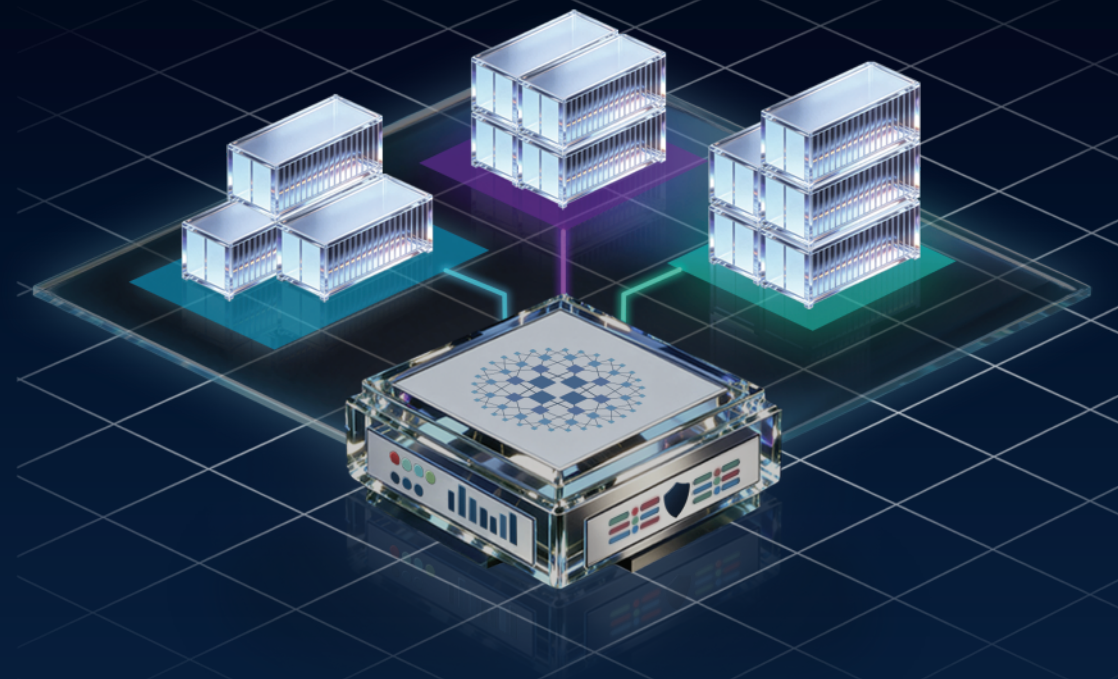
Unified Kubernetes

A single, consistent model for discovering, routing, securing, and observing traffic



HAPROXY®

Unified Kubernetes

A single, consistent model for discovering, routing, securing, and observing traffic

By Frank Mancina, Daniel Skrba, and Ron Northcutt

v4.0.0

© 2026 HAProxy Technologies

Introduction **6**
What you can expect to learn 8
Chapter 1: The Kubernetes networking challenge **11**
Flat network model 12
Before we begin 14
Challenge 1: Opaque internal routing 14
Challenge 2: External access complexity 16
Challenge 3: Configuration drift 17
Challenge 4: Multi-cluster silos 19
Challenge 5: No security at the boundary 21
Where this leads 22
Chapter 2: Rethinking Kubernetes networking **24**
Architecture and configuration are different 25
How hops translate to latency 26
A distributed two-plane architecture 28
Gateways at the boundary 30
Security belongs at the boundary 32
One pattern, any scale 34
How it works in practice 35
A simple shift in thinking 36
Chapter 3: Choosing your architectural path **38**
1. The external gateway pattern 40
2. The in-cluster proxy pattern 43
3. The hybrid pattern 46
What the cluster boundary costs 48
Cluster as a black box 49
Traffic that never leaves a cluster 50
Three ways to handle internal traffic 50
Comparing the paths 51

Flexibility matters 54
Chapter 4: The multi-cluster problem 56
How organizations end up with many clusters 56
What Kubernetes is missing 57
Failover at the pod level 58
Load balancing without boundaries 59
Blue/green deployments across clusters 60
How service discovery works 61
Controlled self-service at enterprise scale 62
What becomes possible 64
Chapter 5: Real-world applications 66
Deployment strategies 66
Blue/green at scale 67
A/B testing and header-based routing 68
Self-service without chaos 69
Security at the boundary 73
End-to-end observability 76
The architecture in action 79
Chapter 6: HAProxy One: the platform built on these principles 80
Platform overview 81
HAProxy Fusion: the control plane 81
HAProxy Enterprise: the data plane 83
HAProxy Edge 85
Kubernetes-native offerings 86
HAProxy Enterprise Kubernetes Ingress Controller 86
HAProxy Unified Gateway 87
Running the platform in Kubernetes 89
HAProxy Fusion in Kubernetes 89
HAProxy Enterprise in Kubernetes 89

Mapping the three paths to HAProxy products 89
Path 1: external gateway 90
Path 2: in-cluster proxy 91
Path 3: Hybrid 91
Reference architecture 92
Why build it this way? 95
Chapter 7: Migrating without a big bang 96
Why the coexistence model works 96
Service-by-service: the unit of migration 98
Replace, don't upgrade 100
Building confidence before committing 101
Practical groundwork 102
When you're done 104
Conclusion: What Kubernetes networking looks like at scale 105
Where the industry is heading 106
What you can do next 106
Appendix 108
Glossary of terms 109
Kubernetes terms 109
Core networking concepts 111
Architecture terms 113
Traffic and security terms 115
HAProxy One components 116
More on HAProxy and Kubernetes 119

Introduction

Kubernetes solved deployment. It didn't solve networking. The same platform that lets application teams ship ten times a day leaves infrastructure teams stitching together ingress controllers, cloud load balancers, and per-provider security controls. That patchwork is where production incidents start, and this book is about fixing it.

Kubernetes, also known as “K8s”, is now the default way organizations deploy and manage applications.

What started as an open-source project for container orchestration has become an industry standard for deploying distributed applications across enterprises of all sizes.

According to Kubernetes Market Trends, more than 60% of enterprises currently use Kubernetes, and that number is projected to surpass 90%. In another report, Kubernetes Adoption Statistics predicts the global Kubernetes market will reach $14 billion by 2034.

Organizations increasingly recognize Kubernetes as a critical component for new technologies, like artificial intelligence (AI). That's a change from its original job. Kubernetes started as a way to orchestrate containers, but it's becoming the foundation for AI infrastructure. According to the Data on Kubernetes report, organizations view Kubernetes for AI/ML workloads as a competitive advantage and essential to accelerating their AI strategy.

This means Kubernetes is here to stay.

The growing demand for Kubernetes stems from its ability to resolve the conflict between prioritizing infrastructure stability and application speed.

Team	Goal	Conflict
Infrastructure	Stability	Minimize changes to ensure high availability and performance
Application	Speed	Release updates as fast as possible to remain competitive

Infrastructure teams strive for stability and often minimize changes to ensure high availability and performance. Conversely, application teams need to move fast and release updates to production as quickly as possible. Kubernetes meets both needs. It abstracts the network, allowing infrastructure teams to keep it stable while application teams deploy at whatever speed the business requires.

Adoption brings complex networking challenges. We're no longer experimenting with containers; we're building core business functions on them.

Spectro Cloud's 2025 State of Production Kubernetes report found that 40% of organizations face three recurring problems:

- Rising operating costs as cluster utilization grows.
- Inconsistent configuration across bare metal, cloud, and edge environments.
- Security breaches that nearly every team is still trying to get ahead of.

The takeaway? Kubernetes remains hard to run well. The root cause is that organizations now need high availability at massive scale, strong security, and connectivity that holds across distributed, complex environments. Traditional Kubernetes networking models weren't built for that.

While tools like ingress controllers are built to handle the fast-moving world inside a single cluster, they often hit a wall as your organization expands. These tools operate as isolated silos, making it difficult to coordinate traffic across multiple clusters or hybrid-cloud environments. Without a way to unify these pieces, you're left with a patchwork of configurations. That means management bottlenecks, extra hops that add latency, and security policies that are hard to keep consistent.

Each cloud provider also ships its own load balancing and security controls, so there's no single place to set policy. The result is four things you can't keep consistent: policy enforcement, integration, traffic routing, and troubleshooting.

These distributed infrastructures need a unified Kubernetes layer that hides the underlying complexity and delivers consistent traffic management, security, and observability at scale.

That's the gap this book closes. Kubernetes solved deployment; the chapters ahead are about solving the networking that deployment left behind.

What you can expect to learn

This book gives you a mental model for Kubernetes networking that you can reuse on your own clusters. Our goal is to cut through the vendor claims.

Here is what each chapter covers:

- **In Chapter 1**, we will cover the five traffic management challenges inherent to Kubernetes: opaque internal routing, external access complexity, configuration drift, multi-cluster silos, and no defense

at the door. We will define the terms and concepts we'll be using throughout the book and explain why these problems are structural, not incidental.

- **In Chapter 2**, we will rethink traffic and security management from the ground up. Rather than asking which tools to add, we will ask a more fundamental question: are most teams approaching Kubernetes networking architecture in the right order?

  Two architectural decisions define your approach: where your data plane lives, and how your control plane knows what to route. Those choices determine your performance, your operational burden, and your ability to scale.

- **In Chapter 3**, we will move from philosophy to architecture. We will break down three distinct paths for Kubernetes networking: the external gateway pattern, the in-cluster proxy pattern, and the hybrid approach. Each path gets the same treatment: who it is for, what it enables, and where its limits are. You will leave this chapter with a clear picture of which path fits where your organization is today, and where it's trying to go.

- **In Chapter 4**, we will go deep on the problem that most Kubernetes teams haven't fully solved: multi-cluster networking. We'll look at why Kubernetes-native tools can't see past the cluster boundary, and what changes when a control plane sits outside the cluster and can see everything. This is also why multi-cluster federation is usually the capability that makes the architectural shift worth it.

- **In Chapter 5**, we will explore real-world applications. You'll see how the architecture from earlier chapters solves common problems: self-service routing that gives developers autonomy without giving up governance, blue/green and A/B deployments that span clusters, security enforcement at the outer edge, and observability across the whole fleet. This is also where we introduce the individual platform components. Chapter 6 puts them together.

- **In Chapter 6**, we introduce the full HAProxy One platform. We'll show what each product does in the architecture, which HAProxy combination matches each of the three paths from Chapter 3, and how the reference deployment fits together.

- **In Chapter 7**, we will address the question every team asks before committing to a new infrastructure approach: how do we get there without a disruptive cutover? You'll learn why the coexistence model (running the old path and the new one side by side) makes incremental migration the default, and how to move one service at a time with rollback available at every step. We'll also cover what to validate before you commit a service to the new path.

By the end of this book, you'll be able to evaluate your current Kubernetes networking approach, choose the right path for where your organization is today, and move toward it without betting your infrastructure on a single cutover date.

Let's get started.

Chapter 1: The Kubernetes networking challenge

To understand the challenges with Kubernetes networking and what breaks, we need to make sure we are speaking the same language.

Here are the core concepts we will be talking about throughout this book:

- **Containers:** A container is the unit that actually runs your application. It packages your application together with everything it needs to run, like its code, runtime, libraries, and dependencies, into a single standardized unit. The point is *portability*: the same container runs the same way on a developer's laptop, a test server, or a production node.

- **Pods:** A pod is a thin wrapper around one or more containers. Containers in the same pod share a network address and storage, so they behave as a *single unit*. Most pods run just one container, but tightly coupled helpers can ride along in the same pod.

- **Clusters:** A cluster is simply a group of machines, called nodes, that run your containerized applications. If a pod is a single worker, the cluster is the factory floor. As organizations scale, they rarely stick to just one cluster. It's common to run multiple clusters spread across different geographic regions or cloud providers.

- **Edge boundaries:** Communication inside your clusters is private by default, so traffic needs a way in and out. The edge is the perimeter of your network, where external traffic first reaches your internal services. We look at it in two layers: the outer edge,

the global entry point into your infrastructure, and the inner edge, the entry point into a single Kubernetes cluster.

- **Sidecars:** As networks get more complex, teams often use a "service mesh" to manage how pods talk to each other. Traditionally, this is done using a sidecar proxy: a small piece of software injected into every single pod in your cluster to intercept and route its traffic.

- **North-south traffic:** Network traffic flowing vertically into and out of your infrastructure, like a user on the internet accessing your application.

- **East-west traffic:** Internal network traffic flowing horizontally within your infrastructure, like a frontend microservice talking to a backend database.

- **Direct-to-pod routing:** A streamlined network path where an external gateway uses service discovery to send traffic directly to a pod's IP address. By skipping internal proxy layers, traffic reaches the application with lower latency.

Note: A glossary of terms is also provided in the Appendix for quick reference.

With these definitions in mind, let's look at how applications and services actually communicate, and where the major networking challenges begin to surface.

Flat network model

In Kubernetes, all pods, regardless of their node location, are part of a single, flat IP address space. This flat network lets every pod reach any

other pod using its unique IP address, wherever either pod runs. The address is the identity with no port mapping, no proxy, and no hostname translation in between.

The flat network is the result of three requirements the Kubernetes network model places on every cluster:

- **IP-per-Pod**. Every pod receives a unique, dedicated IP address within the cluster.
- **No NAT between pods**. Any pod can reach any other pod across nodes using those addresses, with *no* network address translation in the path.
- **Node-to-pod direct reachability**. Agents running on a node, such as the kubelet, can talk directly to every pod on that same node. The kubelet depends on this, so health probes, container logs, and exec sessions all reach the pod directly, without routing through the cluster network.

The no-NAT guarantee covers only pod-to-pod and node-to-pod traffic. Traffic that leaves the cluster can still be translated, and usually is. That's why a pod's own view of its address and the internet's view of that same pod don't match.

While this "flat" model simplifies internal communication, it creates a clear boundary around the network. Nothing in the model obliges a pod to be reachable from outside the cluster, and by default, it isn't. Pods are private. To let traffic in, Kubernetes relies on specific entry points.

This boundary, and the pods that keep changing behind it, is where five networking challenges emerge.

Before we begin

If you've run Kubernetes in production, you've seen these problems before, and you know they've been solved, at least in part. Service meshes, ambient mesh, eBPF-based networking, multi-cluster federation, and a long list of products all exist because of the five challenges in this chapter.

But most of those answers arrived one problem at a time, and each one added another layer to an architecture that was already difficult to reason about. Internal traffic gets a mesh, ingress gets an HTTP proxy, security gets a WAF, and multi-cluster failover gets a global DNS manager. That's four separate control planes to operate, and four more places for a failure to hide.

For the next five sections, we'll set the existing solutions aside and examine the problems on their own terms. Judging whether a fix is right is hard until you're clear about what it's fixing. We'll return to what's available and to what a more holistic answer might look like, once the problems are on the table.

If a section makes you think "there's a tool for that," good. Hold the thought, because that's exactly the point.

Challenge 1: Opaque internal routing

Every pod gets its own IP address on the flat network, so pods can talk to each other directly. The problem is that those addresses don't last. A pod can restart or be rescheduled to a different worker node at any time, and when it does, it returns with a new IP address. That's healthy behavior for a cluster, but it makes direct pod-to-pod connections unreliable. You can't build a service on an address that might change under you.

Kubernetes solves this with Service objects. A Service holds a stable virtual address and acts as a stable proxy in front of a group of pods running the same application. Behind that address, Kubernetes keeps a live list of the pods that are ready: the EndpointSlices behind the Service.

Instead of connecting to a pod, you connect to the Service. When a pod restarts and gets a new IP, the list updates and the Service routes to whichever pods are healthy. The consumer never needs to know.

For traffic moving between pods inside the cluster, this works efficiently. The kernel intercepts packets destined for a Service's virtual IP and rewrites the destination to the actual pod IP via network address translation.

The rewrite happens locally, in the node's own network stack. This is the one place NAT appears inside the cluster. Traffic sent straight to a pod IP still travels unmodified, exactly as the flat network model promises. The translation is what makes a stable address work on top of unstable ones.

That layering is cheap when everything is healthy, but it costs you visibility. A single call in a multi-tier application can pass through several Services, each one rewriting the destination in the kernel, none of it ever leaving the internal network. When latency climbs or a request fails, there's no obvious place to look.

And because the translation happens at Layer 4, you can't separate a slow application from slow DNS, a slow node-to-node hop, or a slow rule lookup. Limited observability makes it difficult to manage, maintain, and improve your application delivery.

Challenge 2: External access complexity

The same Services that stabilize internal pod communication can also expose your applications to the outside world. The simplest way to do this is a NodePort Service, which opens a consistent port on every worker node in the cluster and routes incoming traffic to the pods behind that Service.

It works, but it creates two problems.

The first is port sprawl. Each Service your teams deploy needs its own NodePort. An e-commerce application needs one for the cart service, another for inventory, and so on. You end up with dozens of Services, each holding a high-numbered port open on every host in the cluster. That's a fragile surface area for anything beyond a simple prototype.

The second is latency. A NodePort opens a port on every worker node, so a user's request can land on any of them. If the pod that handles the request isn't running on the node that received the traffic, Kubernetes forwards the request to a node that is. That extra hop costs latency, and nothing guarantees the first node a request hits is the right one.

What you want is something that knows which node is running the target pod and routes the request there directly.

In a cloud environment, you get it for free. Setting a Service's type to LoadBalancer tells the cloud controller manager to provision a real load balancer and attach a public IP to it. On a cluster using the provider's native CNI, that load balancer can target pod IPs directly, in IP mode, and skip NodePort entirely. The extra hop disappears, and the platform does the wiring for you.

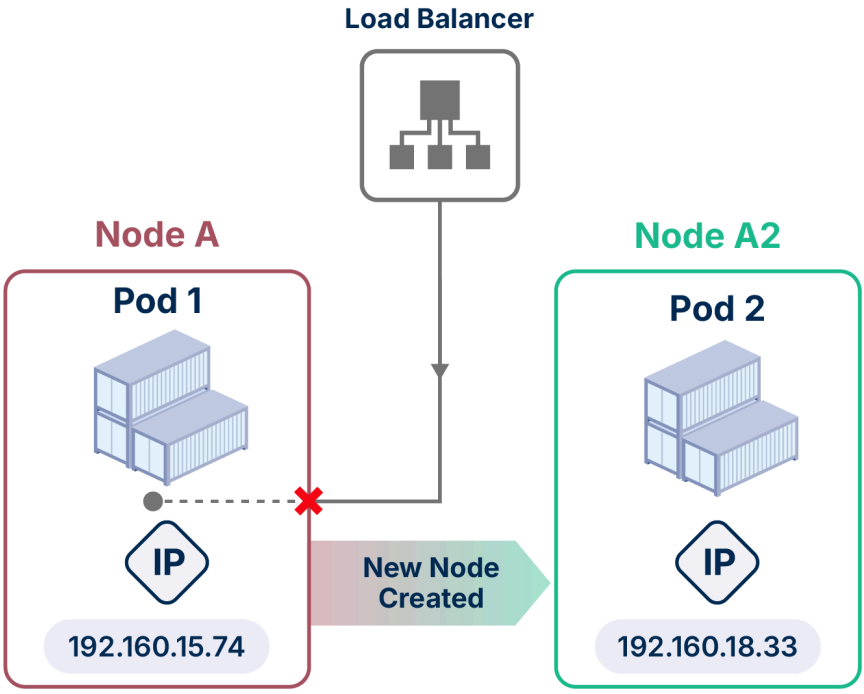
On bare metal, none of that exists. Kubernetes ships no implementation of a LoadBalancer Service outside the cloud providers. Create one on a self-hosted cluster and its external IP stays in pending forever, because nothing in the cluster can hand out an address or announce it to the network. There is no native mechanism to allocate an external IP pool, advertise addresses to physical switches, or drive a hardware load balancer.

So on-premises teams build the missing layer themselves. An IP address manager such as MetalLB or kube-vip assigns external addresses from a pool you define, then announces them to the physical network over Layer 2 with ARP or NDP, or over Layer 3 with BGP. And teams that already own load balancing appliances wire those in instead, running a controller that watches Services and reconfigures the appliance as pods move.

Each of those is a real answer. None of them is *the* answer. The choice depends on your network, your routing protocols, and what hardware you already own, and someone has to make it and maintain it by hand. That's the gap. In the cloud, external access is a Service type. On-premises, it's a project.

Challenge 3: Configuration drift

Kubernetes is built on the same networking protocols and routing tools that have powered the internet for decades. It writes routing rules behind the scenes, relays traffic between worker nodes, and keeps everything coordinated as your application scales.



But that dynamism has a cost. Pods are continuously created, deleted, and rescheduled as a cluster responds to demand. Each time a pod is destroyed and re-created elsewhere, it gets a new IP address. If your external load balancer isn't tracking those changes in real time, its routing table goes stale, still pointing at addresses that no longer exist.

Stale records don't fail quietly. Traffic sent to a deleted pod's address is blackholed. Requests that arrive before the new pod is ready return 502s and 503s. And when an address is recycled to a different service, traffic intended for one application ends up on another. That's an availability problem and a security one.

Inside the cluster, none of this happens. Kubernetes handles pod churn natively: its controllers monitor the API and update the EndpointSlices

associated with every Service within milliseconds. The gap opens at the boundary. External systems were built for static infrastructure, where addresses rarely change.

That's where "configuration drift" changes meaning. In traditional server environments, drift is slow, the product of manual changes made over months. In Kubernetes, it's automatic and continuous. Any external component that isn't watching the Kubernetes API falls out of sync within seconds, and never catches up on its own.

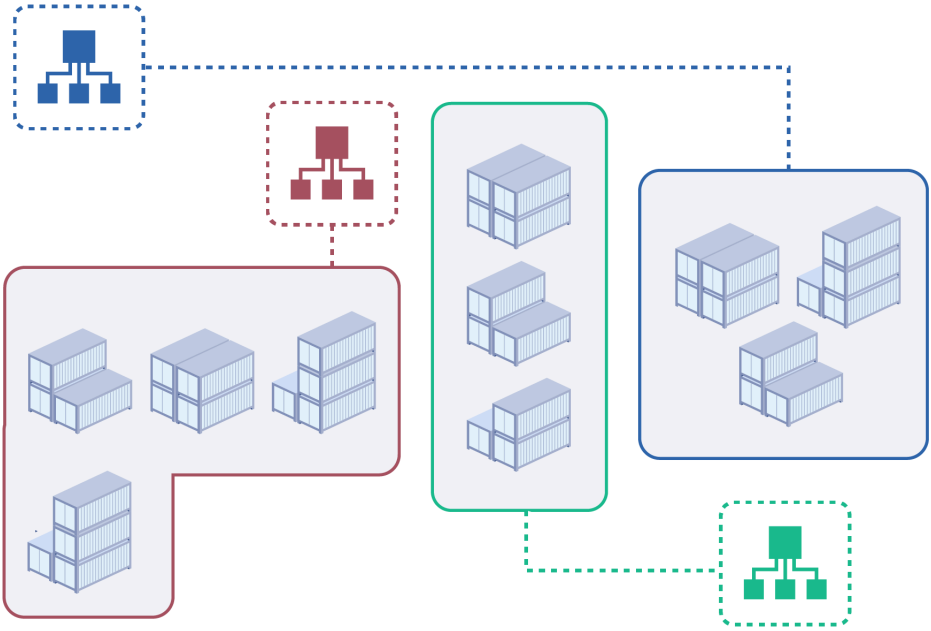
Manual updates can't solve that. Pods move faster than any human process can track.

What closes the gap is a controller: software that watches the Kubernetes API and reconfigures the external system the moment something changes. An ingress controller subscribes to EndpointSlices and updates its proxy targets in memory, without dropping connections. ExternalDNS tracks Services and Ingresses and rewrites DNS records as addresses change. A cloud provider's load balancer controller does the same for target groups.

The pattern is the same in every case: watch the API, react to events, never poll. Without a networking layer that stays synchronized with Kubernetes, stale configuration isn't a risk you manage. It's the steady state.

Challenge 4: Multi-cluster silos

Your infrastructure grows with your organization. Many teams end up running multiple Kubernetes clusters across different regions and hosting environments. Yet Kubernetes doesn't provide a built-in way to route traffic between them. Each cluster becomes an isolated silo.



That isolation is structural, not a gap someone forgot to fill. Every cluster has its own control plane, its own DNS, and its own IP space. A ClusterIP means nothing outside the cluster that assigned it. A service name resolves only inside its own .cluster.local domain. Nothing in one cluster can reach anything in another using the names and addresses it already knows — not the Service name, not the ClusterIP, not the DNS entry.

So multi-cluster routing depends on components *outside* Kubernetes. Ingress and Gateway resources only direct traffic within a single cluster. Global routing needs a load balancer, anycast addressing, or an external DNS controller.

Cross-cluster calls are worse still. When a service in one cluster calls a service in another, the traffic leaves the cluster through an egress gateway. It crosses the public internet or a VPN, then comes back in

through an ingress gateway. That hairpin costs latency and money. It also adds a security surface nobody asked for.

All of that makes multi-region failover, blue/green deployments, and A/B testing nearly impossible to run. Failure is inevitable in any distributed system. But high availability is the discipline of designing systems that stay up when individual components fail. That's hard when every cluster is an island.

Challenge 5: No security at the boundary

The flat network model we've described is good at one thing: moving traffic to a healthy pod as fast as possible. It doesn't care what's inside. An SQL injection attempt, a cross-site scripting payload, a bot stuffing stolen credentials into your login form, a flood of requests designed to exhaust your resources — to Kubernetes, these are all just valid HTTP requests bound for a running application.

The cluster routes them as faithfully as it routes a legitimate customer.

This is by design. Kubernetes networking works at Layer 3 and Layer 4. It moves packets by IP address and port, and it never opens them.

The security primitives Kubernetes offers work at that same layer. NetworkPolicy determines which pods are allowed to communicate with which other pods. It can prevent one pod from opening a connection to another, but it can't see what travels over it once that connection is open. RBAC decides who can touch the cluster's API.

Both control who is allowed to connect. Neither one ever looks at what's inside the connection. There is no native firewall for application traffic, no

rate limiting, and no bot detection. Kubernetes secures the cluster; it does nothing to secure the application from the people trying to abuse it.

So teams add that protection themselves, and this is where the flat network model works against you. With no inspection at the boundary, malicious traffic enters the cluster freely and gets routed deep into your services before anything questions it.

Security ends up scattered. It's bolted onto individual services one at a time, applied inconsistently, and always reacts after the traffic is already inside. Each team reinvents the same defenses, slightly differently, and the gaps between those reinventions are exactly where attackers get in.

By the time a request reaches a pod, you've already given up your resources to admit, route, and process it. What should have been turned away ends up as an expensive lesson.

Where this leads

Step back and look at the five challenges: opaque internal routing, external access complexity, configuration drift, multi-cluster silos, and no defense at the door. The path forward looks daunting.

Many engineering teams react by taking a piecemeal approach. They layer point solutions on top of one another to plug individual holes as they appear. They deploy one tool for external load balancing, a separate firewall for security, a heavy sidecar mesh for internal pod communication, and yet another piece of software for multi-cluster failover.

But this fragmented approach carries an operational cost. It creates a fragile, bloated architecture that is difficult to observe, expensive to scale,

and hard to troubleshoot. You end up managing a complex patchwork of overlapping tools instead of focusing on application performance.

Observability, external access, service discovery, and multi-cluster routing. It's a lot to balance. Throw security into the mix and a high-performance infrastructure starts to look impossible.

But it isn't.

To build a flexible, secure, and scalable Kubernetes network, you don't need more tools. You need a **fundamentally different way of thinking about network architecture**. Instead of adding another tool for each gap, you need to make two deliberate decisions that most teams leave to chance. Everything else flows from there.

In the next chapter, we will explore what those decisions are, why they matter more than any individual tool choice, and how answering them well changes the architecture entirely.

Chapter 2: Rethinking Kubernetes networking

Two decisions shape every Kubernetes network: where your data plane lives **(architecture)**, and how your control plane decides where traffic goes **(configuration)**. Get them right and the rest of the stack gets simpler. Get them wrong and you spend the next year adding tools to work around the first choice.

In the last chapter, we identified five problems every serious Kubernetes deployment runs into: opaque internal routing, external access complexity, configuration drift, multi-cluster silos, and no security at the boundary. The result is usually the same: a patchwork of tools.

Most teams never stop to ask where their data plane lives and how their control plane routes traffic. That's the decision that shapes everything below it.

Before you pick a load balancer, a service mesh, an ingress controller, or a security layer, there's a more fundamental decision to make. That decision sets your network's performance ceiling and your team's operational burden more than any product choice does. Most organizations make it by accident, as a byproduct of adopting Kubernetes-native defaults. The ones that make it deliberately end up with faster, simpler, and more scalable infrastructure.

Those are actually two separate questions. Treating them as one is the source of most of the complexity you end up with. So make the call on purpose. The rest of this chapter shows what changes when you do, and what it costs when you don't.

Architecture and configuration are different

Architecture and *configuration* are not the same thing, even though many teams treat them as if they were.

Architecture is the physical reality of your network. It's where your data plane components run, which network paths traffic actually takes, and how many processing layers a request passes through before it reaches an application pod. Architecture determines *latency*. It decides how many components can fail before your application goes down. And it sets whether you can load balance across clusters or only within one.

Configuration is how you tell the system what to do. It's the language you use to express routing rules: whether that's a Kubernetes `Ingress` resource, a Gateway API `HTTPRoute`, a custom resource definition (CRD), a custom annotation, or a configuration file on a proxy. Configuration adapts to your workflow. Different teams can use different configuration models on top of the same architecture, and the architecture doesn't change.

“Your configuration workflow is flexible. Your architectural decisions have consequences.”

Teams routinely conflate the two. The result is conversations like: *"Can we reduce latency by switching from Ingress to Gateway API?"* The honest answer is: not by itself, no.

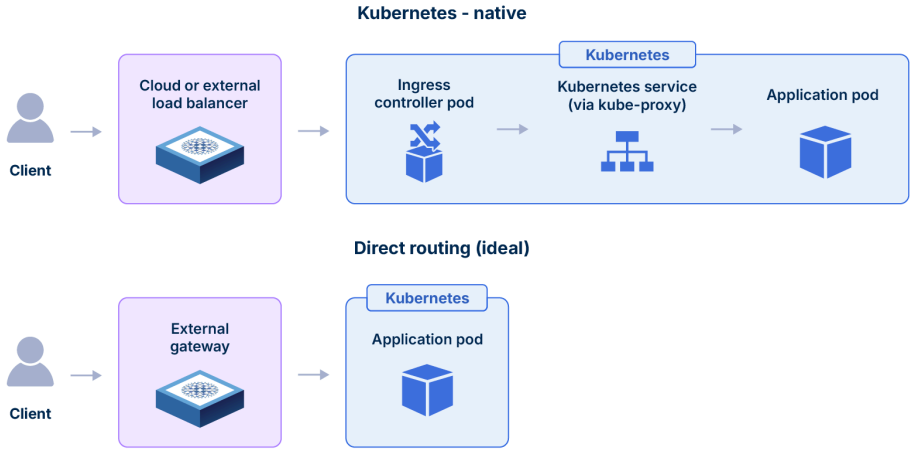
Switching from Ingress to Gateway API is a *configuration change*. It gives you richer routing semantics and cleaner role separation. But if your data plane still sits inside the cluster, receiving traffic through the same number

of hops, the latency profile doesn't change. The traffic still travels the same physical path.

Architecture determines latency through two things: how many processing layers traffic passes through, and where those layers sit relative to your application pods.

How hops translate to latency

Every layer that traffic passes through on its way to an application pod adds latency. More hops mean more time. The goal is always the fewest hops possible between the client and the pod.



In a standard in-cluster proxy setup, a user request follows a path like this:

Client → Cloud or external load balancer → Ingress controller pod → Service Routing (kernel NAT/kube-proxy) → Application pod

That's at least three processing layers before traffic reaches your application. And that's before you count the routers between those components, which compound the problem.

Each layer involves packet handling, routing table lookups, and context switching. Each is also a potential point of failure. The ingress controller and `kube-proxy` NAT layers exist because Kubernetes's internal networking model requires them. They're not doing anything your application specifically needs. They simply route packets.

Now consider an alternative path:

Client → External gateway → Application pod

With this path, there is only one processing layer before traffic reaches the application. Through service discovery, the external gateway knows exactly where the pod lives, routes directly to its IP address, and bypasses the cluster's internal routing layers entirely. The application gets the request faster, with fewer components in the path.

This second path is a *different architecture*. Getting from the first to the second requires changing where the data plane lives and how it learns about pod locations, not which YAML format you use to express routing rules.

This is also why you have to answer the architecture question before the configuration question. Once you've committed to running your data plane inside the cluster, direct-to-pod routing isn't available to you, regardless of how you configure routing rules.

But if you've committed to an **external data plane with service discovery**, you can use almost any configuration model you like: Ingress

rules, CRDs, Gateway API objects, custom annotations, or native proxy configuration. The latency benefit holds either way.

So pick your architecture first. Then pick whatever configuration model your team prefers.

A distributed two-plane architecture

Most of the complexity in Kubernetes networking comes from tools that mix two different jobs: deciding how to route traffic and actually moving it.

An ingress controller, for example, is simultaneously watching the Kubernetes API for routing updates, processing real traffic, enforcing TLS, and making load balancing decisions. It's doing the work of both the brain and the muscle, inside the cluster where it has to compete for resources with your applications.

A better approach explicitly separates these concerns.

The **data plane** is where your actual traffic passes through. Its job is to be fast and reliable. It handles layer 4 and layer 7 routing, executes security rules, processes TLS, and makes load balancing decisions at the individual request level. Speed matters here: every millisecond of added latency in the data plane is directly felt by your users. Put the data plane at the *boundary* of your environment, not buried inside the cluster.

The **control plane** is what tells the data plane what to do. Its job is to decide where traffic goes. It watches Kubernetes APIs, tracks pod lifecycles, aggregates telemetry, manages policies, and pushes configuration updates to data plane instances across every cluster in your environment. The control plane *should never be in the path of live traffic*. Its job is to manage complexity, not to process requests.

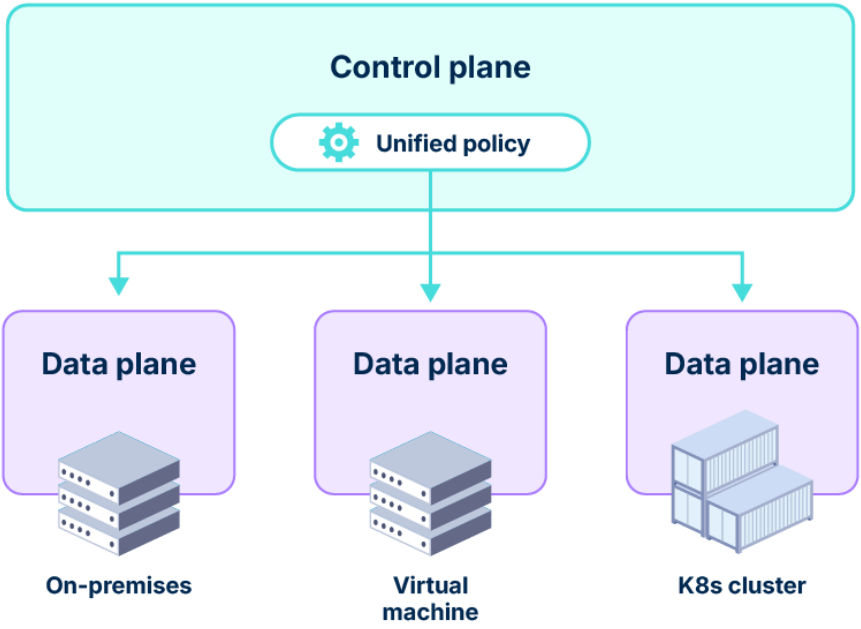
Keeping these two things separate is what allows each to do its job well. The data plane stays lean and fast because it's not also responsible for service discovery. The control plane stays intelligent and responsive at scale because it's not also responsible for handling traffic.

And critically, the control plane can also sit entirely outside your clusters and watch multiple Kubernetes APIs at once, across multiple clouds. That gives you a global picture no single cluster-scoped tool can match.

Terminology: planes and layers

Inside a Kubernetes cluster, "control plane" and "data plane" already mean something specific: the API server versus the `kubelet` and `kube-proxy`. To avoid confusion, we use two sets of terms for the traffic architecture:

- **The abstract planes (the philosophy):** The control plane and data plane represent a structural separation of concerns, keeping *routing intelligence* completely split from physical *traffic handling*.
- **The functional layers (the execution)**: To ground these abstractions, we name two layers. The *orchestration layer* is the control plane managing discovery, policies, and configuration. The *gateway layer* is the data plane gateways executing routing and security at the boundaries.



This separation is also what makes multi-cluster routing possible. An ingress controller that runs inside a cluster knows only what its cluster's Kubernetes API tells it. A control plane that runs outside can watch ten clusters at once and direct traffic across their boundaries. That's what makes multi-cluster load balancing and failover work across clusters. We'll cover that in depth in Chapter 4.

Gateways at the boundary

Once you've committed to an external data plane, the next question is: how do you handle internal service-to-service traffic? The traditional answer has been a service mesh with sidecar proxies: a small proxy injected into every pod to intercept and route internal traffic.

This approach works, but it carries a real cost. Running a proxy in every pod consumes CPU and memory that would otherwise go to your applications. It adds latency to every internal request. And when something goes wrong, the resulting tangle of per-pod routing rules is genuinely difficult to debug.

These costs scale with your pod count. The more you grow, the more you pay in resource consumption, in operational complexity, and in troubleshooting time. This is sometimes called the "**mesh tax**."

The alternative is to route internal traffic through the same boundary gateways that handle external traffic. A *gateway* is an infrastructure-agnostic, location-based term for a data plane component positioned at a network boundary. Rather than referring to a specific technology, it defines a role in the traffic flow — acting as the single checkpoint through which traffic enters, exits, or transitions between network zones.

When a service in Cluster A needs to reach a service in Cluster B, the traffic goes to the local boundary gateway. That gateway already knows where the destination service lives, because the control plane told it. It routes the traffic securely to its counterpart at the other cluster's boundary, which delivers it to the destination pod.

No sidecars required.

The same gateway infrastructure that handles your external north-south traffic also handles your internal east-west traffic. Your resource footprint shrinks, and your routing model becomes explicit rather than invisible. Troubleshooting reduces to a manageable set of gateway logs rather than a distributed web of per-pod traces.

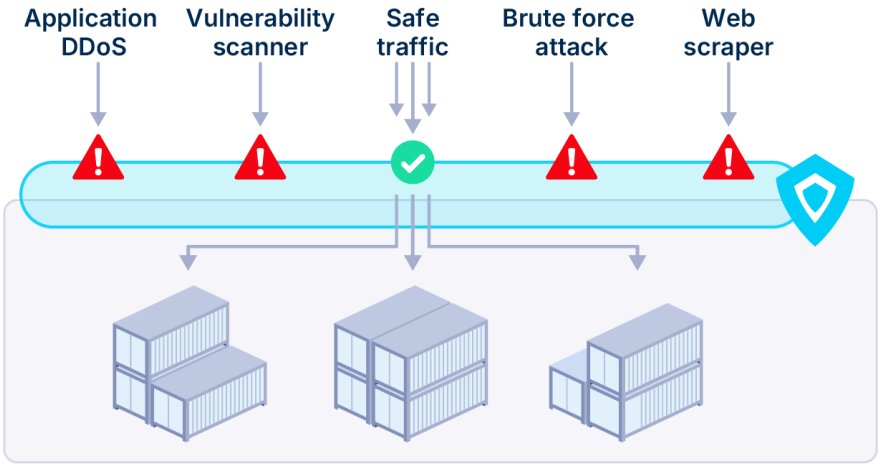
Once this gateway infrastructure handles both your external and internal traffic paths, protecting your applications gets simpler. These gateways are already the designated checkpoints for every request, so they're the natural place to enforce security. When you control the perimeter, security belongs at the boundary.

Security belongs at the boundary

Where you enforce security is an architectural decision with significant performance implications.

If your primary security enforcement happens deep inside the cluster, at the pod level, malicious traffic consumes cluster resources before anything rejects it. By then it has already crossed your ingress controller and `kube-proxy`. Every DDoS probe, every bot request, every malformed API call that makes it to your internal network has already cost you something.

The more efficient approach is to enforce your strongest security measures at the outermost boundary of your network, before traffic enters the cluster. WAF rules, bot detection, DDoS mitigation, and TLS authentication all belong there, where they can reject threats before they touch anything internal.



This **strongest armor at the border** principle doesn't replace internal security. Network policies, role-based access control, and pod-to-pod authentication all still matter. But if your perimeter security is doing its job, far fewer threats reach the point where internal security needs to act. Your application pods spend their resources handling legitimate traffic instead of processing attacks that your perimeter should have stopped at the door.

The practical implication is that your data plane gateway needs to be a full-featured security enforcement point, not just a traffic router. That's why "just use the cloud provider's load balancer" is an incomplete answer. Most cloud load balancers are designed to route traffic, not to inspect it.

Why not simply tack on the cloud provider's standalone security layers? Because they bring back the latency tax of extra processing hops. Forcing traffic through an assembly line of disjointed security services introduces the multi-hop latency we are trying to eliminate.

Instead of building a streamlined path, you end up constructing a multi-tiered pipeline that degrades performance before packets ever

reach a cluster node. Keep enforcement at the boundary, and the cluster only ever sees traffic worth processing.

One pattern, any scale

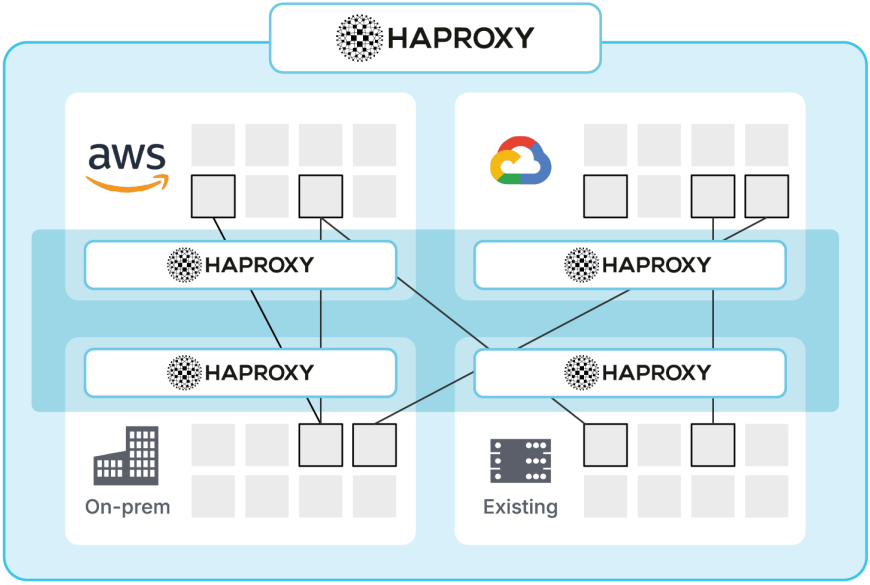
A Kubernetes networking architecture that scales is built around a single pattern, applied consistently at every tier. Repeat that pattern and you can add clusters without changing the underlying model. This ensures you only ever have to manage one system.

That single pattern is a data plane of boundary gateways, managed by a centralized control plane.

At the cluster level, this pattern provides a gateway at the entrance of each cluster (your inner edge) that handles traffic routing into and between services within that cluster. Zoom out and the same pattern gives you a gateway at the regional or data center level, your outer edge. It handles global load balancing, multi-cluster routing, and initial threat scrubbing before traffic reaches any cluster. Zoom out further, and the same pattern extends to a globally distributed edge network, covering traffic from everywhere on the internet.

This fractal consistency matters operationally. Your inner and outer edge gateways are the same component, managed by the same control plane, with the same configuration model. Your team only has to learn one system. Security policies applied globally propagate consistently to every layer. Telemetry from every level aggregates into a single view. Adding a new cluster is just applying the same pattern.

We call this architectural pattern the **Universal Mesh.** It's a way of thinking about traffic management that scales from a single cluster to a globally distributed infrastructure without changing the underlying model.



How it works in practice

The Universal Mesh pattern deploys in layers.

The control plane runs as a *sovereign control plane* that you deploy and manage, rather than a vendor-managed SaaS platform. It connects to the Kubernetes API across multiple clusters simultaneously, watching for changes such as pod creation, deletion, IP address changes, and service updates. This keeps the data plane synchronized with the actual state of your infrastructure.

The data plane gateways run at the boundaries of your environments. They receive configuration from the control plane, handle live traffic, enforce security rules, and route requests to their destinations. Because they sit outside the cluster and receive pod IP data from the control plane,

they can route directly to pods without going through the cluster's internal routing layers.

The result is a network that knows what's running at any given moment. It routes traffic along the shortest available path, applies security at the entry point, and gives you one view across every cluster you run.

To understand why this layout holds up, look back at the five challenges we highlighted in Chapter 1. Instead of just treating the symptoms by piling on more tools, this design fixes the core issues at the network level:

- **Clear internal routing.** The gateway routes traffic straight to the pod's IP address instead of hiding it behind internal network layers. When latency spikes, you have one obvious place to look.
- **Simpler external access.** One component handles Layer 4 and Layer 7 traffic at the boundary, so you stop stitching together cloud services, external IPs, and high-numbered NodePorts.
- **No more configuration drift.** The control plane watches the Kubernetes API in real time, so it catches pods being created or deleted and updates the gateways before traffic reaches a stale IP.
- **Unified multi-cluster environments.** The control plane builds a global map across every environment, and failover happens at the pod level instead of through slow DNS changes.
- **Active defense at the front door.** Malicious traffic and bots are blocked at the edge, keeping bad traffic out of the cluster and preserving your compute resources for legitimate users.

Those five fixes are what the rest of this chapter builds on.

A simple shift in thinking

This approach brings us back to the most critical takeaway of this chapter: **architecture and configuration are two entirely different things**.

Many engineering teams get caught in a cycle of adding new tools because they try to fix structural and physical problems with software settings. For example, switching from a basic Ingress file to a Gateway API resource gives you better configuration options. It won't reduce your latency, though, if your traffic still jumps through multiple proxy layers inside the cluster to find a pod.

The Universal Mesh changes the default assumption of *where* networking intelligence belongs. By physically separating the brain (the control plane) from the muscle (the data plane), you build a simpler network where complexity stops being a bottleneck as you grow.

But as we'll explore in the next chapter, the path to get there isn't the same for every organization. Some teams are ready to externalize everything. Others have deep investment in Kubernetes-native tooling and need a migration path. The Universal Mesh accommodates both.

In Chapter 3, we'll break down the three architectural paths in detail, and help you identify which one fits where your organization is today and where it's trying to go.

Chapter 3: Choosing your architectural path

In the previous chapter, we showed what the Universal Mesh looks like in practice. A centralized control plane coordinates a layer of boundary gateways that handle all traffic (north-south and east-west) without sidecar proxies or in-cluster routing bottlenecks.

This boundary gateway architecture is the most efficient. But every engineering team has its own workflow and organizational structure. Some teams are deeply invested in Kubernetes-native tooling and want their routing to live inside the cluster. Others are migrating legacy infrastructure and need a bridge.

The real choice you have to make is exactly how you will manage the inner edge. That choice shapes your visibility, your security posture, and whether you'll eventually need to pay the "mesh tax."

This chapter covers three primary patterns, starting with the one the Universal Mesh is built on.

1. **The external gateway pattern** *(direct-to-pod routing)*
2. **The in-cluster proxy pattern** *(Gateway API and Ingress)*
3. **The hybrid pattern** *(the migration bridge)*

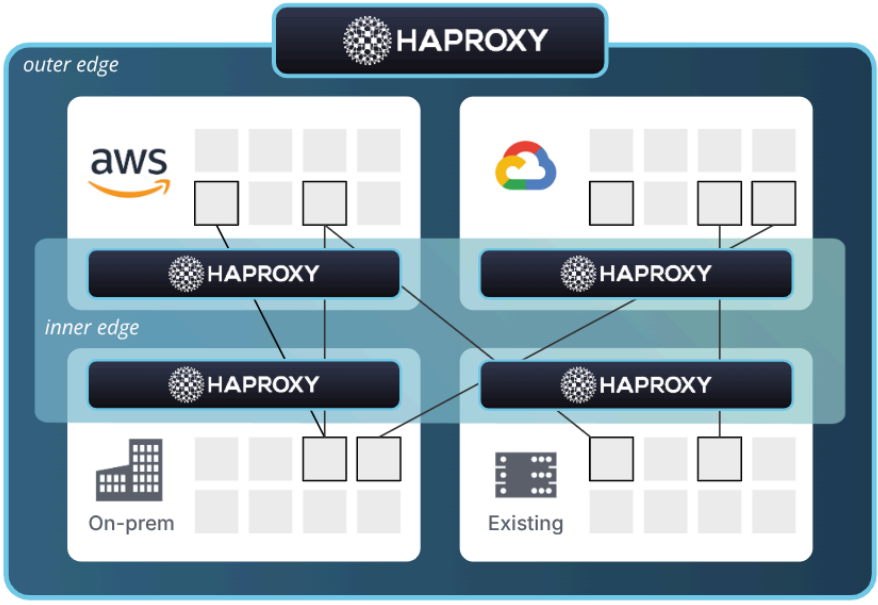
How we name the layers

To prevent confusion with Kubernetes' native internal components (like the API server or `kube-proxy`), we keep our functional terminology consistent with Chapter 2:

- **The orchestration layer (control plane):** the component that manages configuration, policies, and service discovery across your gateways.
- **The gateway layer (data plane):** the gateways (or proxies) that handle L4/L7 application traffic and security enforcement.

In the patterns ahead, the orchestration and gateway layers sit alongside your clusters rather than inside them.

1. The external gateway pattern



When you need to eliminate the "mesh tax" entirely and gain full visibility and control of your traffic, the external gateway pattern is designed for that. This architecture is the Universal Mesh at its simplest: the centralized orchestration layer manages configuration across your entire gateway fleet.

The Universal Mesh uses a flattened design. You deploy one layer of gateways that unify traffic across your entire infrastructure, whether it's multi-cloud, multi-region, or on-premises. Because the gateway-to-gateway routing for internal services is functionally the same as routing external traffic, you create a single, cohesive mesh that handles both the inner and outer edge.

Routing, load balancing, security, and rate limiting work the same way at every gateway in the mesh. That uniformity gives you a single-layer architecture, and because each gateway handles traffic identically, you can load balance across servers, VMs, containers, and pods at the same time.

One prerequisite matters here: the external gateway sends traffic directly to individual pod IPs. Your internal network needs a flat topology so those pod IPs are reachable from the gateway, which sits just outside the Kubernetes cluster.

This is a deliberate design choice. Direct-to-pod routing does three things the in-cluster proxy path can't. Where `kube-proxy` uses basic round-robin to distribute traffic, direct routing enables weighted load balancing decisions based on real backend state. The chain preserves the source IP, which matters for logging, rate limiting, and any policy that depends on knowing where a request originated. And with fewer hops in the path, you get lower latency and a simpler troubleshooting surface.

- **Conceptual boundaries:** The orchestration layer and gateways manage the inner edge and the outer edge. Together they form the Universal Mesh.

- **Data path:** Data takes the shortest route: *Internet* → *Gateway* → *Pod*. Because the gateway knows exactly where the pods are, it bypasses the internal ingress controller and `kube-proxy`. Your traffic takes one hop.

- **Kubernetes API interaction:** The routing intelligence lives entirely *outside* the cluster. The orchestration layer connects to the Kubernetes API from the outside. It watches for changes through an informer rather than polling, and batches what it finds on a coalescing interval. It discovers services, objects, and custom

resources, aggregates scaling events, and pushes an updated configuration to all your gateways. The gateway routes any traffic that needs to reach a pod directly to it.

- **Configuration management:** One configuration governs the whole fleet, with the orchestration layer as the single source of truth. Because it discovers services, objects, and CRDs from the Kubernetes API, teams that prefer YAML-driven workflows can keep them, expressing routes through the same manifests they already use. Teams that want API-driven automation get that too, from the same control plane.

- **Observability:** You get end-to-end visibility across all your Kubernetes pods, legacy virtual machines, and bare-metal servers, down to individual requests.

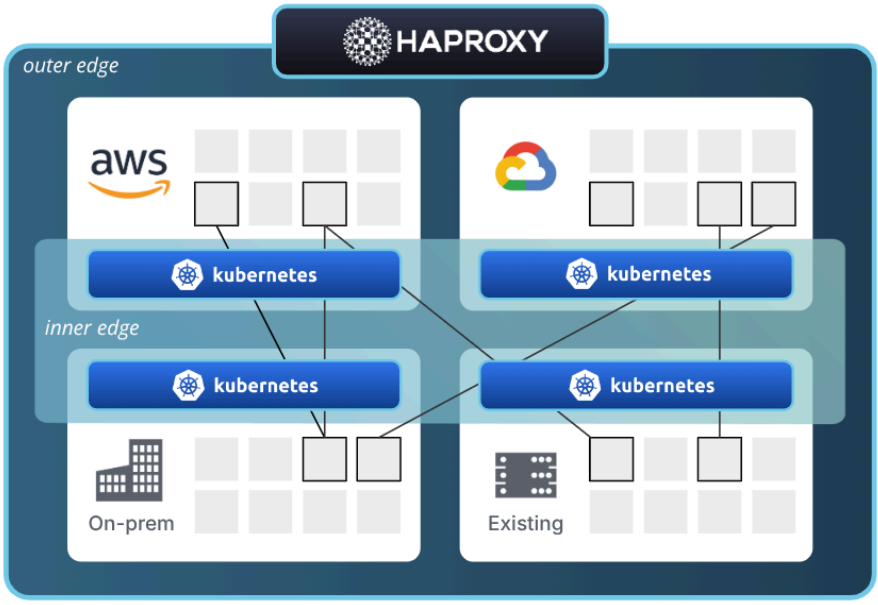
- **When to use it:** Choose this architecture when you operate at scale. It eliminates the "traffic sandwich", where traffic hops through multiple redundant networking tiers. It also removes sidecar meshes for east-west traffic and load balances across legacy servers and Kubernetes pods at the same time.

- **Considerations:** This pattern shifts routing authority from inside the cluster to a global control plane, which changes where ownership lives. It does not mean giving up Kubernetes-native workflows. Because the orchestration layer discovers objects and CRDs from the Kubernetes API, developers can continue expressing routes through YAML manifests in their existing CI/CD pipelines.

  For teams that prefer it, the orchestration layer's API supports

direct integration into automation scripts and internal tooling. Role-Based Access Control (RBAC) and conflict detection keep self-service safe in either mode.

2. The in-cluster proxy pattern



In this model, your orchestration layer sits at the global edge and manages security and routing policies. The external gateway directs traffic down to an ingress controller: a specialized proxy deployed directly on your worker nodes.

This pattern defines the Universal Mesh boundaries. The outer edge is where your orchestration layer and the external gateway reside, handling global routing and initial security scrubbing. The ingress controller manages the inner edge and routes traffic into the cluster.

Kubernetes networking has matured, and the Gateway API is where the maintainers are steering the ecosystem. It introduces a proper separation of concerns: infrastructure teams control physical listeners and TLS, while developers manage their own routing rules independently. It supports advanced routing like header matching, traffic splitting, and gRPC natively, without vendor-specific annotation workarounds.

In practice, though, adoption has been slower than that intent suggests. Migrating existing Ingress configurations is real work, and many teams haven't found a compelling enough reason to take it on. So Ingress endures as a viable and widely used choice.

Easier migration path

You don't have to choose between staying on Ingress and absorbing a full Gateway API migration. There's a way to smoothly transition to Gateway API *without* paying the full migration cost up front, which we'll return to later in the book.

The Gateway API is an improvement. But it's a configuration standard, not an architecture. It dictates how you write routing rules. It doesn't change where those rules are enforced or what traffic the orchestration layer can see.

- **Conceptual boundaries:** The orchestration layer and gateway manage the outer edge. Kubernetes manages the inner edge.

- **Data path:** Traffic travels from *Internet → External Gateway → Ingress Controller Pod → Application Pod*. The ingress controller handles all L7 routing decisions inside the cluster, including host and path matching, service resolution, and pod selection.

- **Kubernetes API interaction:** The routing intelligence lives inside the cluster. The ingress controller runs a continuous control loop, watching the local Kubernetes API to translate standard YAML manifests into routing logic.

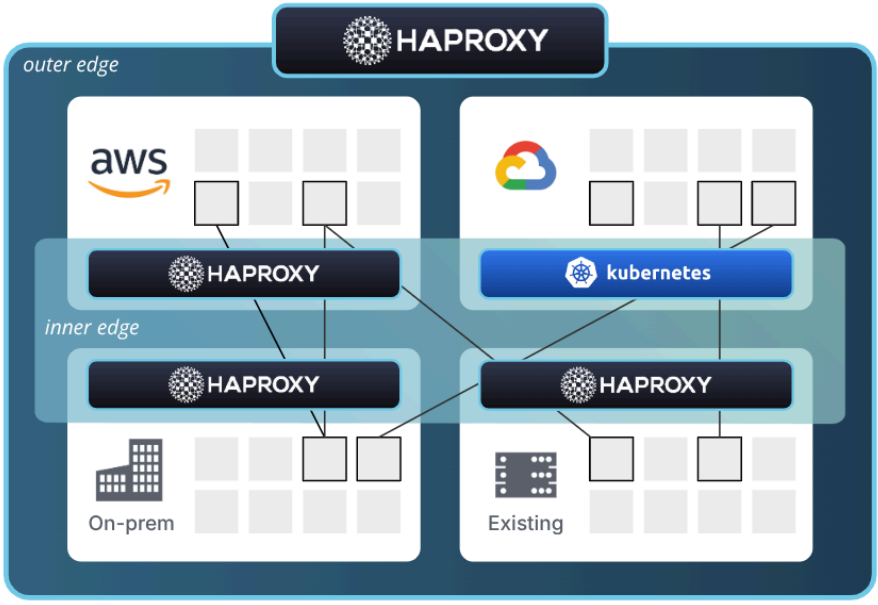
- **Configuration management:** Configuration is per-cluster and ownership is split. Kubernetes YAML governs the routing, allowing developers to manage application traffic via their CI/CD pipelines. The orchestration layer governs security, so your SecOps team pushes WAF rules directly to the global edge.

- **Observability:** Because the orchestration layer manages the outer edge, you gain visibility for traffic entering your network. Kubernetes manages the inner edge, so you'll need a separate solution for logging and metrics inside the cluster.

- **When to use it:** This approach works well when you want routing logic to live inside the cluster alongside the applications it serves. It also fits when existing ingress deployments are too deeply embedded to move right now. Application teams manage routing through familiar Kubernetes primitives while the security team still scrubs malicious traffic at the global edge.

- **Considerations:** Your orchestration layer has no visibility into east-west traffic between pods in the same cluster. The in-cluster proxy sits at the cluster edge, not in the path of internal service communication.

3. The hybrid pattern



In large enterprise environments, your infrastructure is rarely uniform. You might have developers who prefer YAML workflows in one department, while another team is migrating a legacy bare-metal application to the cloud. The most practical choice is a hybrid model that runs both patterns at once, so you can migrate without a cutover date.

The goal is to stop paying the mesh tax for the workloads you can move, without forcing a migration for the ones you can't. The hybrid pattern does this in two ways. It uses the low-latency external gateway for direct-to-pod routing where possible, and it keeps the ingress controller path in place where teams have deployments they need to preserve.

The gateways manage the inner edge wherever they can. Kubernetes keeps running the local routing for the teams that stay on the in-cluster

path. Where you need both Kubernetes and per-pod enforcement inside the cluster, a service mesh is still the right tool for specific workloads.

This model deploys an external gateway for the outer edge, and a mix of external gateways and in-cluster proxies for the inner edge. It lets you keep existing clusters running while adopting the Universal Mesh for new, high-performance workloads.

- **Conceptual boundaries:** The orchestration layer and gateways manage the outer edge and as much of the inner edge as possible. Kubernetes manages the remainder of the inner edge, reducing the complexity of managing security and observability inside and between clusters.

- **Data path:** Your data path adapts to the workload. For in-cluster proxy teams, traffic follows the multi-hop path through the internal ingress controller (*Internet → External Gateway → Internal Ingress Controller → Pod*). For optimized workloads, traffic takes the single-hop path directly from the external gateway to the pod or server.

- **Kubernetes API interaction:** The same Kubernetes API drives both paths. The internal ingress controller watches the local Kubernetes API for YAML routing updates. The external gateway discovers endpoints from the same objects your teams already publish, so it fits Kubernetes-native workflows without a separate configuration system.

- **Configuration management:** Responsibility is split based on the team. Local YAML controls internal application routing, while the orchestration layer manages routing and security for the external

gateways that sit in front of your clusters.

- **Observability:** The orchestration layer acts as the global observer for the network. Where Kubernetes is managing the inner edge, internal ingress controllers provide localized metrics for their specific clusters, which you need to gather and manage separately.

- **When to use it:** Use this when you are migrating legacy infrastructure or when you can't disrupt your developers' existing workflows. It lets you gradually transition to the external gateway pattern without forcing a rip-and-replace of your network.

- **Considerations:** This is a migration bridge, not a final destination. Because it retains internal ingress controllers in certain clusters, it doesn't fully consolidate your infrastructure. You'll still need localized service mesh components for pod-to-pod enforcement in specific environments.

What the cluster boundary costs

These three patterns describe where your gateways and routing intelligence live. One architectural fact holds for all three, whichever pattern you pick.

Every pattern in this chapter meets the same limit. Nothing outside the cluster can see the traffic that stays inside it. While the Gateway API has matured to support service meshes inside the cluster, it doesn't dissolve this architectural isolation.

Like the Ingress standard before it, the Gateway API remains **cluster-scoped**. It can manage local mesh rules inside its own boundary,

but your global orchestration layer sits outside it and still can't see or secure that internal pod-to-pod traffic.

Cluster as a black box

For traffic that crosses clusters, the in-cluster proxy pattern makes each cluster a opaque system. A request from a pod in Cluster A to a pod in Cluster B passes through two separate controllers, and nothing ties the two halves together.

Picture a security audit that needs to trace one request from a frontend pod in Cluster A to a backend pod in Cluster B. Distributed tracing like OpenTelemetry can track it, but that adds operational complexity. Your teams must instrument every application, manage context propagation, and maintain a heavy tracing stack. Without that extensive setup, you are left attempting to manually correlate fragmented logs across different ingress controllers and meshes.

The Universal Mesh removes that complexity. The request travels cleanly gateway-to-gateway, out of Cluster A's external gateway and into Cluster B's. The infrastructure captures it natively as a single, continuous telemetry event, so you get end-to-end network visibility with zero application-side changes.

This same gateway-to-gateway model enforces mTLS at each gateway, enables identity-aware routing, and delivers the traffic-enforcement layer of a zero-trust architecture. Every request moving between your environments is authenticated, routed, and logged in one place, and you can export it to other systems through an API.

Traffic that never leaves a cluster

That covers everything that crosses a boundary. There is one case it doesn't, and it applies to every pattern in this chapter.

A gateway can only secure traffic that passes through it. When a request enters a cluster, leaves it, or travels between clusters, it goes through a gateway, and the gateway acts on it.

However, when one pod talks to another pod inside the same cluster, that traffic bypasses the external gateway entirely. Only a localized tool like a service mesh sees it (or you route it through a gateway, which we'll come to next). Otherwise, it stays on the internal network by default, leaving nothing for the external gateway to see or enforce.

Three ways to handle internal traffic

For traffic that stays strictly inside a single cluster, you have three options, and they trade off management effort against simplicity.

The first is to trust the internal network and apply nothing. Inside a single cluster, on a private network, behind a gateway that already scrubbed everything coming in from outside, some teams decide pod-to-pod traffic doesn't need its own enforcement. This is the least work and the worst protection. It can be a reasonable call for low-sensitivity workloads, and a poor one for anything regulated or multi-tenant.

The second is to use the controls Kubernetes already provides. Network Policies act as an internal firewall, deciding which pods may talk to which others by IP and port. Pod Security Standards constrain what a workload can do at runtime. These are reliable and require no sidecar, but they are

a second system to manage: a separate policy model, owned and kept current alongside whatever your gateway layer enforces.

The third is to route internal traffic through the external gateway as well, so the same enforcement point that governs traffic crossing the boundary also governs traffic inside it. This collapses everything into one place: one policy model, one set of logs, no second system to maintain. The catch is latency: traffic leaves the cluster and comes back in, a hairpin.

Sending same-cluster traffic to a gateway and back adds a hop, and as we've said throughout, hops cost latency. This option only makes sense when the **gateway is fast enough** that the added hop is negligible, at which point the operational simplicity can outweigh the cost. With a slower gateway, you'd be trading real latency for convenience, and the math stops working.

Which option fits depends on your workloads and your appetite for managing a second control system. The gateway model doesn't force the choice on you. Whether you trust the internal network, lean on in-cluster controls, or pull internal traffic through the gateway, you get defense in depth. No proxy sits in front of every pod.

You eliminate the cluster boundary as a blind spot for the traffic that matters most. You close the remaining gap whichever way fits your workloads, without paying the mesh tax on every pod to cover a case that, for most teams, barely needs it.

Comparing the paths

Here's how the three paths compare across discovery, latency, observability, security, and operational complexity.

(Note: All three paths assume an orchestration layer for edge security and global observability.)

	Path 1: External gateway	**Path 2: In-cluster proxy**	**Path 3: Hybrid**
Routing source	Discovery-based from Kubernetes objects, orchestration layer APIs, or both	Gateway API, Ingress rules, CRDs	Blends local Kubernetes YAML with global API policies
Data path	Single hop via direct-to-pod routing	Higher latency with more hops	Mixed
Observability	Holistic; native L7 observability for north-south and cross-cluster east-west	Fragmented; requires additional tooling for L7 east-west observability	Fragmented by department/workload

Internal traffic security	Native gateway-to-gateway mTLS; no sidecars required	Needs service mesh for per-pod enforcement	Selectively reduces the need for service mesh
Operational complexity	Low: orchestration layer manages global fleet	High: ingress config required per cluster	Medium: local controllers plus centralized global policies
Sidecar mesh	Actively replaces internal proxies and sidecar meshes	Adds proxy complexity inside the cluster	Allows for gradual consolidation alongside the Universal Mesh
Multi-cluster routing	Natively unified across clusters, VMs, and regions	Routes between localized ingress controllers	Orchestration layer manages direct gateways globally while ICs handle local traffic

Internal traffic	Securely handles internal traffic gateway-to-gateway, eliminating sidecars for cross-cluster traffic	Needs a service mesh to secure internal traffic	Selectively reduces the need for sidecar meshes, relying on gateway-to-gateway for multi-cluster traffic
Security model	Native security consolidated at the gateway boundary, with the flexibility to apply distinct internal, external, or per-app policies	Scrubbed at the outer edge, but the orchestration layer can't see east-west traffic inside the cluster	Scrubs at the global edge while enforcing unique local routing rules inside
Architecture	Sidecarless / flat	Layered / complex	Complex migration bridge

Flexibility matters

Your architecture should adapt to your current needs, but it should also be able to evolve. The external gateway pattern puts one enforcement point in front of cross-cluster and external traffic. Policy lives in a single place, and requests take the shortest path to your services. It also gives you visibility into that traffic, because every request crosses a gateway.

The in-cluster proxy and hybrid patterns are starting points too. Choose a platform that lets you start where your team is and evolve as your scale

and security requirements demand, without giving up your orchestration layer.

The next chapters show how these patterns solve real multi-cluster problems, and how HAProxy makes them work in practice. Start with the pattern that fits your team today.

Chapter 4: The multi-cluster problem

The multi-cluster problem starts where per-cluster networking stops working. If your organization runs a small number of clusters, the in-cluster proxy or hybrid patterns can serve you well for a long time. That holds as long as teams are comfortable managing routing inside each one.

But there is a threshold. Organizations that cross it recognize it clearly in retrospect: the moment when the number of clusters, regions, and teams made the per-cluster approach stop working. Failovers that required manual intervention. Load balancing that maxed out at the cluster boundary. Developers who needed cross-cluster routing and couldn't get it without a support ticket. Observability that required stitching together logs from four different systems to answer a simple question about where a request failed.

This is the multi-cluster problem. It's a common reason that organizations with serious Kubernetes infrastructure decide their current networking approach isn't enough.

This chapter explains why it happens, why cluster-scoped tools can't fully solve it, and what changes architecturally when you have a control plane that can communicate with all clusters.

How organizations end up with many clusters

Most organizations that run Kubernetes at scale end up with more than one cluster for many reasons.

Teams typically run production, staging, and development as separate clusters for isolation and cost control. Geographic distribution pushes teams toward regional clusters to reduce latency for clients and meet data sovereignty requirements. Large organizations often give individual product teams or business units dedicated clusters to prevent one team's deployments from affecting another's. Multi-cloud, or cloud-provider diversity (running some workloads on AWS, others on GCP, others on-premises), naturally leads to cluster sprawl.

None of these are bad decisions. They reflect real organizational and technical constraints. But they produce infrastructure that Kubernetes's built-in networking model wasn't designed to coordinate. The rest of this chapter shows what breaks first, and what you can do about it.

What Kubernetes is missing

Kubernetes networking is deliberately cluster-scoped. An ingress controller knows what's running inside its cluster because it watches that cluster's API. It has no mechanism to query a different cluster. A Service in Cluster A cannot natively route to a pod in Cluster B. `kube-proxy` maintains routing rules for the nodes it runs on, and those rules stop at the cluster boundary.

This is a design boundary, not an oversight. Kubernetes manages the lifecycle and networking of containers within a cluster. Coordinating traffic across clusters is a different problem, and one that Kubernetes leaves to you to solve.

The solutions teams reach for vary. DNS-based global load balancing distributes traffic between clusters, but only at a coarse level. Custom controllers watch multiple Kubernetes APIs and try to keep routing tables synchronized. Cloud provider tools work well inside one provider's

environment and break down the moment you step outside it. Others accept the limitation and design applications to be fully self-contained within a single cluster, trading away resilience.

None of these are wrong. But none of them provides what a genuine control plane provides. A control plane is a single layer that watches every cluster in the fleet, keeps an accurate picture of the services running, and routes across all of them in real time.

Failover at the pod level

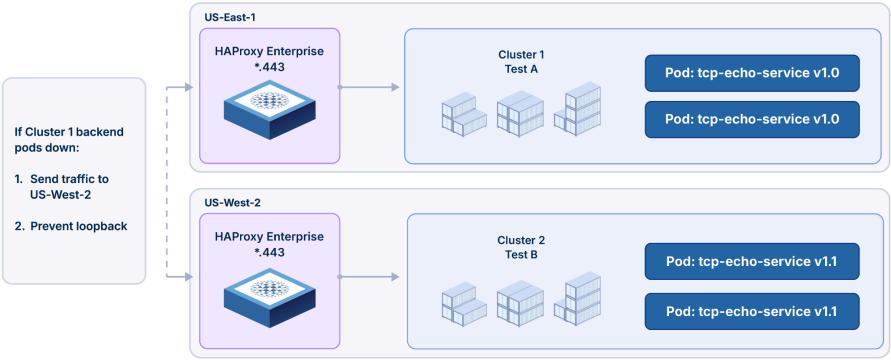
The consequences of not having cross-cluster visibility show up most clearly during failures.

A cluster goes down in an in-cluster proxy setup. If you're running active/active across two clusters, global DNS or cloud load balancing reacts on a health-check interval, not instantly. DNS adds a second delay on top: TTLs (time-to-live) and resolver caching. Even after DNS detects the failure, clients keep hitting the failed cluster until those caches expire.

DNS-level failover is binary. It knows whether a cluster endpoint responds, but it cannot see service health inside. While Kubernetes manages readiness probes *within* a cluster, it provides no native way to share those probe states across cluster boundaries. As a result, external routing can't tell when a cluster is reachable but all pods behind a specific service are failing their readiness probes.

Making that distinction requires a control plane that performs continuous service discovery across your clusters. It knows whether a cluster's API is responding and whether the pods behind each service are healthy. When pods start failing health checks in one cluster, it shifts traffic to the

corresponding pods in another. No DNS TTL has to expire, and no one has to intervene.



This is the difference between cluster-level failover and pod-level failover. Kubernetes-native tools can make that call inside a single cluster; a control plane that spans every cluster makes it across the fleet.

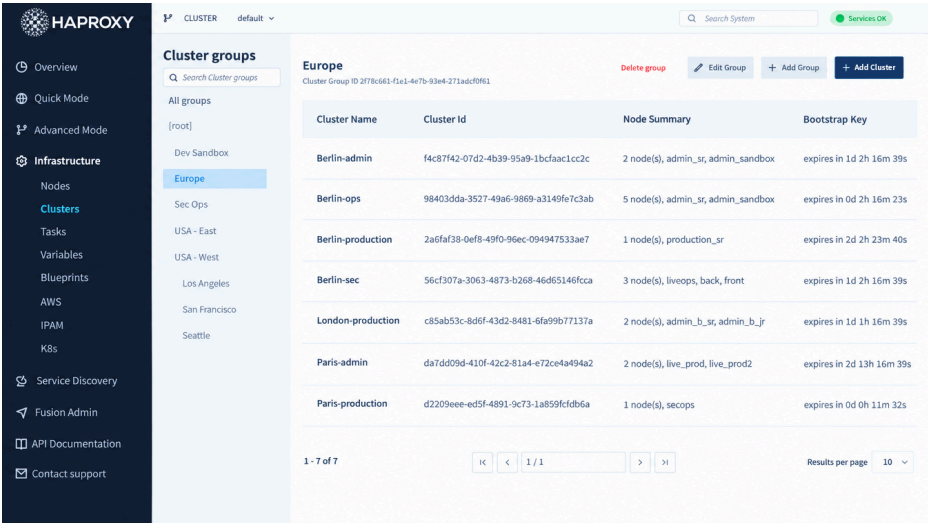
Load balancing without boundaries

Even when all clusters are healthy, the lack of cross-cluster coordination creates a load balancing problem.

The ingress controller distributes traffic across the pods it knows about, which are the pods in its own cluster. If you have two clusters in two regions, and 80% of clients are in one region, you can tune your DNS weights to send more traffic to the closer cluster. But that's a coarse adjustment, and it's not responsive to the actual load or health of individual pods.

A control plane sees the pod population across both clusters simultaneously. It can distribute traffic based on backend availability rather than static cluster weights, tracking how many healthy pods are

running across each environment. When a deployment rolls out in one cluster and temporarily reduces its pod count, traffic automatically redistributes across the active endpoints according to your load balancing algorithm. Service discovery tracks the live state of the infrastructure, without requiring manual weight adjustments during routine scale events.



This is also where direct-to-pod routing matters at scale. A request can go straight from the external gateway to the pod that will handle it, regardless of which cluster that pod is in. The gateway makes the routing decision once, with full information.

Compare that to a multi-hop path: the request hits a cluster-level load balancer, then an ingress controller, then `kube-proxy`, before it reaches a pod. Each hop operates on less information than the one before it.

Blue/green deployments across clusters

The multi-cluster problem also shows up when teams try to implement deployment strategies that span clusters.

Blue/green deployments are a good example. The concept is straightforward: run two identical environments, route traffic to one, deploy changes to the other, verify them, then shift traffic. If something goes wrong, shift back immediately.

Inside a single cluster, most ingress controllers can do a version of this. But a blue/green deployment that spans clusters needs traffic control at the boundary between them. Your "green" environment might sit in a different cluster or region from your "blue." An ingress controller running inside Cluster A cannot shift traffic between Cluster A and Cluster B. It only knows about Cluster A.

With a control plane coordinating gateways that sit outside both clusters, this becomes a configuration change rather than an infrastructure operation. You define how much traffic goes to each cluster, adjust the weights as you gain confidence in the green environment, and shift fully when you're ready. Rollback is the same operation in reverse: shift the weights back. No cluster-level changes required.

The same logic applies to A/B testing, canary releases, and any other strategy that requires coordinated traffic control across multiple environments. Set and adjust the weights to match the split you want.

How service discovery works

Service discovery is the mechanism by which the control plane knows what's running in your clusters at any given moment. But if the control

plane sits outside the cluster, how does it learn pod locations to keep the gateways updated on where to route traffic?

The control plane connects to the Kubernetes API of every cluster and watches in real time as services come online and go down. A scaling event or a node failure triggers the same response. Each change maps back to the service it belongs to, and the routing configuration updates automatically. That mapping has to hold up whether a cluster is running ten pods or ten thousand.

In standard Ingress and Gateway API deployments, achieving high availability requires running multiple controller replicas. During rapid pod scaling, every instance independently watches the Kubernetes API and reloads its underlying proxy engine, compounding load on the API server and triggering configuration reload storms.

A decoupled control plane with centralized service discovery avoids this. Rather than let multiple controller replicas hammer the Kubernetes API, the control plane batches changes into a single update using a coalescing interval. That also stops the gateways from drowning in a constant storm of individual updates. This keeps configuration drift from creeping in, so the data plane stays synchronized with the infrastructure even under aggressive scaling.

This synchronized discovery loop runs continuously across every cluster the control plane connects to. An ingress controller running inside a single cluster watches the same objects, but its view ends at the cluster boundary. The external control plane has no such boundary. It sees the same level of Kubernetes object details across ten clusters that an ingress controller sees across one.

Controlled self-service at enterprise scale

There is another dimension to the multi-cluster problem, and it is less about traffic routing than about governance. How do you give development teams the ability to manage their own routing without letting any team accidentally affect the others?

At a small scale, platform teams manage developer access by maintaining separate `IngressClass` resources and localized Helm templates. At enterprise scale, this fragmented model breaks down.

Independent deployments across teams frequently suffer from path collisions or hostname shadowing within the same ingress scope. Since standard ingress controllers operate in isolated silos without a service-discovery-capable control plane, they cannot detect cross-cluster routing conflicts or enforce global governance before it affects live traffic.

These aren't hypothetical problems. They're why platform teams in large organizations build guardrails around Kubernetes networking. It’s also why self-service governance comes up consistently in enterprise Kubernetes conversations.

A service-discovery-capable control plane addresses this at the architectural level rather than through a team process. The control plane adapts to your existing setup, so application teams don't have to refactor their configurations to fit a rigid standard.

In the real world, this means applications don’t need to change at all; you update a mapping rule in the control plane to match their nomenclature, and it works. Developers apply those labels and annotations to their services, in YAML manifests, CI/CD pipelines, or directly via the Kubernetes API. The control plane's continuous service discovery picks them up

automatically and routes traffic to those services, with no manual ticket or configuration change from the network team.

The administrator decides the limits here, not the software vendor. You expand the system by adding features through the control plane, with no disruptive software upgrade.

Crucially, the control plane doesn't invent the governance boundary; it inherits it. Kubernetes already determines the scope of what any team can affect, through the namespace-scoped permissions and RBAC that decide which teams can annotate which services. The control plane operates within those parameters, no more, and extends that same model across every cluster in the fleet through a single interface.

What becomes possible

The multi-cluster problem becomes a routing decision you make once and apply everywhere. That happens when you have a control plane with cross-cluster visibility, continuous service discovery, and routing governance you can enforce across the fleet.

Failover becomes service-aware, not just cluster-aware. Load balancing decisions are made with full visibility into the actual health and capacity of backends across all clusters. Deployment strategies that span clusters become routine configuration changes. Observability covers the full request path across cluster boundaries from a single view. And self-service for engineering teams doesn't require choosing between autonomy and operational safety.

Organizations that have adopted the control plane and gateway architecture cite multi-cluster coordination as the capability that justified the architectural shift.

The next chapter shows these capabilities in real-world scenarios. Blue/green deployments span clusters. A/B testing targets real user segments. Self-service routing doesn't sacrifice governance. Security is enforced at the boundary before traffic reaches a cluster. Observability covers the full request path end to end.

Each one is a problem in-cluster networking only partially solves, and each is solved by the same control plane and gateway layer this chapter described.

Chapter 5: Real-world applications

The previous chapters made the architectural case for the **Universal Mesh** pattern; this chapter shows what running it looks like in practice.

Two components do the work: **HAProxy Fusion** is the control plane, watching clusters, discovering services, and pushing routing configuration, and **HAProxy Enterprise** is the gateway layer at the network boundary (chapter 6 will cover both of these components in detail).

Below are five real-world applications. Each scenario is common enough for most users, but is still challenging and worth solving.

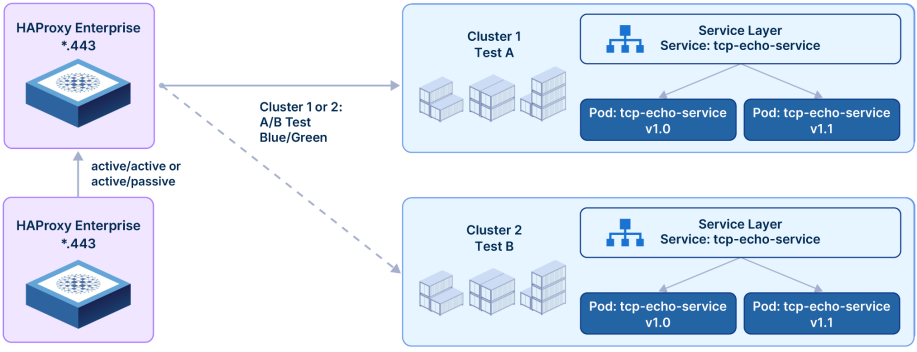
1. Blue/green deployments that span clusters
2. A/B testing that targets real user segments
3. Self-service routing without losing governance
4. Security enforced at the boundary, before traffic reaches a cluster
5. End-to-end observability across the full request path

Deployment strategies

The first two scenarios are both about controlling which version of an application receives traffic, but they solve different problems. Blue/green is about safely moving from one version to the next. A/B testing is about deliberately splitting traffic to measure a difference. Both depend on the same capability: precise, centrally managed traffic control at the boundary.

Blue/green at scale

The previous chapter introduced blue/green as a multi-cluster coordination challenge. For this scenario, you need something that can control traffic routing at the boundary between two environments, but an ingress controller running inside one cluster can't see the other cluster.



HAProxy Fusion manages the orchestration, and HAProxy Enterprise handles traffic at the outer edge. Together they turn a blue/green deployment across clusters into an automated configuration change rather than an infrastructure operation.

HAProxy Fusion tracks the pod populations in both environments in real time. HAProxy Enterprise applies traffic weights at the gateway, shifting a percentage of requests to the green environment while the rest continue to blue. The shift can be gradual: start at 5%, watch error rates and latency, then move to 25%, then 50%, then complete the cutover. It can also be immediate if you're confident in the new environment.

If something goes wrong after the cutover, rolling back is simple. Traffic weights flip back. No changes inside the cluster. No DNS propagation wait.

The gateway is already watching both environments; it just changes which one it prefers.

A/B testing and header-based routing

A/B testing splits traffic on purpose and measures the difference against your goals. That's a different job from blue/green, which moves traffic from one version to another.

In a typical A/B test, you split traffic between an established incumbent (version A) and a new challenger (version B). You route a controlled slice of live traffic to the challenger while the main cohort continues hitting the incumbent baseline, measuring conversion rates, error spikes, and latency profiles. Once sufficient data is gathered, you either promote the challenger to become the new incumbent, retire it, or iterate with the next challenger.

Ingress supports traffic splitting, but only through basic weight-based rules. On routes with variable or low traffic volume, a simple percentage split often fails to deliver enough requests to the challenger to reach statistical significance. Coarse weight-based splitting cannot target specific user segments, evaluate request headers, or pin traffic by session identity.

With HAProxy Enterprise at the boundary, routing can key on any request attribute: HTTP headers, cookies, source IPs, user agent strings, or a custom header your application sets. That means you can target a specific user cohort (clients with a particular cookie value or session ID) instead of a random sample of all traffic. You define the split with L7 routing logic, the same configuration language that governs every other routing decision, and manage it centrally through HAProxy Fusion.

In practice, a product team can define the parameters of their A/B test: which header to look for, which value routes to which version, and what percentage fallback applies to clients who don't match. Then they apply it without touching the cluster at all.

The team that owns the gateway configuration (usually platform or DevOps) applies the rule through HAProxy Fusion. When the test is done, the team removes the rule. The change is audited, reversible, and doesn't interfere with any other team's routing.

Multi-cluster A/B testing routes across incumbent and challenger workloads spread over different clusters. HAProxy Fusion tracks service endpoints across your entire fleet, allowing HAProxy Enterprise to enforce routing decisions across multi-cluster backends. This doesn’t require application code changes or workflow refactoring, leaving developers free to focus on business logic.

Self-service without chaos

Chapter 4 framed self-service as a governance problem the architecture solves. Developers want to manage their own routing; operators need to ensure no team can affect another. A control plane resolves that tension structurally. Here's what that looks like with HAProxy Fusion in place.

The platform team establishes an annotation convention, the shared vocabulary developers use to express routing intent. A developer who needs a service exposed doesn't open a ticket and wait for the network team to create a domain or update a config file. They apply the agreed annotations to their service, and HAProxy Fusion's continuous service discovery picks it up automatically and routes to that service. The workflow is self-contained and immediate.

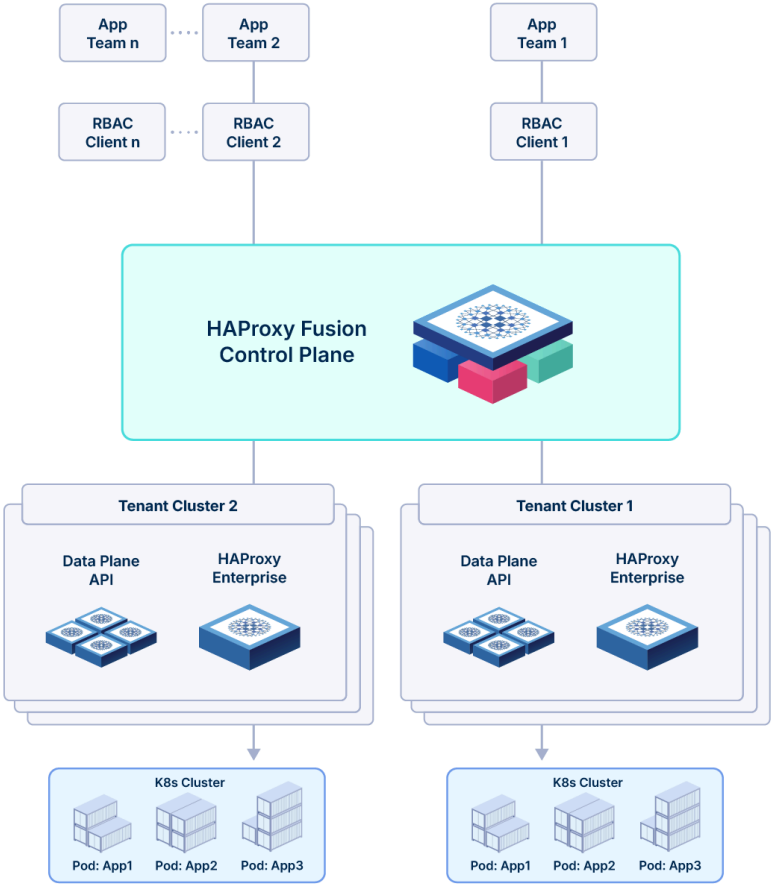
In practice, the convention is small. Exposing a service takes a few annotations on its Service manifest:

```
metadata:
  annotations:
    fusion.haproxy.com/ipam-definition: 192.0.2-range
    fusion.haproxy.com/routing-policy: pods
```

One names an address pool, and one tells HAProxy Fusion to send traffic straight to the pod IPs rather than through the worker nodes. That's the direct-to-pod path: the gateway layer reaches the application without the cluster's internal routing layers in between. Neither annotation describes a route: that part stays with the platform team. If you run more than one cluster, a third annotation names which cluster the configuration should go to.

The governance side lives at the Kubernetes layer. HAProxy Fusion connects to the Kubernetes API using a service account or an external kubeconfig that the cluster administrator controls. Respecting native Kubernetes RBAC, it needs read access to the namespaces, services, labels, and annotations the administrator scopes it to. It gets write access only to the services it assigns addresses to, and never reaches outside that designated scope.

The platform team configures those permissions deliberately, so each team's annotations can only affect their own services. HAProxy Fusion inherits those boundaries; it doesn't define them.



This is also why the model is safe by default rather than by enforcement. A label only does something if HAProxy Fusion has been configured to act on it. The platform team decides which keys and values carry meaning; everything else is inert.

A developer who applies an annotation that hasn't been agreed on, or mistypes one that has, doesn't trigger an error or a conflict. Nothing happens at all, because HAProxy Fusion has no rule that responds to it. There's no way to accidentally reach past the agreed vocabulary, because

reaching past it has no effect. The only annotations that do anything at the control plane level are the ones the platform team already sanctioned.

This also makes the model easy to expand. When the platform team needs to introduce new routing capabilities, there is no complex integration process. They agree on the new keys and values, configure HAProxy Fusion to act on them, and it's done.

Two things make this matter in practice. First, the vocabulary and discovery behavior are identical across every cluster in the fleet. HAProxy Fusion, as opposed to per-cluster ingress configuration, is the single point where routing intent becomes routing reality. New clusters inherit the model automatically.

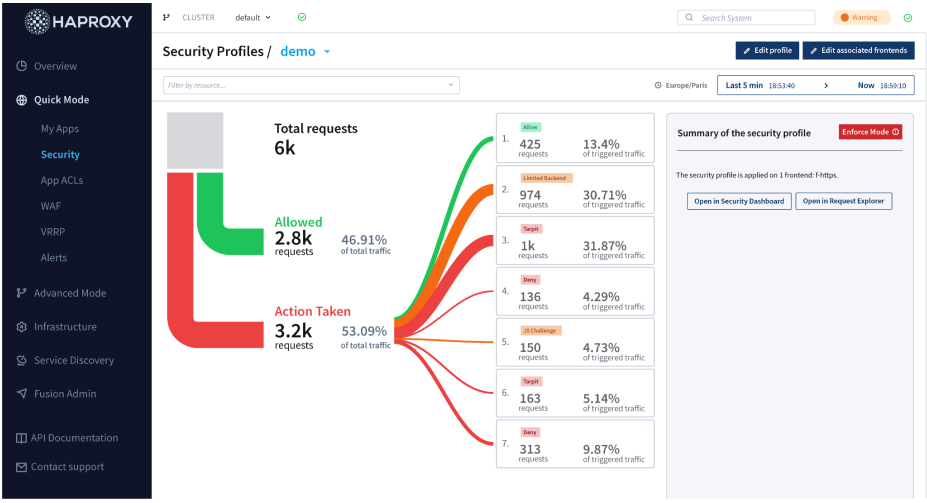
Second, because every configuration change flows through HAProxy Fusion before reaching a gateway, there's a complete audit trail. When something goes wrong with a routing rule, the question "who changed what and when?" has a clear answer, which alone resolves a whole class of incident investigations.

For developers, the experience is clear, intuitive, and allows them to stay in their lane. They don't need to become network engineers, figure out complex Kubernetes `Ingress` specs, or wrestle with Gateway API resources just to expose a service.

The biggest hurdle to deploying a new traffic management solution is the demand it places on application teams to rewrite their existing manifests. HAProxy Fusion eliminates this friction by adapting to them, mapping to the labels, annotations, and workflows they already use today. The annotation vocabulary functions as a contract between the application team and the platform, requiring no application-side reconfiguration. Developers ship without waiting. Operators govern without firefighting.

Security at the boundary

Security in a Kubernetes environment often works its way inward. Teams add network policies inside the cluster and inject sidecars for mTLS. These controls are valuable. But there's an efficiency problem: by the time a malicious request reaches any of those controls, it has already consumed cluster resources.

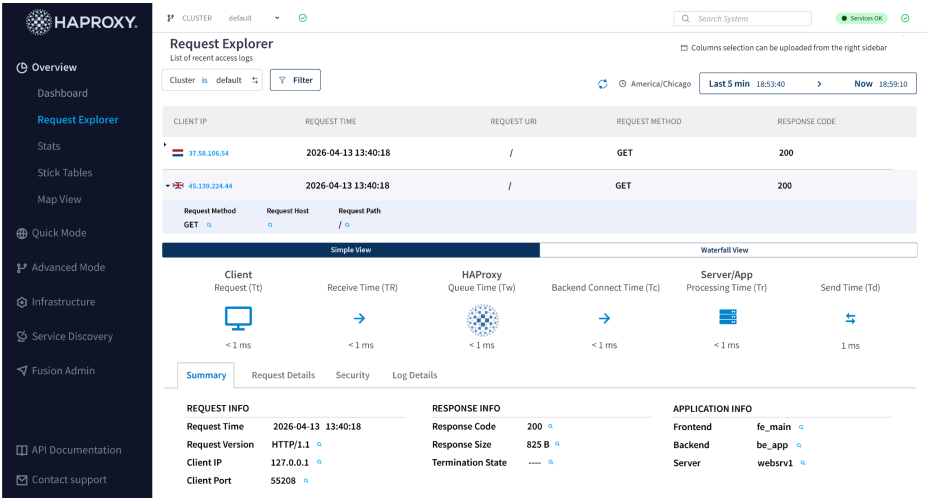


The more efficient model puts security enforcement where traffic arrives first: at the outer boundary.

HAProxy Enterprise, positioned at the outer edge, applies security controls before traffic enters any cluster at all. This includes:

Web application firewall (WAF). HAProxy Enterprise WAF, powered by the Intelligent WAF Engine, achieves 99.65% balanced accuracy (99.45% true-positive, 99.81% true-negative) with sub-millisecond latency. It operates at the gateway level, inspecting requests at L7 before they reach your application pods. Because it sits outside the cluster, WAF

enforcement is consistent regardless of which cluster eventually serves the request.



Bot management. The HAProxy Enterprise Bot Management Module identifies and classifies request sources: legitimate users, known bad actors, automated scanners, scraping tools, application-layer DDoS probes. Classification happens at the gateway, not inside the cluster. Threats are blocked, challenged, or rate-limited before they consume application resources.

DDoS mitigation. Large-scale DDoS attacks are absorbed at the boundary. Application pods never see the volume; the gateway handles the scrubbing. This applies to both network-layer and application-layer attacks.

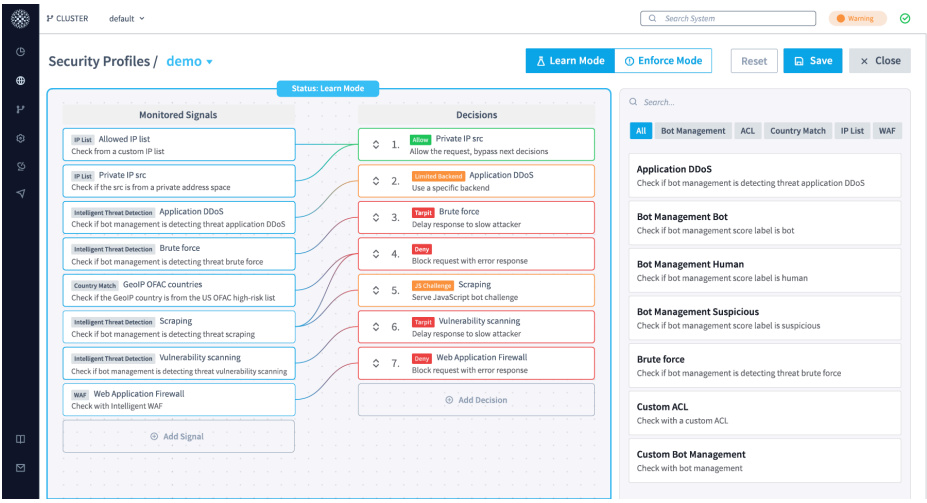
Rate limiting. Rate limiting guards against API abuse, brute-force login attempts, and credential stuffing. Stick tables do the counting, and the Global Profiling Engine (GPE) aggregates those counts cluster-wide in real time, so you can track client behavior across your entire multi-cluster fleet.

Standard Kubernetes ingress controllers can only count requests locally within a single proxy instance, meaning an attacker can bypass rate limits by distributing requests across multiple ingress replicas and clusters. With HAProxy Fusion, the Global Profiling Engine aggregates those counts globally in real time.

A malicious actor sending five requests per second across Cluster 1 in North America and Cluster 2 in Europe is counted once at the fleet level, not per cluster. When the threshold is crossed, a sub-millisecond block or a native CAPTCHA challenge is enforced at the gateway boundary.

HAProxy Fusion manages these controls across a distributed fleet. It acts as the security control plane. Instead of configuring WAF rules per cluster or per gateway instance, you define security profiles in HAProxy Fusion and push them to every Enterprise gateway simultaneously. A policy change (a new WAF rule, a bot scoring threshold, a rate limit update) applies globally in a single operation.

Two HAProxy Fusion features make this practical at scale. **Security Profiles** let SecOps teams define and apply preset security postures without needing to work directly in HAProxy's configuration language. The **Threat-Response Matrix** provides a visual policy builder that maps monitored signals (specific request patterns, anomaly scores, rate thresholds) to automated Decisions (block, tarpit, challenge, flag). Teams configure threat response logic through the interface rather than through configuration code.



Deploying a new profile in Learning Mode evaluates real traffic by simulating block or challenge actions without active enforcement. This deployment mode validates rules efficacy and minimizes false positives, which is particularly useful when onboarding new applications or hardening policies in response to emerging threats.

Boundary security doesn't replace internal cluster security. Network policies, pod security standards, and namespace isolation all still apply within the cluster. What the boundary layer provides is a high-throughput first line of defense. It handles most threats before they reach internal controls, which reduces the load on those controls and gives you one place to inspect and respond to threats.

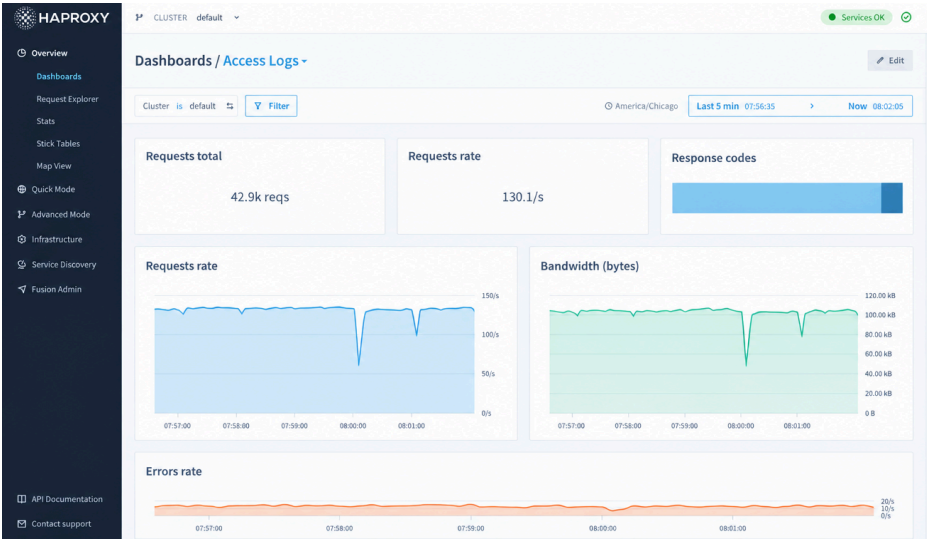
End-to-end observability

Kubernetes networking creates an observability problem that grows with scale. When traffic flows through multiple layers (a cloud load balancer, an ingress controller, `kube-proxy`, and finally a pod), the logs for a single

request are spread across several systems. When something goes wrong, tracing the request path means stitching together data from each of those layers, often using different tooling for each.

This becomes difficult at enterprise scale across multiple clusters. A request that enters in one region, hits a cluster in another, and fails at the pod level inside that cluster leaves traces in at least three systems. Finding those traces quickly, correlating them correctly, and interpreting what happened requires serious tooling investment or deep institutional knowledge (usually both).

HAProxy Fusion solves this by pulling telemetry from every HAProxy Enterprise gateway instance into a single view. HAProxy Enterprise is the point through which all traffic passes, so it's the right place to capture the full picture. That picture includes request volume, latency by backend, error rates, connection counts, health check results, and individual request data.



The result is an end-to-end view that doesn't require stitching. A request may be served by a pod in Cluster A or Cluster B, routed by the external gateway or through an ingress controller in the hybrid pattern from Chapter 3. Either way, the telemetry flows to HAProxy Fusion. Teams can logically group clusters and services, making it easier to distinguish a problem isolated to one region from one affecting traffic globally.

The **Request Explorer** extends this to individual requests. When an error or anomaly shows up in aggregate metrics, you can drill into specific requests. You'll see which gateway received them, which backend they were routed to, what health check state that backend was in, and what the response looked like.

This is the kind of tracing capability that typically requires a full APM stack. Here it comes from the gateway layer, without instrumentation changes to your application code.

While HAProxy Fusion acts as your central view, it never locks your data into a closed system. HAProxy Enterprise natively supports OpenTelemetry, the standard for Kubernetes observability. Metrics, connection stats, and trace data stream directly from your gateways to standard OpenTelemetry collectors or APM tools like Grafana, Datadog, and Dynatrace. As a result, platform teams get full gateway visibility inside HAProxy Fusion while continuing to use their existing monitoring pipelines.

A few specific situations where this observability pays off:

During deployments. When you shift traffic from blue to green, watch error rates and latency for both environments in real time. HAProxy Fusion provides this from the gateway perspective, so you can watch the shift unfold and stop it immediately if the green environment starts producing errors.

During incidents. When application behavior changes unexpectedly, the first question is: is this networking or application? Gateway-level telemetry answers the networking side of that question quickly. If requests are arriving normally and being routed correctly, the problem is downstream. If they're not arriving at all, or being routed to a failed backend, the problem is at the network layer.

For capacity planning. Traffic patterns at the gateway level — which services get how much traffic, when, and from where — are key inputs to capacity planning. HAProxy Fusion aggregates this data across your fleet, making it possible to see patterns that aren't visible when each cluster is observed in isolation.

The architecture in action

Each scenario above has two things in common. First, it's a problem that Kubernetes-native networking addresses partially or not at all. Second, the same underlying architecture solves it: a control plane managing a gateway layer that handles all traffic at the boundary.

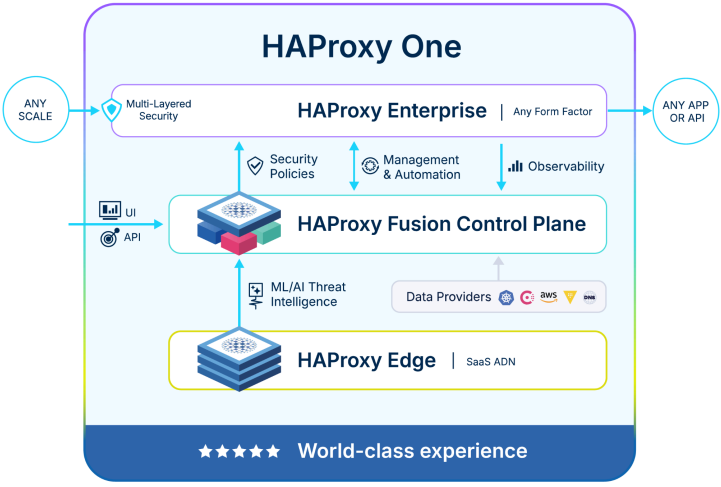
The platform implementing this architecture (using HAProxy Fusion as the control plane and Enterprise as the gateway layer) performs concrete functions in every scenario. The control plane manages traffic weights centrally and applies them at the edge, and it validates routing rules before they reach a node. It also pushes WAF profiles globally for local enforcement and captures telemetry at the gateway to aggregate in the control plane.

Chapter 6 takes this further. We'll introduce the full HAProxy One platform and map the three architectural paths from Chapter 3 to specific HAProxy components. Then we'll walk through the reference deployment architecture in detail.

Chapter 6: HAProxy One: the platform built on these principles

By this point, you've seen HAProxy Fusion and HAProxy Enterprise doing real work: managing traffic weights for blue/green deployments, applying security globally, and aggregating telemetry across clusters. You've also seen routing governance enforced through an operator-defined vocabulary.

HAProxy One is the platform this book has been building toward. HAProxy Fusion and HAProxy Enterprise are two of its components, each purpose-built for a specific role in the Universal Mesh architecture we've described. This chapter covers what each product is, how they relate, and how the three architectural paths from Chapter 3 map to specific HAProxy Technologies products.



Platform overview

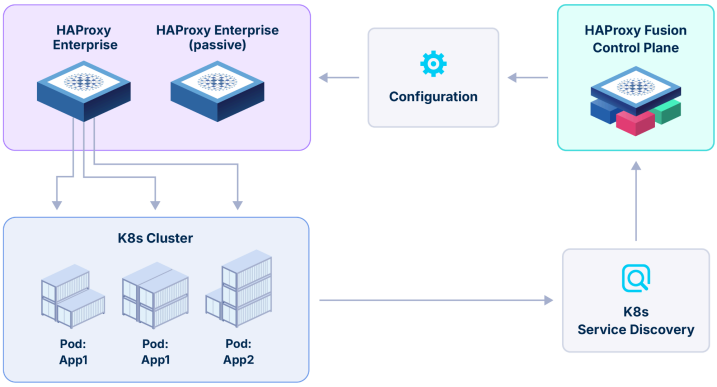
HAProxy One is the world’s fastest application delivery and security platform. It rests on the principle this book keeps returning to: separate the intelligence from the execution, and put each at the layer where it works best. Intelligence and execution then operate together without getting in each other's way.

The platform has three components, each with a defined role. They are Kubernetes-integrated, and beyond the platform, we provide two Kubernetes-native offerings: components that live inside the cluster and let Kubernetes be their control plane.

HAProxy Fusion: the control plane

HAProxy Fusion is the control plane, and it can operate within Kubernetes and beyond the cluster. Rather than being confined solely to container workloads, HAProxy Fusion acts as a centralized control plane capable of orchestrating traffic across pods, VMs, and bare metal from one place.

It connects to Kubernetes clusters, whether it runs outside them or inside one of them. From there, it monitors the Kubernetes API continuously, discovers services and objects, and translates that knowledge into routing configurations for the gateways it manages.



HAProxy Fusion manages, observes, and automates all gateway instances simultaneously. It maintains a live picture of every cluster you connect. That's what makes cross-cluster routing, multi-region failover, and global policy enforcement possible from one UI or the REST API.

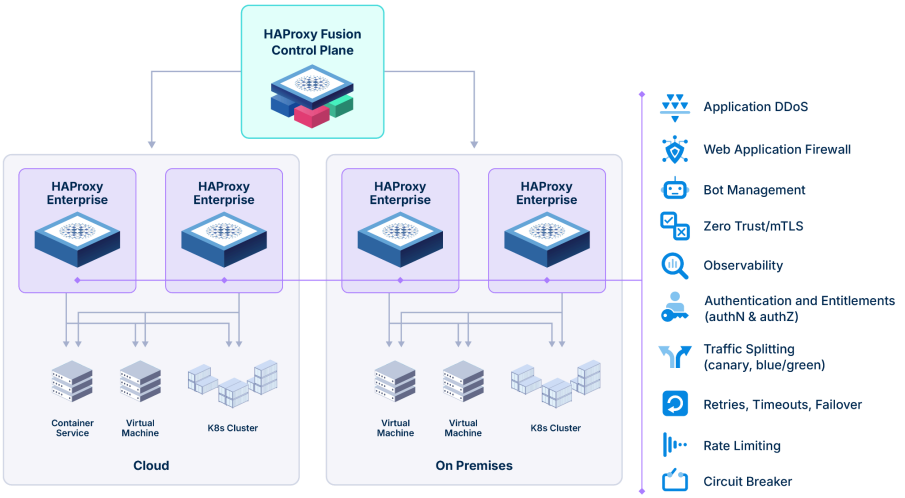
HAProxy Fusion provides five main capabilities:

- **Automated service discovery –** HAProxy Fusion watches service lifecycles across every connected cluster, aggregating changes before pushing clean configuration updates to gateways. Pod churn during scaling events doesn't produce configuration storms; HAProxy Fusion smooths them into coherent updates. It can dynamically generate over 100,000 lines of configuration in seconds without encountering bottlenecks.
- **Multi-cluster management –** You manage traffic distribution, failover, and deployment strategies that span clusters from one central place. The data plane delivers the traffic; you configure active/active, active/passive, and health-aware routing policies in HAProxy Fusion, which applies them globally across all gateways.

- **Policy orchestration –** Teams define security profiles, WAF rules, rate limits, and routing governance policies just once in HAProxy Fusion and can push them to every HAProxy Enterprise gateway simultaneously. HAProxy Fusion applies changes globally in a single operation.
- **Self-service governance –** HAProxy Fusion defines and enforces the annotation and label vocabulary that gives developers self-service routing within operator-defined guardrails. RBAC and concurrency controls ensure teams operate independently without interference.
- **Centralized observability –** HAProxy Fusion aggregates telemetry from each HAProxy Enterprise gateway instance into a unified view, from cluster-wide traffic patterns down to individual request traces.

HAProxy Enterprise: the data plane

HAProxy Enterprise is the high-performance gateway. It usually sits at the boundary of your network, outside the cluster and at the edge of your infrastructure handling your application traffic. However, it can also run *inside* a Kubernetes cluster.



HAProxy Enterprise is built on HAProxy, the world's most widely used software load balancer. The enterprise edition adds multi-layered security and operational tooling that production deployments require, without compromising HAProxy's performance.

For Kubernetes, HAProxy Enterprise does the work that traditionally required multiple overlapping tools: external load balancing, TLS termination, WAF enforcement, bot management, DDoS mitigation, and direct-to-pod routing. Each of these runs in the same process, at the same layer, with no additional hops between them.

Key capabilities:

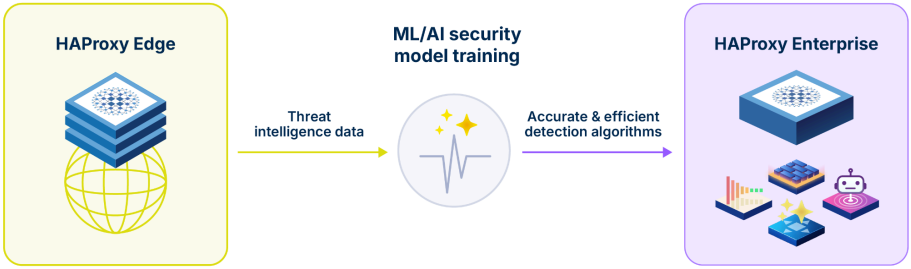
- **Layer 4 and Layer 7 routing –** HAProxy Enterprise handles TCP, HTTP/S, UDP, and QUIC traffic with the same performance profile. It replaces the combination of cloud load balancer and ingress controller with a single gateway.

- **Direct-to-pod routing –** When HAProxy Fusion provides pod IPs, HAProxy Enterprise routes directly to them, bypassing the ingress layer and `kube-proxy` entirely. One hop from gateway to pod.
- **Next-generation security –** HAProxy Enterprise WAF detects threats with 99.65% balanced accuracy while adding sub-millisecond latency. HAProxy Enterprise Bot Management Module classifies and handles threats before they reach application pods. DDoS mitigation happens at the boundary.
- **Advanced SSL/TLS processing –** HAProxy Enterprise supports high-throughput encryption and decryption with linear performance scaling on multi-core systems, using the AWS-LC crypto library for improved performance.
- **Mutual TLS (mTLS) authentication –** HAProxy Enterprise supports mutual authentication between clients and servers using mTLS. By verifying both parties' identities, mTLS strengthens security for applications, APIs, and zero-trust architectures.

HAProxy Edge

HAProxy Edge is the secure edge network that enhances the security layers across the rest of the platform.

HAProxy Edge is an optional SaaS application delivery network (ADN) that provides globally distributed, fully managed DDoS protection, threat scrubbing, and delivery services at the outermost perimeter. For Kubernetes deployments, it adds a layer of protection in front of your HAProxy Enterprise gateways, absorbing large-scale attacks before they reach your infrastructure.



Even organizations that don't deploy HAProxy Edge directly benefit from it. The threat intelligence data that HAProxy Edge collects from its global network trains our machine-learning models behind the WAF and bot management capabilities in HAProxy Enterprise. Every HAProxy Enterprise deployment gets security algorithms informed by real-world attack data, whether or not you use HAProxy Edge.

Kubernetes-native offerings

The three platform components cover the external model in full. But two of the three architectural paths keep some routing inside the cluster, and that calls for components built to run there. HAProxy Technologies ships two Kubernetes-native products for this: an ingress controller for teams on Ingress, and a unified gateway for teams adopting Gateway API.

HAProxy Enterprise Kubernetes Ingress Controller

The HAProxy Enterprise Kubernetes Ingress Controller is the product for Path 2: teams that want their routing logic to live inside the cluster alongside the applications it serves.

HAProxy Enterprise Kubernetes Ingress Controller provides the same performance and reliability that HAProxy is known for, in a fully

Kubernetes-native deployment model. It runs inside your cluster, watches the local Kubernetes API, and translates Ingress rules and annotations into HAProxy configuration. You also get enterprise support and an integrated WAF.

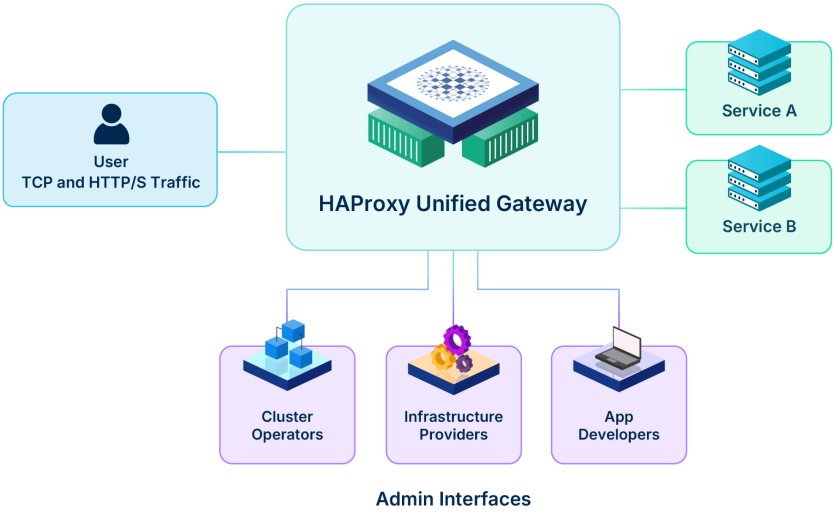
In a hybrid architecture, HAProxy Enterprise Kubernetes Ingress Controller coexists with the external HAProxy Enterprise gateways. The gateways route to HAProxy Enterprise Kubernetes Ingress Controller instances as backend servers for teams that prefer YAML-driven intra-cluster routing. Meanwhile, the rest of the network benefits from the external gateway pattern. Developer teams keep their native workflows. The platform team keeps centralized visibility and security at the outer edge.

HAProxy Enterprise Kubernetes Ingress Controller also provides a migration path to Gateway API through the HAProxy Unified Gateway, which runs Ingress and Gateway API resources side by side.

HAProxy Unified Gateway

HAProxy Unified Gateway provides high-performance, Kubernetes-native routing for organizations adopting Gateway API, whether as a new deployment or migrating from Ingress. Its defining capability is **dual-mode routing**: a single controller that processes both existing Ingress manifests and new Gateway API resources concurrently.

This means organizations don't need to choose a cutover date. They can migrate incrementally, route by route, as teams are ready.



This matters because migration is where most Gateway API adoptions stall. Typical Gateway API tools force a choice: a disruptive cutover, or running two separate controller platforms side by side. That doubles the operational surface just when risk is highest. HAProxy Unified Gateway removes that choice.

With its native Ingress mode, platform teams consolidate the routing tier onto a single engine, then developers upgrade their manifests to HTTPRoutes as each team is ready. The old Ingress resources keep working the entire time.

HAProxy Unified Gateway implements role-based access control (RBAC) through the Gateway API, separating what Cluster Operators, Infrastructure Providers, and Application Developers can each do.

HAProxy Enterprise routes to HAProxy Unified Gateway-managed workloads the same way it routes to HAProxy Kubernetes Ingress

Controller, so organizations that want the Kubernetes-native experience keep a clear path to the external gateway model.

Running the platform in Kubernetes

The two Kubernetes-native products live inside a cluster. The platform components (HAProxy Fusion and HAProxy Enterprise) can run there too. Where you place them is a deployment choice rather than a product choice, and it doesn't change what they do.

HAProxy Fusion in Kubernetes

Since HAProxy Fusion 2.0, you can install Fusion inside a cluster using the HAProxy Fusion Operator, which is currently an experimental feature. You install the operator, then describe the installation in a Fusion custom resource. That resource sets replica counts, resource limits, StorageClass, service type, and whether Fusion manages its own databases or uses externally managed ones. The operator provisions Fusion and keeps it updated, and a single cluster can host more than one Fusion installation.

HAProxy Enterprise in Kubernetes

HAProxy Enterprise runs in a cluster as a Deployment or a DaemonSet. When Fusion and Enterprise both run in Kubernetes, a Kubernetes link authenticates with the cluster's service account instead of a pasted `kubeconfig`. Fusion can also apply custom resource definitions, so you manage service discovery with `kubectl`. Enterprise running in the cluster can also reach pod IPs directly on the pod network, so direct-to-pod routing never leaves the cluster.

Mapping the three paths to HAProxy products

The three architectural paths from Chapter 3 map directly to specific HAProxy Technologies product combinations. The platform is the same in every case. The roles each component plays shift based on which path you're on.

Path 1 pairs Fusion with Enterprise. Path 2 adds the Ingress Controller or Unified Gateway at the inner edge. Path 3 supports both.

Path 1: external gateway

This path uses the full HAProxy One stack. HAProxy Fusion connects directly to the Kubernetes API. It watches Kubernetes services to discover active pod IPs and worker node endpoints, then pushes them to HAProxy Enterprise. HAProxy Enterprise bypasses the in-cluster ingress layer entirely and routes directly to pods or hosts.

You get the same developer self-service here as anywhere else. HAProxy Fusion's service discovery imports from the Kubernetes API, Consul, and other sources, then maps what it finds into gateway configuration using patterns defined by the platform team. Developers keep defining routes through Kubernetes objects and annotations; Fusion reads them and translates them into external gateway configuration automatically.

For cross-cluster east-west traffic, HAProxy Enterprise acts as the mTLS termination and re-origination point at the cluster boundary. Traffic flows from gateway to gateway across clusters, with HAProxy Fusion coordinating routing decisions.

- **Outer edge:** Internet → HAProxy Enterprise → Pod
- **Cross-cluster east-west:** Pod → HAProxy Enterprise → Pod

Path 2: in-cluster proxy

In this path, HAProxy One operates as the global outer edge. HAProxy Enterprise handles inbound traffic and routes it to the appropriate cluster. Inside the cluster, HAProxy Kubernetes Ingress Controller or HAProxy Unified Gateway takes ownership of the inner edge. All L7 routing decisions, pod selection, and service resolution happen there.

From HAProxy Fusion's perspective, HAProxy Kubernetes Ingress Controller and HAProxy Unified Gateway instances are backend servers. HAProxy Fusion routes traffic to them and manages security at the outer edge. It doesn't see inside the cluster boundary. That's intentional: this path is an architectural choice for platform teams that want a dedicated in-cluster proxy layer, not a requirement for developer self-service.

Developer teams retain full control of routing through Kubernetes-native YAML workflows, just as they do in the other paths.

- **Outer edge:** Internet → HAProxy Enterprise → HAProxy Kubernetes Ingress Controller or HAProxy Unified Gateway → Pod
- **Intra-cluster east-west:** Pod → Pod (via CNI, no HAProxy involvement)
- **Cross-cluster east-west:** Pod → HAProxy Kubernetes Ingress Controller → HAProxy Enterprise → HAProxy Kubernetes Ingress Controller → Pod

Path 3: Hybrid

HAProxy Fusion acts as the single orchestration layer for both approaches simultaneously. HAProxy Enterprise routes directly to pod IPs for latency-sensitive workloads, and routes to HAProxy Kubernetes Ingress Controller or HAProxy Unified Gateway for teams that prefer in-cluster

routing. Both paths share one control plane, one interface, and one set of security policies.

The distinction between the two approaches is architectural, not operational. Developers self-serve in both: through standard Ingress and Gateway API resources where an in-cluster controller manages routing, or through Kubernetes objects and annotations that Fusion ingests directly for direct-to-pod workloads.

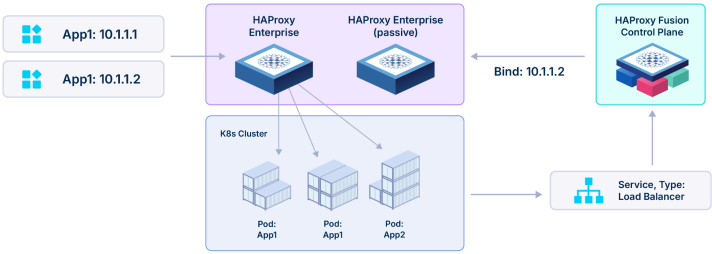
The tradeoff is visibility. This is a migration bridge. Where HAProxy Kubernetes Ingress Controller manages the inner edge, HAProxy Fusion's visibility stops at the cluster boundary. But for every workload migrated to direct-to-pod routing, that visibility gap closes.

- **Direct north-south:** Internet → HAProxy Enterprise → Pod
- **IC-managed north-south:** Internet → HAProxy Enterprise → HAProxy Kubernetes Ingress Controller → Pod
- **Direct east-west:** Pod → HAProxy Enterprise → Pod
- **IC-managed east-west:** Pod → HAProxy Kubernetes Ingress Controller → HAProxy Enterprise → HAProxy Kubernetes Ingress Controller → Pod

Migrate the workloads that need the extra hop gone first; the rest can stay on the in-cluster controller.

Reference architecture

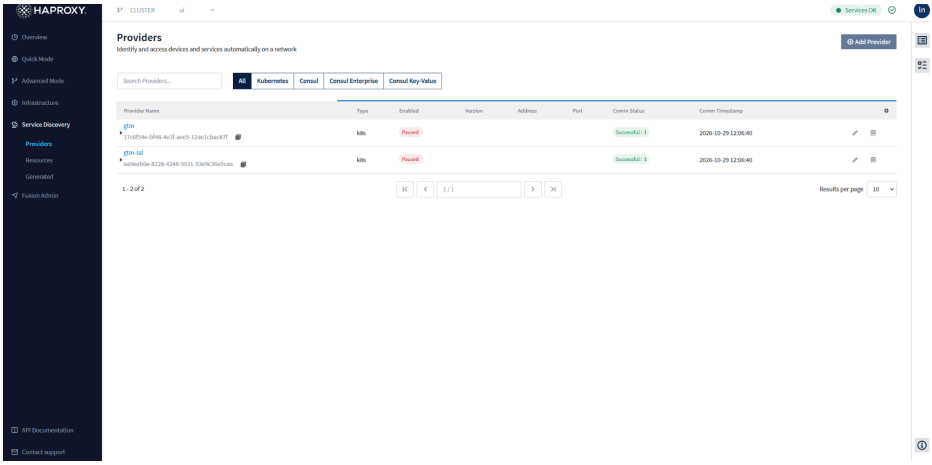
While HAProxy One supports all three paths, the reference architecture is the external gateway pattern. Both HAProxy Fusion and the HAProxy Enterprise gateway instances run entirely outside the Kubernetes clusters. The boundary handles routing, security, and observability, independent of what runs inside any individual cluster.



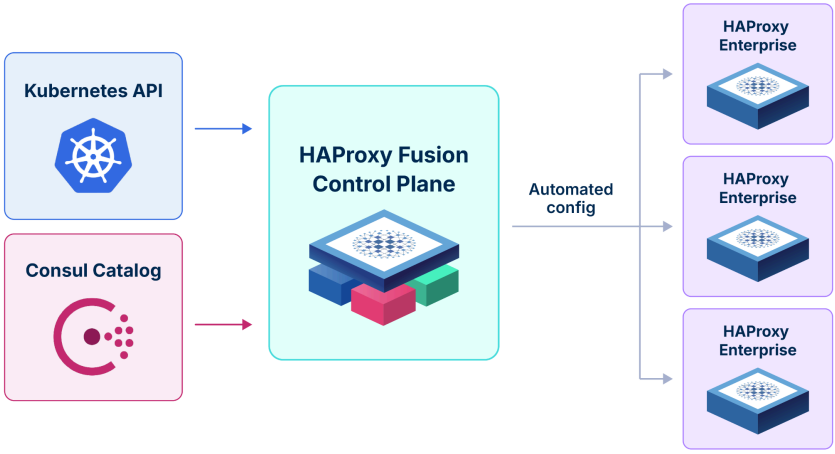
In practice, a single HAProxy Fusion instance (backed by redundant replicas) manages multiple HAProxy Enterprise nodes. HAProxy Fusion allocates addresses from an IPAM pool for the generated frontends, and can use the external IP or hostname of a Kubernetes LoadBalancer Service. Its service discovery keeps the backend server lines current with pod IPs as pods scale and move.

The result: traffic flows from the internet to HAProxy Enterprise, and directly from HAProxy Enterprise to pods. One hop. No ingress controller. No `kube-proxy` in the path. HAProxy Enterprise handles L7 routing, WAF, TLS, and load balancing at the boundary.

HAProxy Fusion connects to the Kubernetes API and supports conformant distribution: standard Kubernetes, OpenShift, EKS, AKS, and GKE. Adding a new cluster means connecting its API to HAProxy Fusion. The same routing policies, security profiles, and governance rules that apply to your existing clusters automatically apply to the new one.



This architecture also extends beyond Kubernetes. HAProxy Fusion can pull service data from Consul and Consul Enterprise, letting you route traffic to non-containerized workloads alongside Kubernetes services. You manage legacy VMs and bare-metal servers from the same control plane, so they participate in the same traffic distribution decisions as Kubernetes pods.



Why build it this way?

Each HAProxy One component does one thing well and stays out of the others' way. The result is an architecture that scales without a redesign.

In the reference architecture, HAProxy Fusion sits outside the clusters, so no single cluster's resources constrain it. Adding clusters doesn't change how HAProxy Fusion works; it just extends the scope of what it can see and manage. The control plane scales horizontally.

HAProxy Enterprise's position at the boundary means it handles traffic before it enters your application infrastructure. Security enforcement, load balancing, and routing decisions happen at the edge, where they're most effective and least expensive. Application pods handle application logic. They also don't handle threat mitigation.

The same configuration language, observability model, and governance policies apply whether you run two clusters or twenty. The approach doesn't change as the organization grows; the architecture scales with it.

That consistency is what **unified** means in practice: a platform where every component has a defined role. Together, they cover the full spectrum of what Kubernetes networking requires.

Chapter 7: Migrating without a big bang

Infrastructure migrations fail for the same reason: someone defines a change date. Everything before that date is planning and preparation. Everything after it is either success or incident response. The closer the date gets, the more pressure accumulates. Teams take shortcuts. They skip parallel validation. The migration becomes a bet rather than a process.

Our approach to migration starts from a different premise. There is no final cutover, no moment when the old system is fully switched off and the new one takes over. Instead, you run both approaches in parallel, for as long as you need, and move traffic incrementally, service by service, as you gain confidence.

The approach works because HAProxy Enterprise and HAProxy Fusion run alongside whatever already handles your traffic, so nothing gets torn out. You add a second path, validate it against real traffic, then shift services one at a time. Move them back if something looks wrong.

The migration becomes a process, not a bet. It ends when the last service has moved, not by declaration, but because there's nothing left to migrate. This chapter shows how that works in practice.

Why the coexistence model works

The coexistence model is possible because HAProxy Enterprise, managed by HAProxy Fusion, operates independently of whatever is currently

handling traffic for your clusters. Existing ingress controllers and routing rules can stay in place. HAProxy Fusion doesn't interfere with them.

HAProxy Fusion connects to the Kubernetes API and starts building its own picture of the infrastructure. Simultaneously, you configure HAProxy Enterprise as the entry point for whichever services you want to migrate first. Those services now have two paths into the cluster. The existing ingress controller path continues to work exactly as it always has. The new HAProxy Enterprise path routes directly to pods, bypassing both the ingress controller and `kube-proxy` entirely.

For teams coming from a standard ingress controller setup, this is where the latency improvement becomes measurable rather than theoretical. Removing `kube-proxy` from the path eliminates a layer of packet processing that existed solely because the cluster needed it, not because your application did.

You choose which path carries live traffic. In the beginning, your existing setup doesn't change. HAProxy Enterprise is ready, but it's not carrying production traffic yet. You use that window to validate by comparing responses, measuring latency, and confirming behavior. When you're satisfied, you shift one service to the HAProxy Enterprise path. The ingress controller keeps handling everything else.

The rest of the migration is repetition. Running both paths in parallel lets stakeholders compare live responses side by side and validate changes. If an application team spots an issue during validation, they can immediately roll back traffic without downtime.

Each migrated service increases stakeholder trust and reduces reliance on the old stack. When the last service has moved, you decommission the ingress controllers. Not because you had to by a certain date, but because you've verified they're no longer needed.

Service-by-service: the unit of migration

Moving services rather than clusters is what makes incremental migration practical. A service is a natural unit of ownership. It typically belongs to one team, serves one application function, and has a clear set of routing rules. However, services rarely exist in total isolation. Before you move a service, audit its upstream and downstream dependencies: shared session state, internal API call chains, custom header expectations. That way the transition doesn't affect coupled workloads.

The migration of a single service follows a consistent pattern. HAProxy Fusion discovers the service through Kubernetes service discovery. It's already watching your cluster, so this happens automatically. You configure the routing in HAProxy Fusion — the host and path rules that select this service, its health checks, and its traffic weights. HAProxy Enterprise receives that configuration and is ready to serve the traffic.

At this point, you have two working paths. You can send a small percentage of traffic to HAProxy Enterprise (5% or 10%) while the ingress controller handles the rest. HAProxy Enterprise sits at the edge, above the cluster, and applies traffic weights to incoming requests before they reach either path.

The split happens at the gateway, not at DNS, so changes take effect immediately instead of waiting for the time-to-live (TTL) to expire. You can adjust the percentage in real time as you watch the results. You'll see key metrics, including error rates, latency percentiles, and health check results. If everything checks out, you increase the percentage. If something looks wrong, you can move the percentage back to zero. The existing path keeps serving. The ingress controller was never taken out of service.

When you're confident the service behaves correctly through HAProxy Enterprise, you shift 100% of the traffic. The ingress controller's routing rule for that service becomes dead weight. Eventually, you can retire the ingress controller and classes, regaining resources and saving on costs.

To make the two paths concrete, here's how they compare in the midst of a service migration:

	Legacy ingress path	**HAProxy Enterprise path**
Traffic entry	Cloud LB or NodePort	HAProxy Enterprise (edge)
Routing logic	Ingress resources and annotations	HAProxy Fusion-managed policies
Network path	LB → Node → kube-proxy → Pod	LB → HAProxy Enterprise → Pod
Visibility	Per-cluster Kubernetes metrics	Real-time HAProxy Fusion dashboard across fleet

This process is reversible at every step. That reversibility is why things rarely go wrong. Teams that know they can roll back instantly don’t skip validation. Pick one service, move 5% of its traffic, and watch the error rate before you widen the split.

Replace, don't upgrade

The coexistence model rests on a broader principle: upgrading infrastructure and replacing it are different operations.

The traditional infrastructure upgrade cycle is painful. You identify a version that has reached end of life, plan a migration project, schedule downtime, and spend months coordinating a cutover. Eventually everyone is on the new version, and the cycle starts again. The older and more complex the infrastructure, the harder the upgrade.

The replace-don't-upgrade approach works differently. When a new version of HAProxy Enterprise is available, you add a new HAProxy Enterprise node running the latest version to your HAProxy Fusion-managed cluster. HAProxy Fusion routes a small amount of test traffic through it. You validate that it behaves correctly. You ramp traffic up. When the new node is handling full production load, you remove the old node. No downtime. No maintenance window. No coordination across teams.

This is possible because HAProxy Fusion is the single source of truth. It’s aware of everything happening in the infrastructure, which lets it manage configuration and traffic distribution centrally. The individual HAProxy Enterprise nodes are stateless with respect to routing logic. They receive their configuration from HAProxy Fusion and act on it. Swapping one node for another is a matter of registering the new node with HAProxy Fusion and deregistering the old one. HAProxy Fusion handles the traffic transition.

Applied to a migration from another platform, the same logic holds. You're not upgrading your existing infrastructure. You’re adding HAProxy One infrastructure alongside your existing platform, validating it at whatever

pace you need, and removing the old infrastructure when you're done. The existing system doesn't need to know anything about the migration until the moment it's decommissioned.

Building confidence before committing

The parallel validation period is where migrations succeed or fail. Teams that invest in it move quickly and confidently. Teams that skip it spend their time in incident response.

Validate these five things during the parallel period:

- **Response equivalence.** Send the same requests to both paths and compare the responses. This catches routing misconfigurations, header translation differences, and TLS behavior discrepancies before they affect production traffic. Using separate IP addresses for the two paths makes this comparison straightforward: you can test the HAProxy Enterprise path directly without touching the production traffic split.
- **Latency profile.** Measure how the HAProxy Enterprise path compares to the existing path. Direct-to-pod routing shows lower latency than the multi-hop ingress path. But if something is misconfigured (an extra TLS negotiation, a health check pointing at the wrong port) you'll see it in the latency data before it becomes a client-facing issue.
- **Health check behavior.** Confirm that HAProxy Enterprise's health checks accurately reflect the state of your pods. A health check that's too aggressive flips backends out of service under normal load spikes. One that's too lenient continues routing traffic to pods that are failing. Get this right during validation, not in production.

- **Security policy parity.** If your existing setup enforces security controls (rate limits, WAF rules, bot filtering), verify that HAProxy Enterprise's enforcement produces equivalent behavior. The goal is identical outcomes for the traffic patterns that matter.

- **Session persistence.** If a service relies on sticky sessions (routing a given client's requests consistently to the same backend pod) this is a common migration pitfall. When traffic is split 50/50 between the old ingress path and HAProxy Enterprise, a client whose session started on one path may end up on the other, losing their session state.

  Validate that your persistence configuration in HAProxy Enterprise matches the behavior of your existing setup before increasing the traffic split significantly. Where possible, test this explicitly rather than assuming equivalent configuration produces equivalent behavior.

The parallel period has a natural endpoint: the moment when you've seen enough normal traffic through HAProxy Enterprise, across enough varied conditions, that you trust it. That's a judgment call that belongs to the team closest to the service. There's no minimum duration and no checklist that concludes it. You know when you're done because you're not learning anything new from the comparison.

Practical groundwork

Before the first service moves, a little preparation pays off:

1. **Document what you have.** The services being migrated, the routing rules that govern them, the annotations and configurations that are doing work in your current setup. This is

much more about knowing what you're translating than about creating paperwork. Undocumented routing logic is a common source of surprises during migration. If you can't describe what the current ingress controller is doing for a given service, you can't verify that HAProxy Enterprise is doing the equivalent.

2. **Remove what you don't need.** Migrations often uncover dead configuration. You'll find routing rules for services that were decommissioned months ago, annotations that no one remembers adding, and legacy health check endpoints that haven't been valid in years. These accumulate in long-running ingress configurations. Removing them before migration keeps the scope clean and prevents you from accidentally carrying technical debt into the new setup.

3. **Freeze configuration during the transition.** Once a service's migration is in progress, don't change its routing logic at the same time. If you're migrating and modifying simultaneously, you don't have a clean baseline for comparison. Problems become ambiguous: is this a migration issue or a configuration change issue? Finish the migration before making the change.

4. **Consider shadow mirroring for high-risk services.** For services where even brief anomalies in the parallel period would be unacceptable, shadow mirroring offers a lower-risk validation approach. HAProxy Enterprise can mirror a copy of production traffic to the new path without sending the response back to the client. The client gets the response from the existing ingress path as normal, while HAProxy Enterprise processes the same request in parallel.

   You get real production traffic patterns, real response data to compare, and no client-visible impact. This is particularly useful for

services with complex routing logic or stateful behavior where synthetic test traffic is unlikely to catch all edge cases.

5. **Define your rollback signal in advance.** Before shifting any production traffic to HAProxy Enterprise, decide what you would need to see to immediately move traffic back. An error rate above a certain threshold? Latency above a certain percentile? A specific type of error in the logs? Having this defined in advance means rollback is a clear decision, not a panicked judgment call during an incident.

When you're done

A migration that uses this model doesn't have a dramatic conclusion. There's no go-live moment and no launch announcement. At some point you notice that the last service migrated successfully, that all traffic is flowing through HAProxy Enterprise and HAProxy Fusion, and that your ingress controllers are sitting idle. You're safe to decommission them.

The migration is done. Not because of a date on a calendar, but because there's nothing left to migrate and nothing to roll back to. Your confidence in the new setup accumulated incrementally, service by service, over however long you needed. The risk was managed at each step rather than concentrated at a single cutover point.

That's the goal. Not a faster migration, but a safer one that results in an infrastructure you've thoroughly validated rather than one you've bet on.

Conclusion: What Kubernetes networking looks like at scale

You started this book with five problems. Latency from extra hops. External access that fragments the moment multiple clusters are involved. Routing tables that go stale because pods move faster than any manual process can track. Clusters that operate as isolated silos, leaving multi-cluster coordination an operational burden. Inefficient security that runs inside the cluster, spending application resources on traffic that should have been stopped at the boundary.

These challenges are a result of Kubernetes' success: as enterprise footprints expand, organizations simply outgrow the original networking design. They are the natural outcome of an architectural assumption that networking intelligence belongs inside a single cluster. While that model made early deployments easy to reason about, it creates an operational ceiling as production scale demands more.

The argument this book has made, across architecture, real-world scenarios, product specifics, and migration guidance, is that a different assumption produces a different outcome. When one control plane watches every cluster you run, the ceiling disappears.

The improvement is concrete. Failover operates at the pod level rather than the cluster level. Load balancing decisions are made with full fleet visibility. Deployment strategies span cluster boundaries. Governance scales with your team count rather than breaking under it.

Where the industry is heading

The direction of travel in Kubernetes networking is not ambiguous. The move away from sidecar proxies reflects a growing consensus that the per-pod, per-cluster model carries more operational overhead than it returns in value.

The organizations furthest along in this transition share a few characteristics. Most run multiple clusters. Their platform teams treat networking as infrastructure rather than an application concern, and they have reached the point where the question is no longer 'how do we get traffic into the cluster?' but 'how do we manage traffic across a fleet?' No Kubernetes-native tool answers that second question as well as an external control plane designed from the start to see across boundaries.

The Universal Mesh is where this trajectory leads: a networking layer that is consistent across every environment you run, that discovers and routes to any object your infrastructure exposes, that enforces security at the boundary where it costs least, and that gives your teams a single interface for observability, governance, and configuration across the whole thing.

What you can do next

If you've read this far, you have a clearer picture of what Kubernetes networking looks like when it's built around the right architectural premise. The practical question is where to start.

For most organizations, the answer is not to migrate everything at once. It's to identify the specific problem that is costing you the most right now. For many, that's federation coordination: the inability to load balance or fail over across clusters without manual intervention. For others, it's

Ingress NGINX end-of-life forcing a migration decision that's better made thoughtfully than urgently. For others still, it's the security story: WAF and bot management that runs inside the cluster, consuming application resources to handle threats that should have been stopped at the door.

Start with the problem. The architecture follows.

HAProxy Fusion connects to clusters, discovers services and objects, and gives HAProxy Enterprise what it needs to route traffic directly to pods, across clusters, with enterprise-grade security at the boundary, from a single control plane. HAProxy Kubernetes Ingress Controller and HAProxy Unified Gateway serve teams that want Kubernetes-native routing, with a structured path to Gateway API and, when they're ready, to the external gateway model.

The migration doesn't require a cutover date. It requires a starting point. Start with one cluster and one problem. We'll help you find the starting point.

Appendix

Glossary of terms

Networking has a lot of vocabulary, and this book uses its share of it. To help you keep the architectural concepts straight, this glossary is grouped by topic rather than alphabetically.

Kubernetes terms

The components and concepts native to the Kubernetes ecosystem.

- **Container:** A container is the unit that actually runs your application. It packages your application with everything it needs to run: code, runtime, libraries, dependencies. It bundles them into a single standardized unit. The point is portability. The same container runs the same way on a developer's laptop, a test server, or a production node.
- **Pod:** The smallest deployable unit in Kubernetes. A pod wraps one or more containers that work together and gives them a shared network address and storage, so they behave as a single unit. Most pods run one container. Pods are ephemeral: Kubernetes creates, destroys, and moves them as demand changes.
- **Node:** The physical bare-metal server or virtual machine that provides the computing power your pods run on.
- **Cluster:** A group of machines, called nodes, that work together to run your containerized applications. If a pod is a single worker, the cluster is the factory floor. Most organizations run more than one, spread across regions or cloud providers.
- **YAML:** A human-readable data serialization language. In Kubernetes, you write YAML files (often called manifests) to declare exactly how your applications and routing rules should be

deployed. YAML is the "infrastructure as code" layer that Kubernetes reads.

- **Service:** A stable network address in front of a group of pods running the same application. Pod IPs change constantly, so other services connect to the Service instead, and Kubernetes routes to whichever pods are healthy. A Service is what makes pod-to-pod communication reliable inside the cluster, and it is the abstraction the in-cluster path is limited to.
- **Endpoint:** A specific destination a router can send traffic to: a pod IP, a node IP, or a host. Service discovery keeps the list of endpoints current as pods move. The external gateway reads endpoints straight from the Kubernetes API.
- **Ingress:** The original Kubernetes resource for declaring how external traffic reaches a Service. An Ingress object is a routing rule. On its own, it does nothing.
- **Ingress controller:** The proxy that reads Ingress and Gateway API resources and turns them into working routing logic. It runs inside the cluster, so it is part of the cluster's internal routing chain. See **In-cluster proxy** for how that role fits into the architecture.
- **Gateway API:** The modern, extensible Kubernetes standard for routing traffic. It replaces Ingress with more expressive, role-oriented resources, so platform teams and application teams can each manage the part of the routing they own.
- **kube-proxy:** The network agent on every node that maintains the rules for routing traffic to Services. It is the Kubernetes default for internal routing and one of the layers the in-cluster path depends on. Architectures that route around the cluster's internal chain bypass it.

- **Flat network model:** A single, shared IP address space in which every pod can reach every other pod directly, regardless of which node it runs on. The flat network is private by default: nothing outside the cluster can reach a pod until you open a way in.
- **Namespace:** A way to divide a single cluster into separate scopes for teams and applications. Namespaces and their permissions decide what each team can change. That is the governance boundary a control plane inherits rather than invents.
- **Kubernetes-native:** Describes the objects and workflows that belong to Kubernetes itself: Ingress and Gateway API resources, CRDs, the manifests your teams already write. Applied to a product, it means the component runs inside the cluster and uses Kubernetes as its control plane. HAProxy Enterprise Kubernetes Ingress Controller and HAProxy Unified Gateway are Kubernetes-native.
- **Kubernetes-integrated**: A component that reads the Kubernetes API and works with native objects, but owns its own control plane rather than using Kubernetes as one. HAProxy Fusion is Kubernetes-integrated.

Core networking concepts

The operational challenges, routing flows, and mechanisms this book discusses.

- **Mesh tax:** The cost of running a proxy in every pod. Each sidecar consumes CPU and memory that would otherwise go to your application, adds latency to every internal request, and leaves behind a tangle of per-pod routing rules. These costs scale with your pod count. The more you grow, the more you pay.

- **Traffic sandwich:** An architecture in which traffic hops through several redundant networking tiers before it reaches your application. A request might pass through a cloud load balancer, an external firewall, and an internal ingress controller in sequence. Each tier adds latency and another place for a request to fail.
- **Sidecarless service mesh:** Securing internal east-west traffic by routing it from external gateway to external gateway, with no proxy in the pod. The gateway already sits at the boundary and already terminates connections, so it can enforce policy there instead of in every pod.
- **Direct-to-pod routing:** A network path where a gateway uses service discovery to send traffic straight to a pod's IP address, skipping the cluster's internal proxy layers. Traffic arrives with fewer hops, and the gateway can address a pod that the Service abstraction cannot.
- **North-south traffic:** Traffic flowing into and out of your infrastructure. A user on the internet reaching your application is north-south traffic.
- **East-west traffic:** Traffic flowing between services inside your infrastructure. A frontend service talking to a backend database is east-west traffic.
- **mTLS termination point:** In a sidecarless architecture, the external gateway where mutual TLS is verified and re-established for traffic crossing clusters or regions. Terminating mTLS at the gateway takes the encryption work off the application pod, which no longer has to run a sidecar to participate.
- **Black box effect:** What happens to a cluster when an external gateway hands all internal L7 routing and pod selection to the ingress controller. The gateway sees the controller's entry point and nothing behind it, so a request that crosses clusters passes

through two controllers with nothing tying the two halves together.

- **Configuration drift:** The gap between your load balancer's routing rules and the real-time state of your pods. Pods move faster than a manual process can track, so rules go stale and traffic gets dropped. Also called service discovery drift. A control plane that watches the Kubernetes API closes the gap by keeping its view current.
- **Coalescing interval:** The window a control plane waits before publishing a configuration update. Instead of pushing a new configuration for every individual pod change, it aggregates the events into one update. This prevents configuration reload storms and the CPU spikes that come with them.

Architecture terms

The Universal Mesh and the three paths you can take to it.

- **Universal Mesh:** An architectural approach that replaces per-pod sidecar proxies with gateways at the network boundary. It unifies traffic management and security across Kubernetes, virtual machines, and bare metal. It is a state you arrive at, not a product you install.
- **Control plane:** The layer that manages configuration, policies, and service discovery across your gateways. Chapter 3 calls it the orchestration layer. It watches every cluster you run, so routing decisions can be made with full fleet visibility, and it never processes application traffic itself. Kubernetes has its own control plane (the API server, scheduler, and etcd), which is a different thing with the same name.

- **Data plane:** The gateways that handle L4/L7 application traffic and security enforcement: routing, TLS termination, health checks. Chapter 3 calls it the gateway layer. In a Universal Mesh, the data plane sits at the boundaries rather than inside every pod.
- **Boundary gateway:** A high-performance component placed at the perimeter of an environment (a cluster, a virtual private cloud, a region) to handle traffic entering, leaving, or crossing between zones. A single boundary gateway can serve both the inner and outer edge, replacing a proxy in every pod.
- **External gateway:** A gateway that makes its routing decisions outside the cluster's internal routing chain. It discovers endpoints through the Kubernetes API and routes to them directly: pod IPs, cluster IPs, nodes, or hosts. Traffic never traverses the ingress controller or kube-proxy. This is the pattern the Universal Mesh is built on.
- **In-cluster proxy:** The ingress controller's role in the traffic path. It runs inside the cluster and is the cluster's internal routing chain, which means it adds a hop between the gateway and your application. The in-cluster proxy pattern is the right choice when a team wants routing to live inside the cluster.
- **Hybrid pattern:** Running the external gateway and the in-cluster proxy side by side. Local routing stays with the developers who own it, while security and observability stay with the platform team. Teams migrate the workloads they can and leave the rest in place, so the migration has no cutover date.
- **Network boundary:** The perimeter of your environment where traffic enters and exits. A Universal Mesh splits the boundary into layers (the inner and outer edge) so each layer handles the traffic it is built for, instead of funneling everything through one chokepoint.

- **Outer edge:** The global or regional entry point into your infrastructure. Gateways here balance load across regions and clusters and filter malicious traffic before it reaches your environments.
- **Inner edge:** The localized boundary that manages internal traffic: routing directly to your applications and handling communication into and between your Kubernetes clusters.
- **Sidecar proxy:** A small proxy injected into every application pod to intercept and route its traffic. A service mesh uses sidecars to manage pod-to-pod communication. The cost is real. Each sidecar consumes CPU and memory that would otherwise go to your application, and the resulting per-pod routing rules are hard to debug.
- **Service mesh:** A system for managing how pods talk to each other, traditionally by injecting a sidecar proxy into every pod. It solves internal traffic management, and it is the approach the Universal Mesh moves away from.
- **Fleet:** The full set of gateways and clusters you manage. Fleet visibility means one control plane can see the health and capacity of every backend across every cluster at once. That is what makes pod-level failover and cross-cluster load balancing possible.

Traffic and security terms

How traffic moves, and where you stop the traffic you don't want.

- **TLS termination:** Ending an encrypted connection at the gateway, then handling the request from there. Terminating TLS at the boundary keeps certificate management and the decryption cost out of your application pods.

- **WAF:** Web application firewall. It inspects HTTP requests and blocks attacks (injection, cross-site scripting, and the rest) before they reach your application. Running the WAF at the boundary means threats are rejected before they consume application resources.
- **Bot management:** Detecting and handling automated traffic separately from human traffic. That covers scrapers, credential stuffing, and inventory abuse. At the boundary, bot management classifies and filters requests before they reach your pods.
- **Threat scrubbing:** Filtering malicious traffic out of the request stream at the edge, so only legitimate requests reach your infrastructure. The outer edge is where scrubbing happens first.

HAProxy One components

The products that bring the sidecarless Universal Mesh architecture to life.

- **HAProxy One:** The application delivery and security platform that the rest of the components belong to. It separates the intelligence from the execution: HAProxy Fusion decides, HAProxy Enterprise enforces, and the components below place each capability at the layer where it works best.
- **HAProxy Fusion:** The Kubernetes-integrated control plane. It connects to your Kubernetes clusters, watches the Kubernetes API continuously, discovers services and objects, and translates that into routing configuration for the gateways it manages. It handles multi-cluster federation and gives one view of the whole fleet.
- **HAProxy Enterprise:** The data plane. It runs as the boundary gateway, executing L4 and L7 routing, WAF policies, and direct-to-pod routing. It usually sits outside the cluster at the edge of your infrastructure, and it can also run inside a cluster.

- **HAProxy Kubernetes Ingress Controller:** The Kubernetes-native option for teams that want routing to live inside the cluster. It translates standard YAML manifests into routing configuration for developer-driven workloads, and it is the product for the in-cluster proxy path.
- **HAProxy Unified Gateway:** The bridge to Gateway API. A single controller processes both existing Ingress manifests and new Gateway API resources concurrently, so teams migrate route by route instead of choosing a cutover date. It implements Gateway API role-based access control, and it supports HTTPRoute and TLSRoute today.
- **HAProxy Edge:** An optional SaaS application delivery network (ADN) at the outermost perimeter. It provides globally distributed, fully managed DDoS protection, threat scrubbing, and delivery services in front of your HAProxy Enterprise gateways. The threat intelligence it collects also trains the machine-learning models behind the WAF and bot management capabilities in HAProxy Enterprise, whether or not you deploy HAProxy Edge directly.

Additional resources

More on HAProxy and Kubernetes

To further deepen your knowledge, explore select webinars and blog posts focusing on advanced traffic engineering with HAProxy and Kubernetes.

- **Beyond fragmented networking: achieving end-to-end security and observability with Universal Mesh.** Stop managing silos and start scaling. If your team is managing sprawling proxy deployments across multiple clouds, this session will show you a cleaner path forward.
- **External load balancing and multi-cluster routing for Kubernetes.** In this session, we show how to simplify deploying Kubernetes in complex scenarios using HAProxy Fusion and HAProxy Enterprise for intelligent external load balancing and multi-cluster routing.
- **Life after Ingress NGINX: how to unify Kubernetes traffic in one stack.** This session was standing room only at KubeCon EU, so we're sharing it for everyone who couldn't make it. In under an hour, you'll get a clear picture of the Kubernetes traffic landscape post-Ingress NGINX, and a practical roadmap for what comes next.
- **Life after Ingress NGINX: a zero-stress migration guide to HAProxy.** Watch our webinar where we break down exactly how to move from Ingress NGINX to the HAProxy Kubernetes Ingress Controller—and why this switch gives you immediate performance wins and a future-proof path to the Gateway API.
- **Universal Mesh in action: how PayPal solved multi-cloud complexity with HAProxy.** PayPal's Meridian is a powerful real-world use case of HAProxy's Universal Mesh succeeding at enterprise scale.



Visit us at https://www.haproxy.com