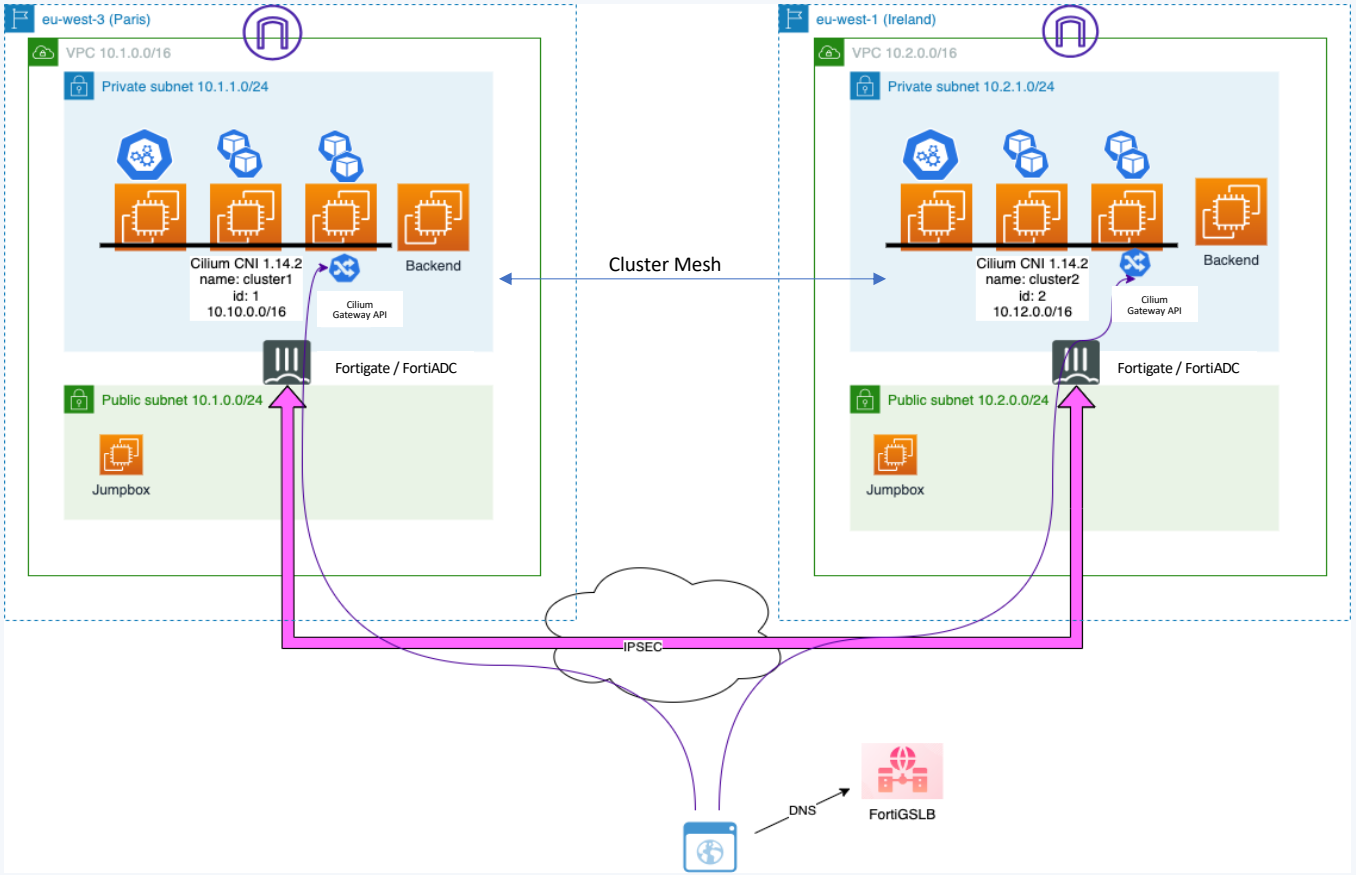
Gateway API vs. service mesh

Same API family, different traffic — one API even spans both (GAMMA)



Cilium / Istio: one API can do both. Cilium uses eBPF for L3/L4 + L7 routing and gives you L7 visibility (Hubble) without sidecars — so 'Gateway vs mesh' is increasingly 'which profile,' not 'which product.'

Gateway API vs. service mesh



Takeaways

One API, two tiers

Gateway/Route = desired state; a controller programs the proxy. The proxy moves packets, the controller never does.

The data plane may be elsewhere

In-cluster Envoy/NGINX — or a cloud LB or your physical LB. Same YAML, very different packet path. Check conformance vs. GA.

Security is built into the model

Role separation + ReferenceGrant (consent) + RBAC (who can write) + allowedRoutes (who can attach). Deny-by-default across boundaries.

L7 needs L3/L4 + TLS around it

Gateway API doesn't do reachability — pair it with NetworkPolicy and a clear terminate/passthrough TLS choice.

Ingress now spans AI & mesh

AI/MCP gateways (agentgateway) and GAMMA/Cilium L7 extend the same API to agents and east-west traffic.