ISOVALENT

Adopting Network Policies in Highly Secure Environments

Speaker: **Raymond de Jong**

Agenda

- Introduction
- Strategies for Designing Network Policies
- Cilium Features that Matter
- Observability as the Network Policy Superpower
- Demo

Introduction

- Open Source Projects

- Company behind Cilium
- Provides Cilium Enterprise

Strategies for Designing Network Policies

Challenges when Enterprises Adopting Network Policy

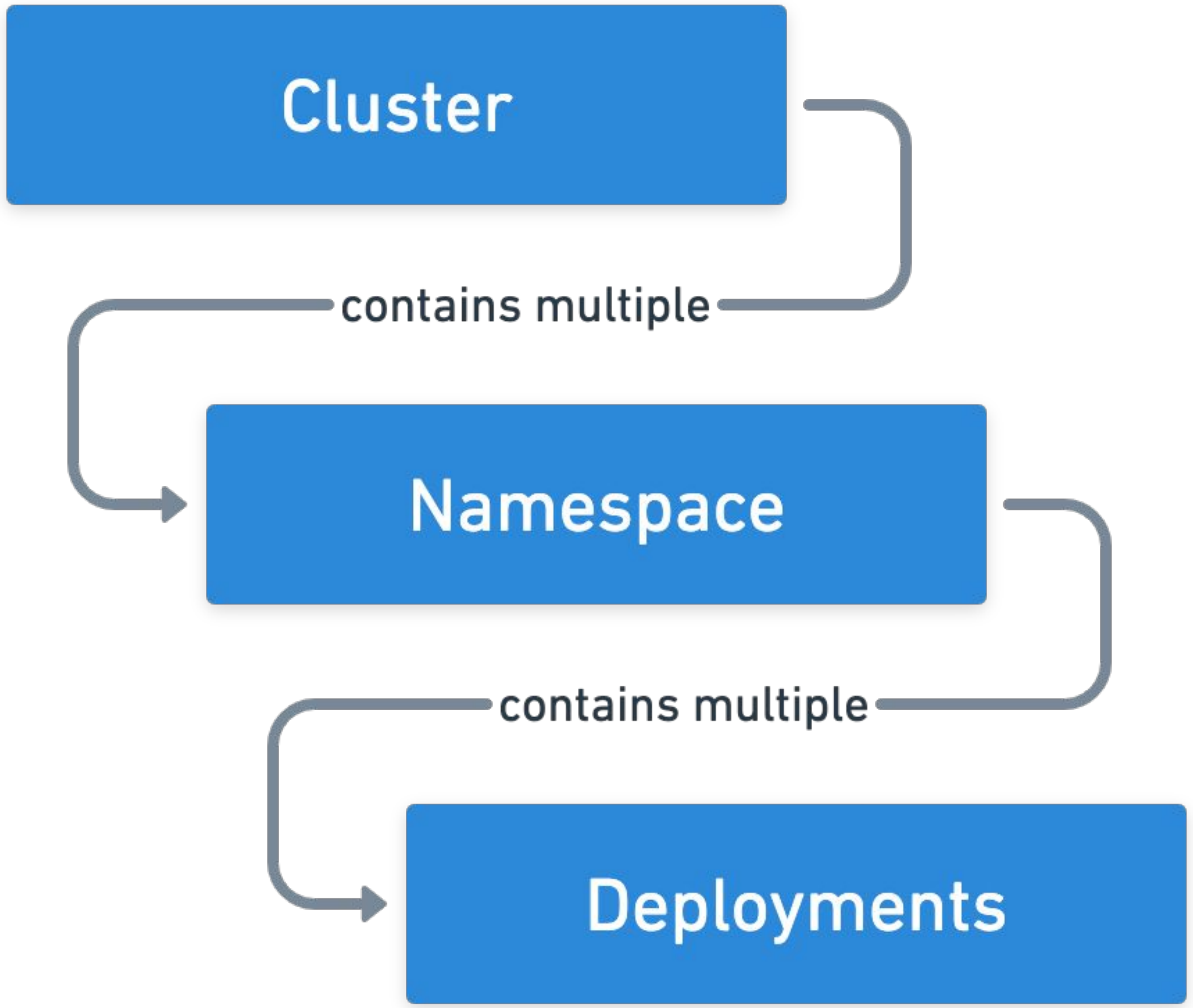
- Where to start? What policies reduce risk with least amount of friction?
- How to troubleshoot Network Policies?
- How to stay up-to-date with Network Policies while applications evolve?
- How to prevent application teams from simply allowing everything?
- How can security teams prove that default deny policies are enforced?
- How can security teams be alerted on denied connections?

There is no “Easy Road” for Adopting Network Policy

- Default Deny Approach:
  - Each service-to-service communication must be explicitly allowed by Network Policy.
  - High chance of misconfiguration which will result in application unavailability and adoption friction.
- Better Approach: Focus on Risk Reduction
  - Define metrics for Risk Exposure.
  - Focus on the most security sensitive namespaces first.
  - Leverage network observability to identify which network policy patterns easily reduce risk with minimal friction.
  - Once tooling and policy management workflows are proven, use the metric to iteratively expand the covered workloads

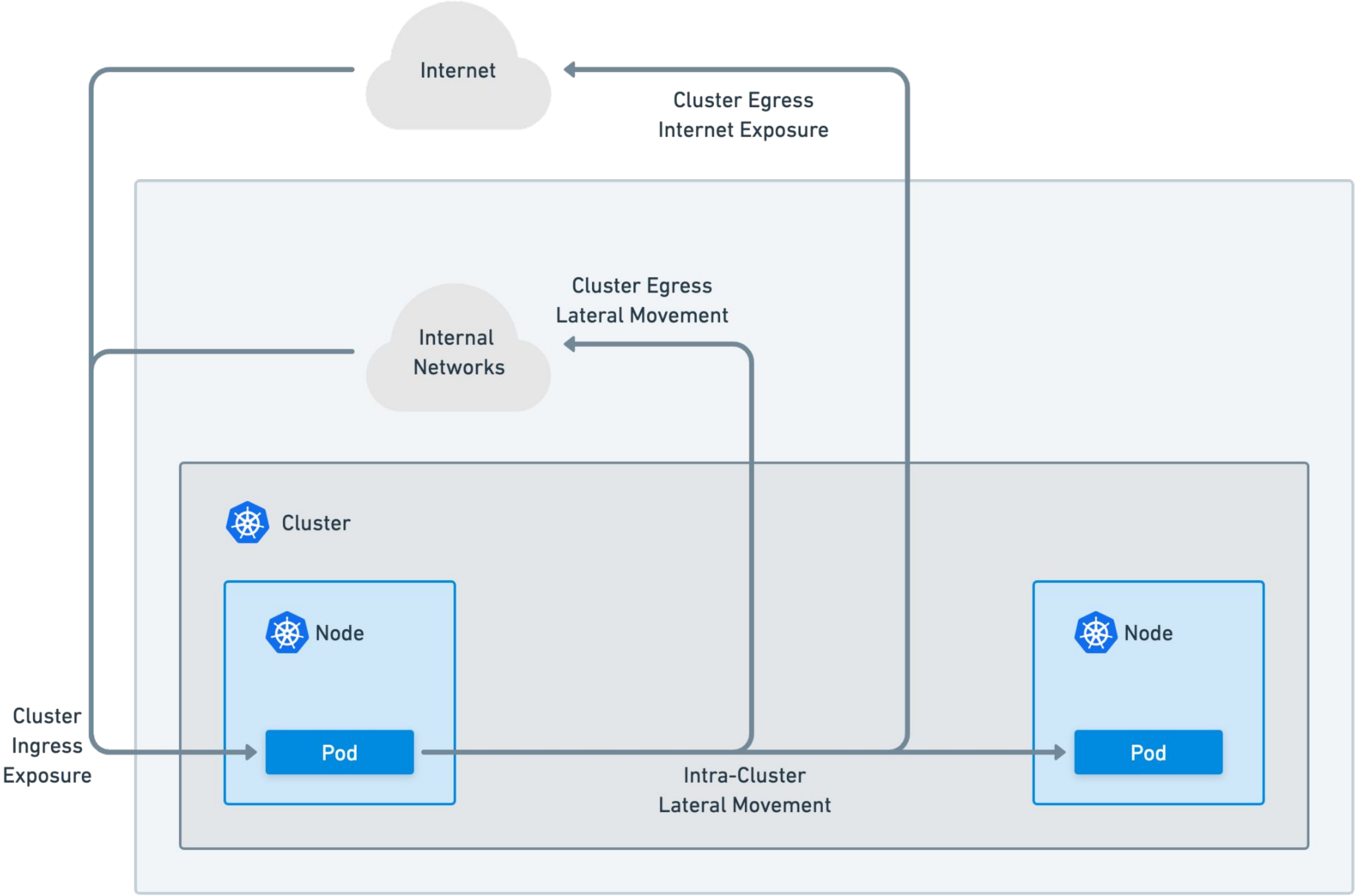
Application Teams & Multi-tenancy

- A Kubernetes cluster is often used by multiple teams, each managing one or more services.
- Each team has one or more namespaces to run their applications.
- Applying network policy at namespace level is a common first step in enabling multi-tenancy.



Note: Not always 1-to-1 team-to-namespace. Team may be a namespace-label, or pod label within a larger namespace, but same concept applies.

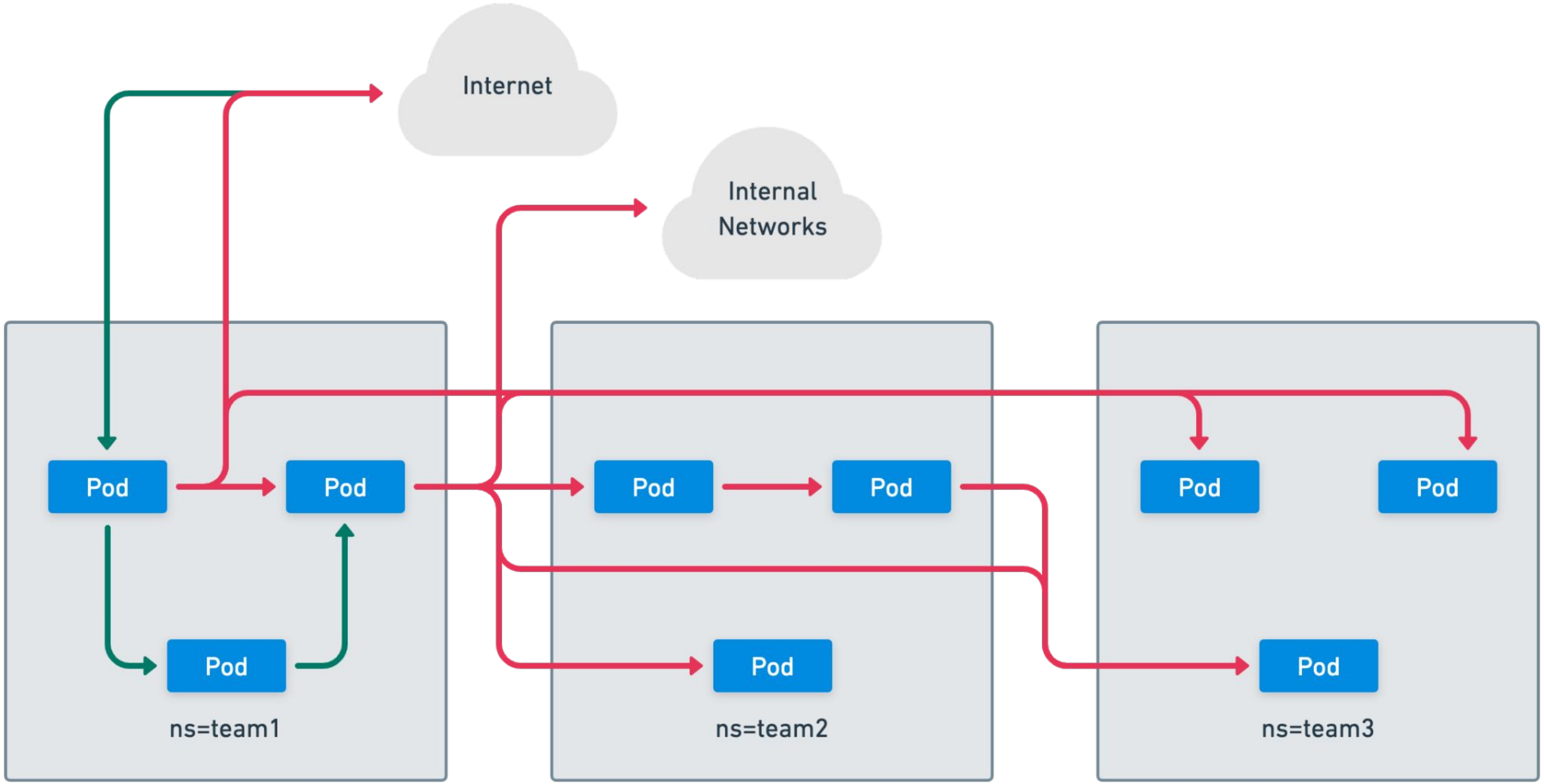
Key Types of Kubernetes Network Risk Exposure



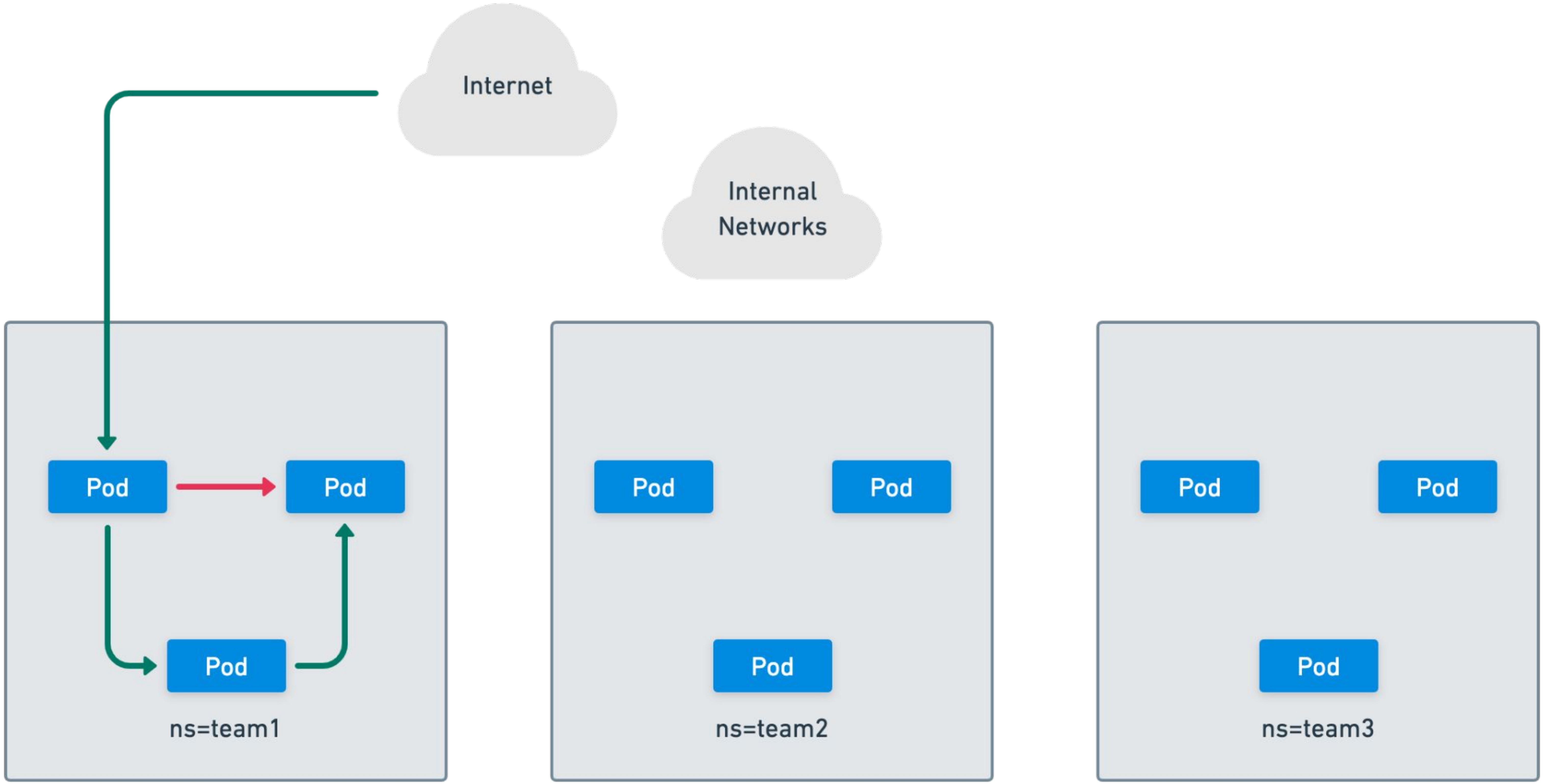
Minimizing Risk: Reduce Unused Access

Primary Goal: Reduce unused network access that creates exposure without being required for the application to function properly.

Risk Reduction Example - Wide blast radius



Risk Reduction Example - Limited blast radius



Measuring Risk: Quantify Unused Access

Measuring Progress: With strong network observability, the “mismatch” between what is allowed by network policy (or the lack of network policy) is what you want to focus on minimizing.

Examples:

- **kube-dns** being exposed to every workload in the cluster is intended, not a risk, but a private tenant service being reachable by all pods is a risk.
- A frontend service on port 443 that is supposed to be public should be reachable from everywhere within and outside the cluster, but the redis server it uses to cache database results should not be.

Measuring Risk: Prioritization

Concrete Measurements to guide prioritization

- # of services reachable via Ingress/Gateway API
- # of services reachable from other namespaces
- # of services with access to External Networks or Internet

The Cost of “Overfitting”

- Strong network observability tools enables you to translate each observed flow into a rule.
- However, simplified approaches to translating traffic flow to rule tend to be cumbersome and brittle.
- Why?:
  - IPs for services outside the cluster can change or are opaque.
  - App dependencies or shared services: kube-dns, external logging, vaults servers are required by all apps. Creating pair-wise rules leads to friction and bloat.
  - Communication within a single app/team. Complex and frequent change. Leads to increased number of rules which change frequently.

Initial “Coarse-Grained” Per-Namespace Strategy

Pattern to avoid “overfitting”

- Allow all ingress/egress communication within a namespace.
- Use Hubble Observability data to identify and permit:
  - Which (service + port) within the namespace are “public services”:
    - Allow from cluster-only.
    - Allow from “world” (type LB/NP) or ingress.
  - Egress Access to:
    - Other services in the cluster (optional, limit ports).
    - Other services in the external private network (optional, limit ports).
    - Other services on the Internet (optional, limit ports).
- Coarse-grained nature of policies mean that policies need only change rarely when an app changes a core aspect of it’s communication pattern.

Key Pattern: Baseline Policies vs. Per-Namespace Policies

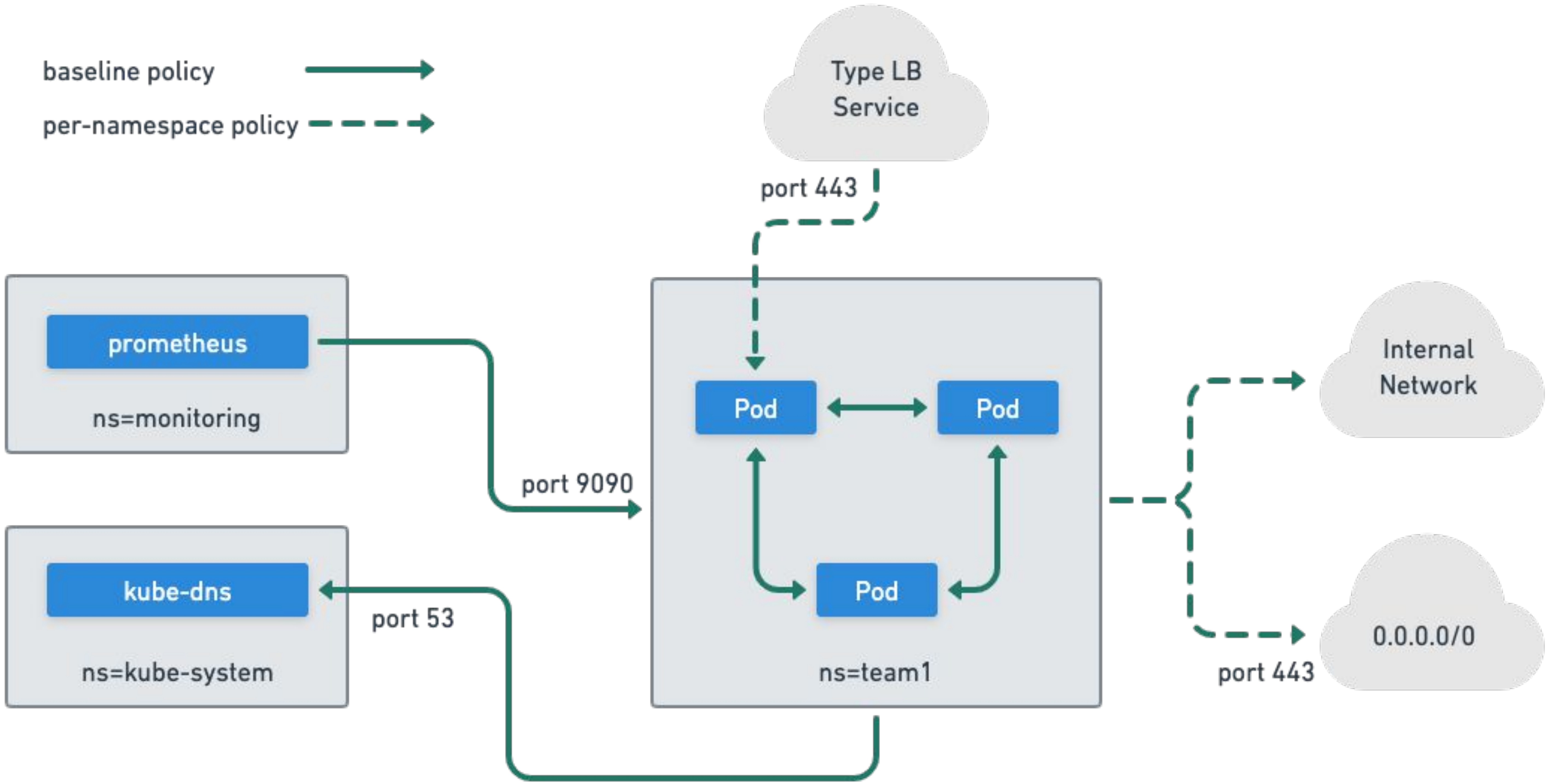
Global Baseline Policies

- Default deny ingress/egress.
- Allow all ingress/egress within namespace.
- Egress to “public services”:
  - cluster-wide shared services (e.g. kube-dns, prometheus).
  - external shared CIDR/DNS (logging, monitoring, vault, etc.)
- Often implemented by: CiliumClusterWideNetworkPolicies.

Per-Namespace Policies

- Per-service ingress (limited by port):
  - Exposed within cluster via ClusterIP service.
  - Exposed externally via Ingress/Gateway API or LoadBalancer/NodePort.
- Namespace-specific egress:
  - No access.
  - Egress to specific CIDR/DNS + port
  - Unrestricted access on specific ports (temporary fallback).

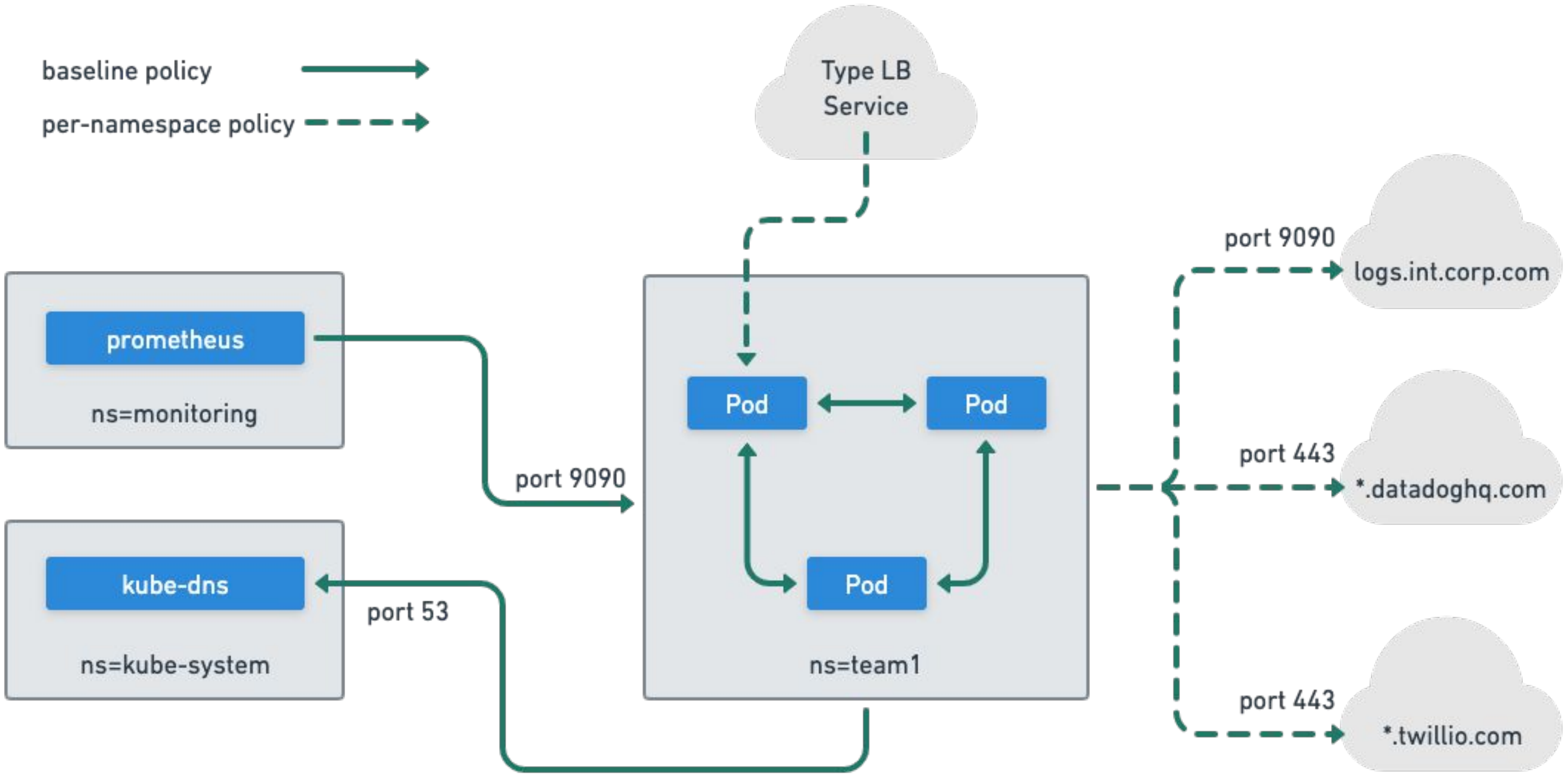
Example Global Baseline + Initial Namespace Policies



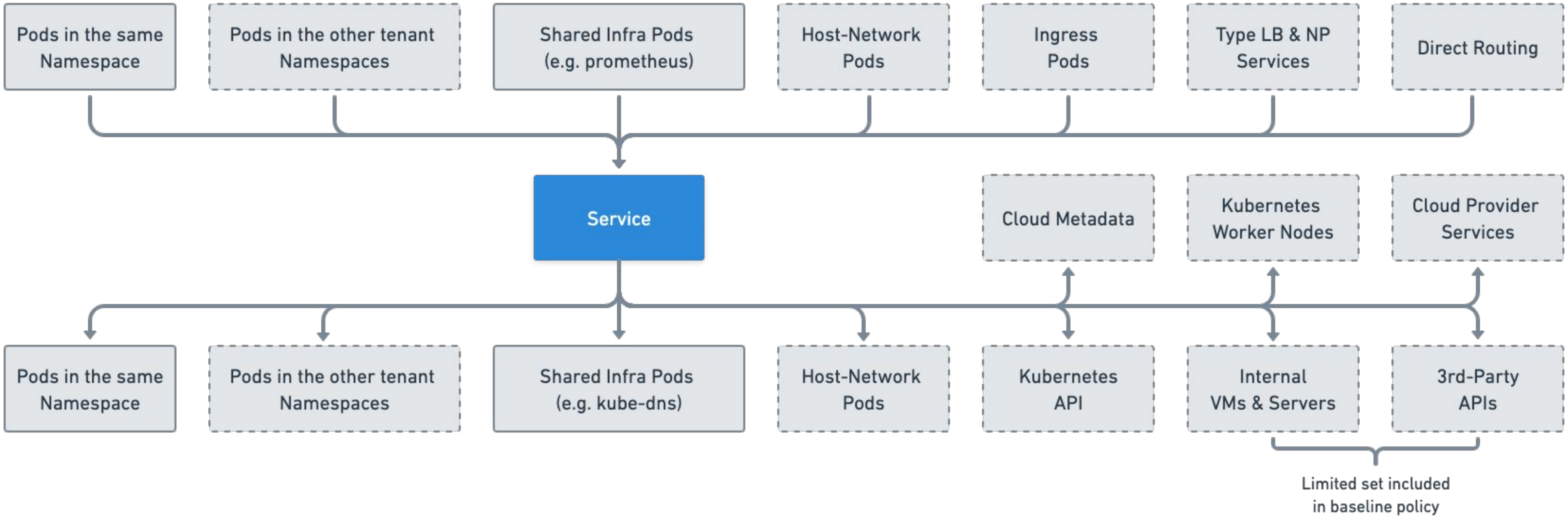
Transitioning from “Coarse” to “Fine-Grained” Policies

- Prioritize namespaces based on:
  - Most security-sensitive applications.
  - Measurement of exposure vs. used connectivity.
- Transition for egress outside the cluster:
  - Access to all private network → Access to specific FQDNs or smaller CIDRs (with port).
  - Access to Internet → Access to specific FQDNs or small CIDRs (with port)
- Transitions within the cluster:
  - Shift rules that allow all to/from cluster to allowing to/from specific namespaces or services.

Example Baseline + Fine-grained Per Namespace Policies

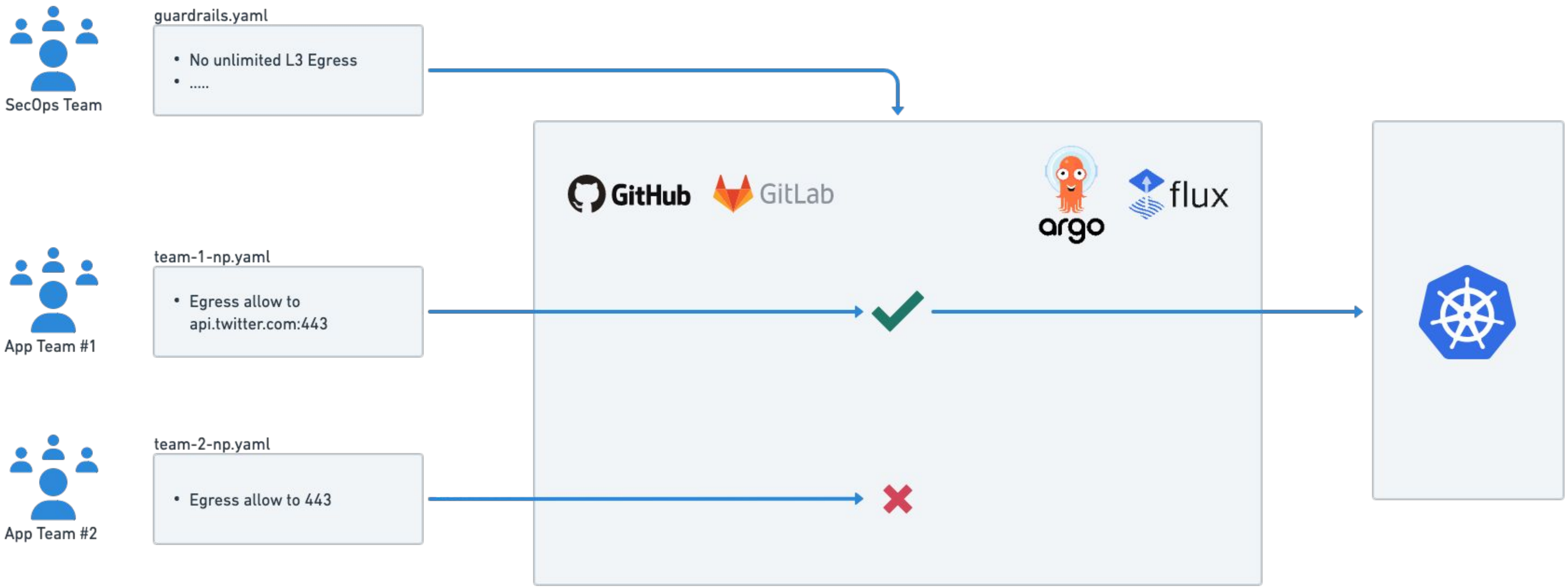


Key Policy Sources & Destinations to Consider



Network Policy Guardrails

Maintain control while granting application teams self-service management of Network Policy.



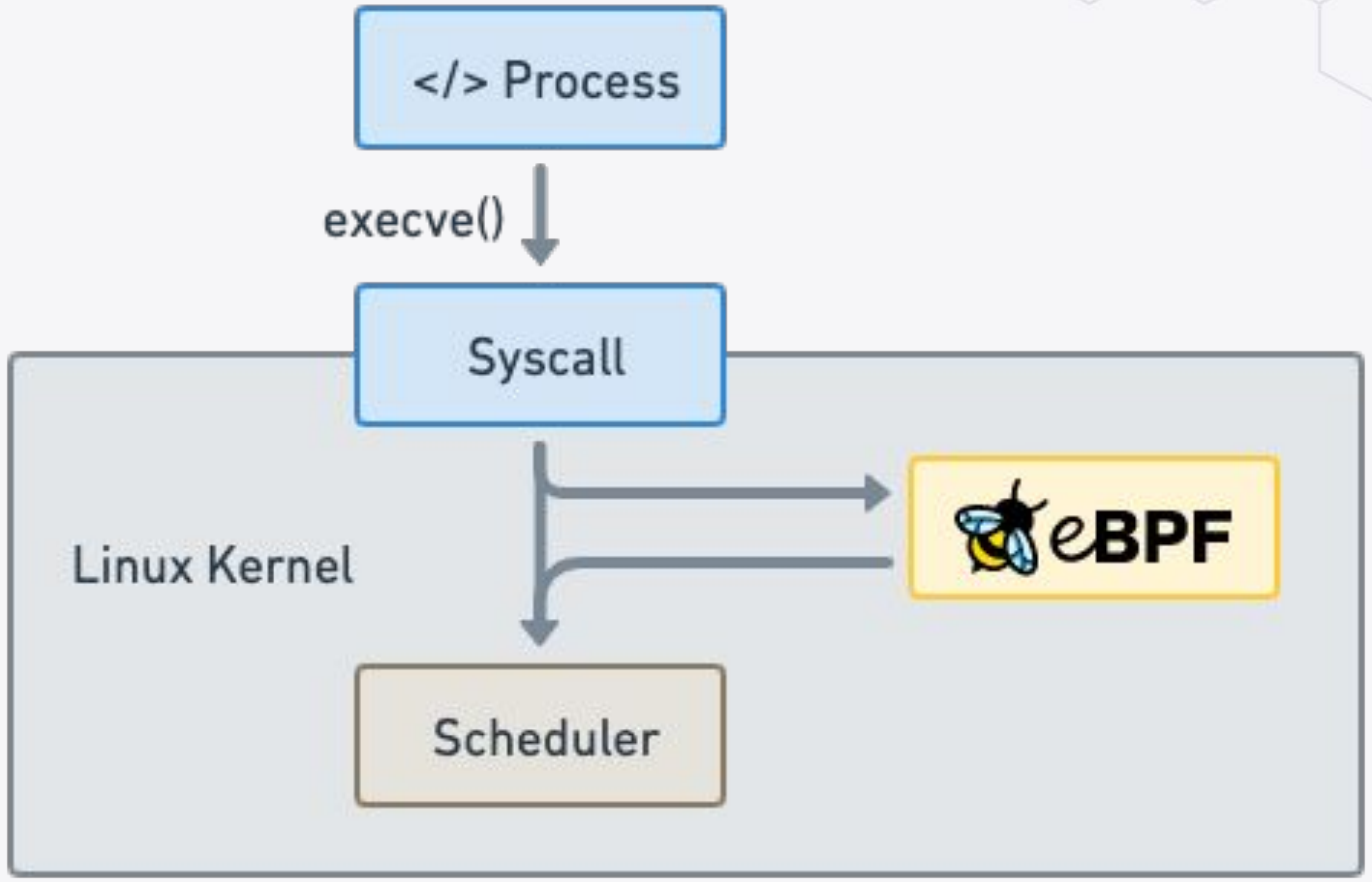
Cilium Features that Matter

eBPF

eBPF

Makes the Linux kernel programmable in a secure and efficient way.

“What JavaScript is to the browser, eBPF is to the Linux Kernel”

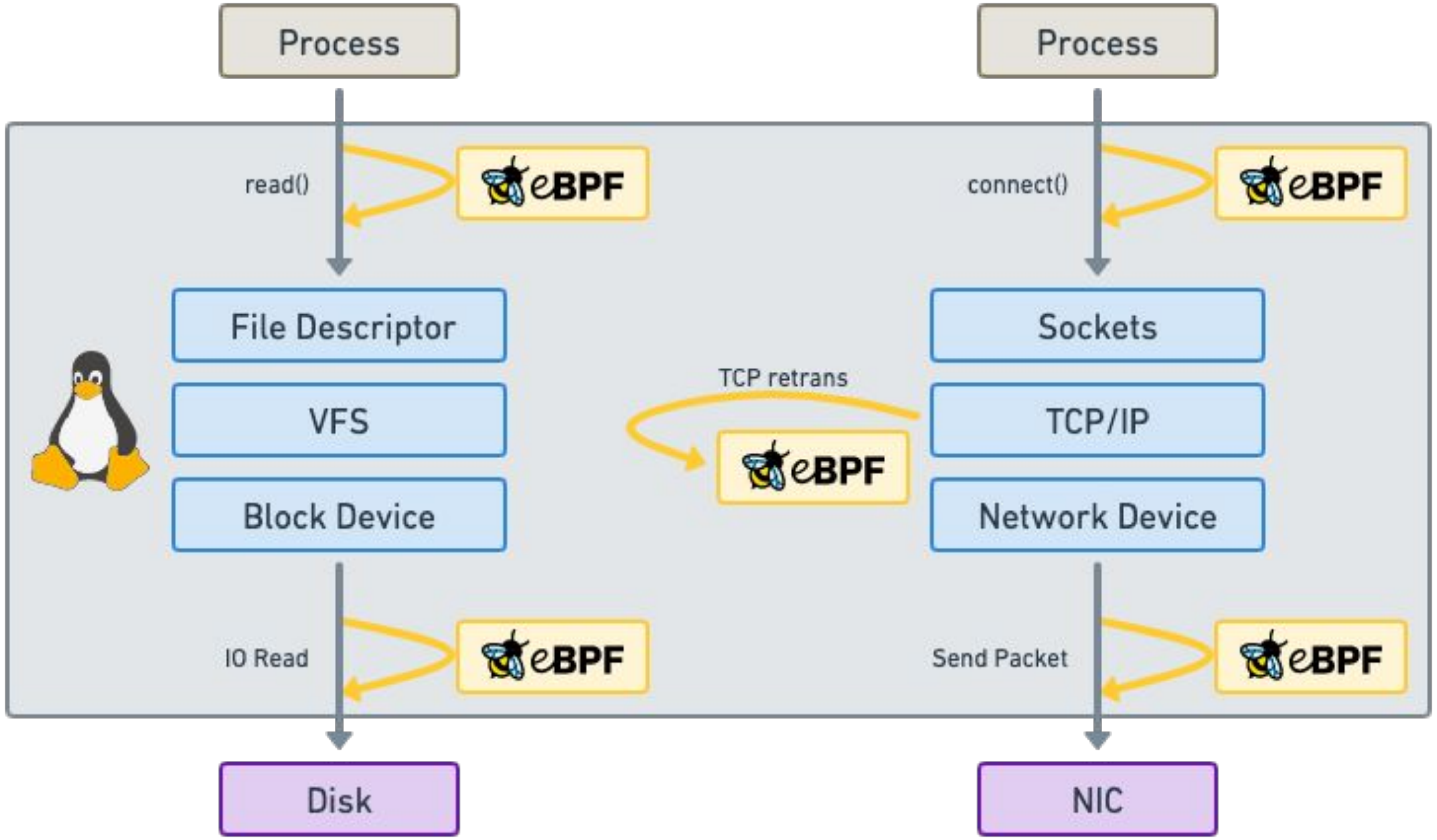



```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

Run eBPF programs on events

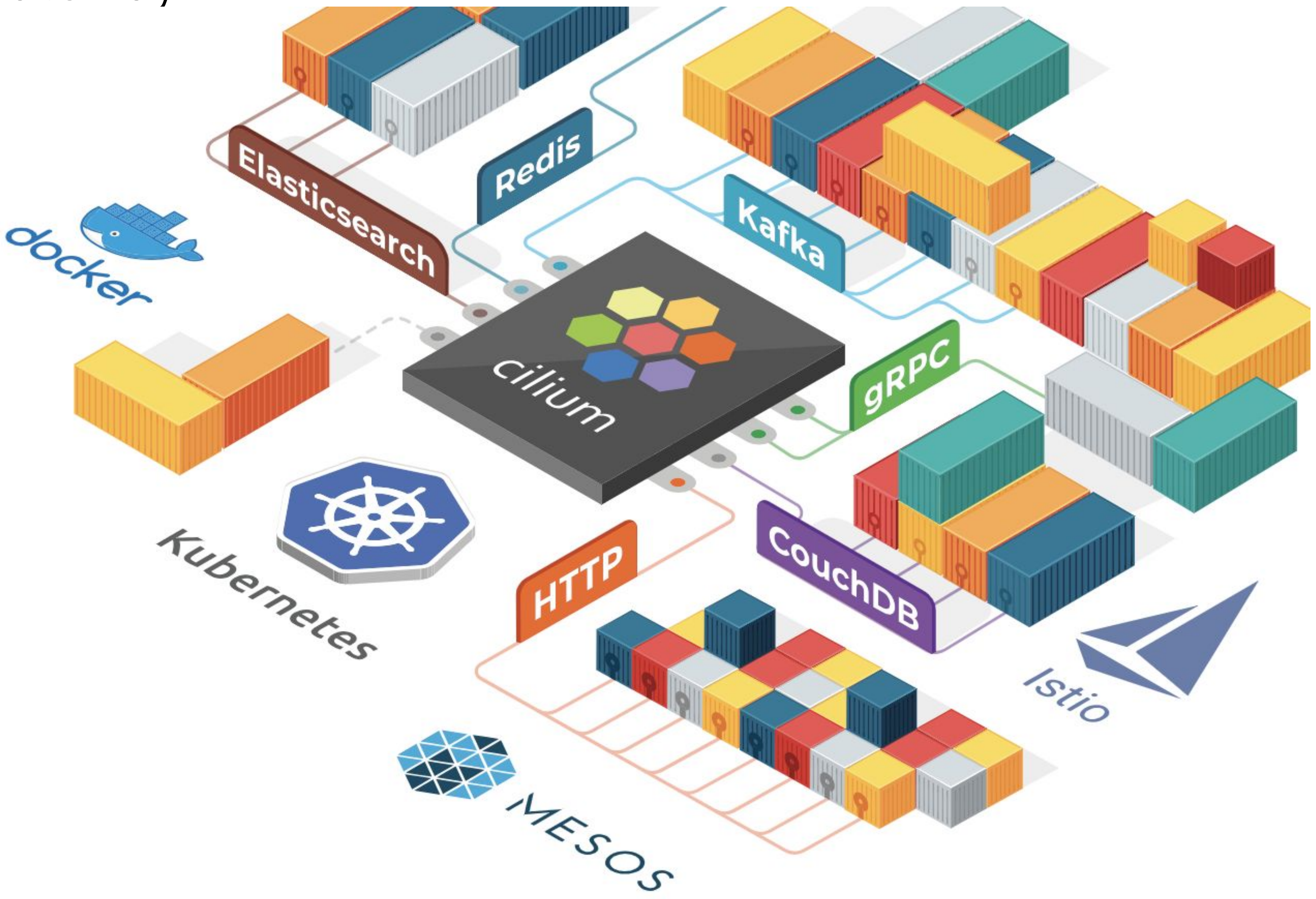


Attachment points

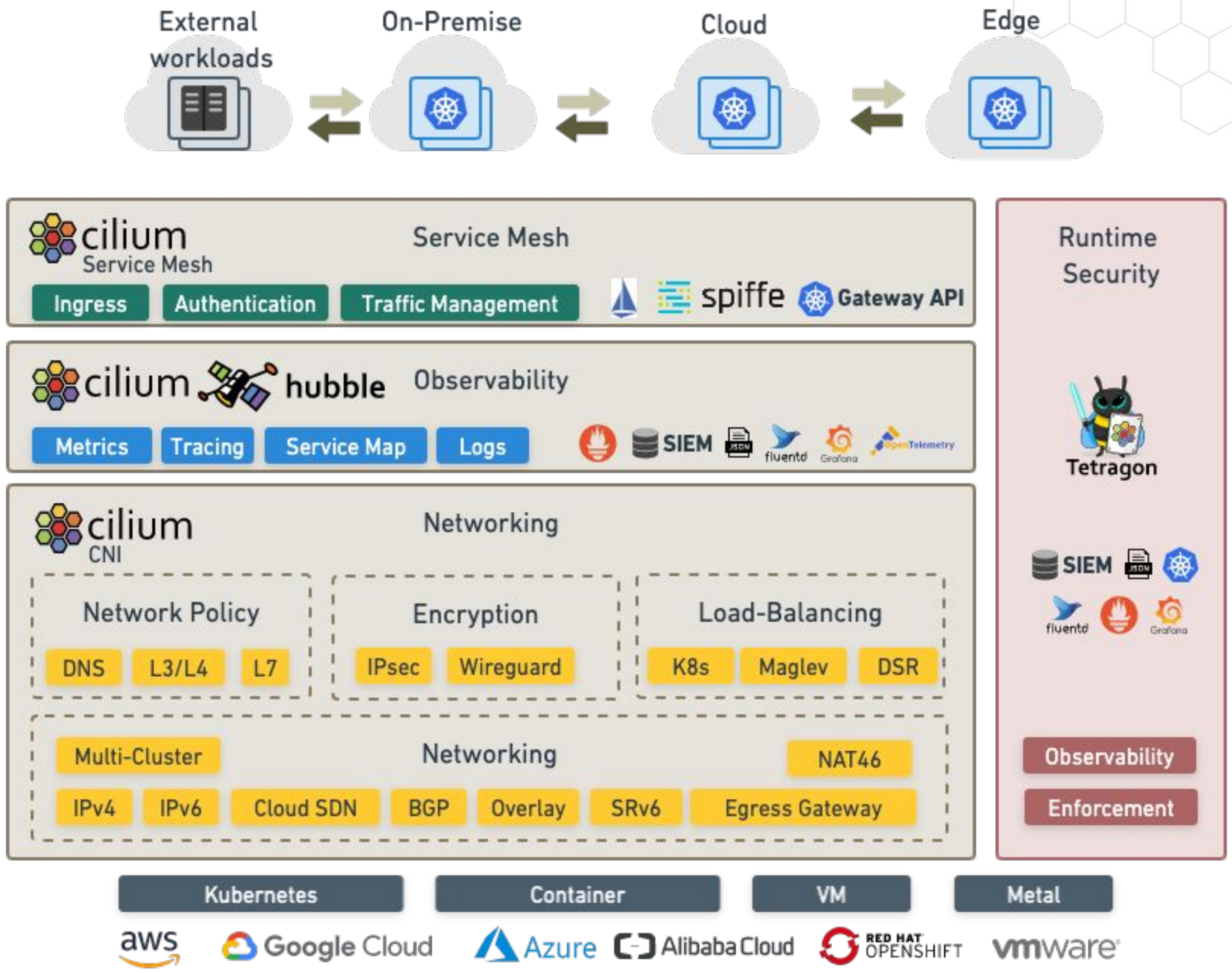
- Kernel functions (kprobes)
- Userspace functions (uprobe)
- System calls
- Tracepoints
- Sockets (data level)
- Network devices (packet level)
- Network device (DMA level) [XDP]
- ...

What is Cilium?

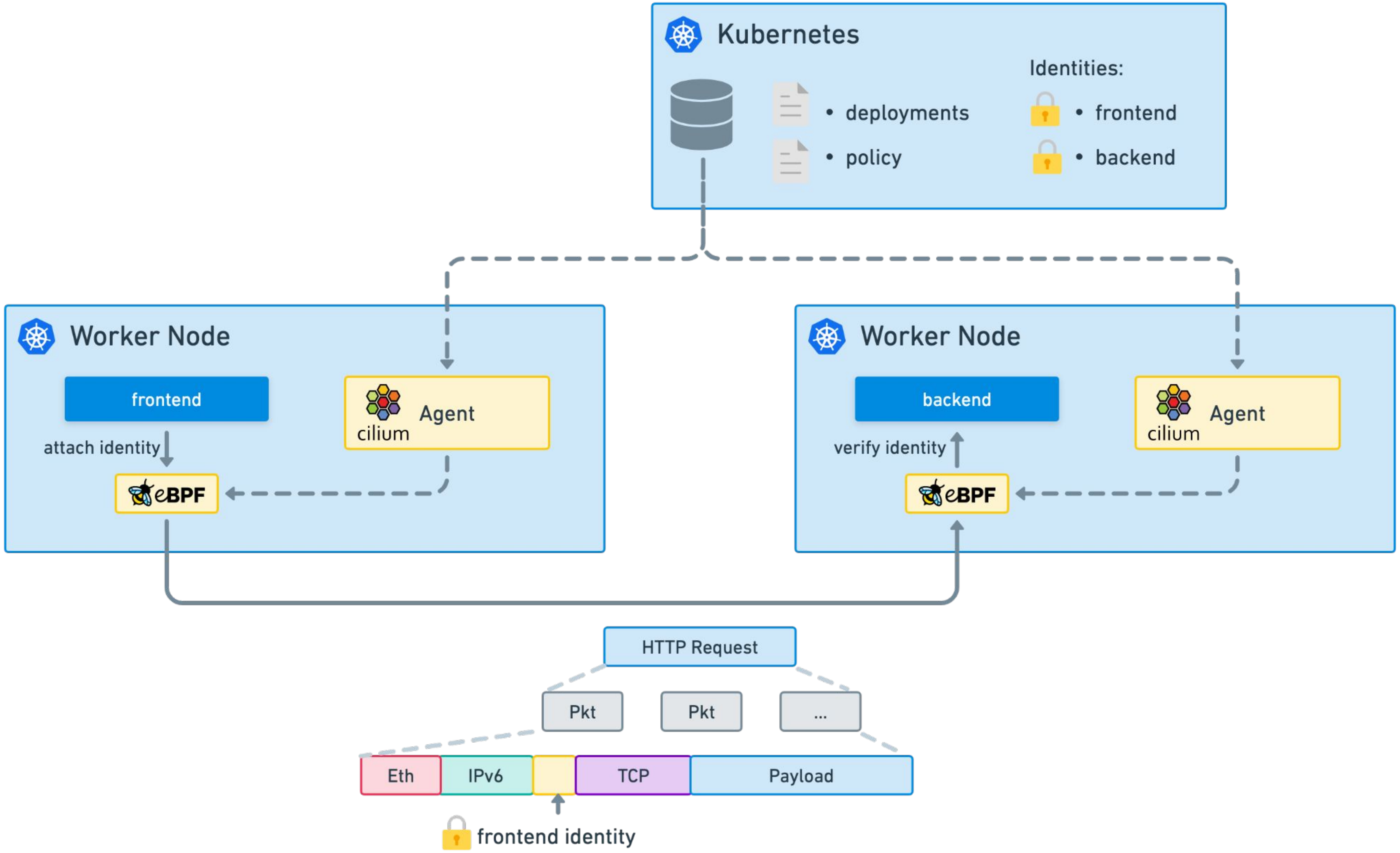
- **Networking & Load-Balancing**
  - CNI, Kubernetes Services, Multi-cluster, VM Gateway
- **Network Security**
  - Network Policy, Identity-based, Encryption
- **Observability**
  - Metrics, Flow Visibility, Service Dependency



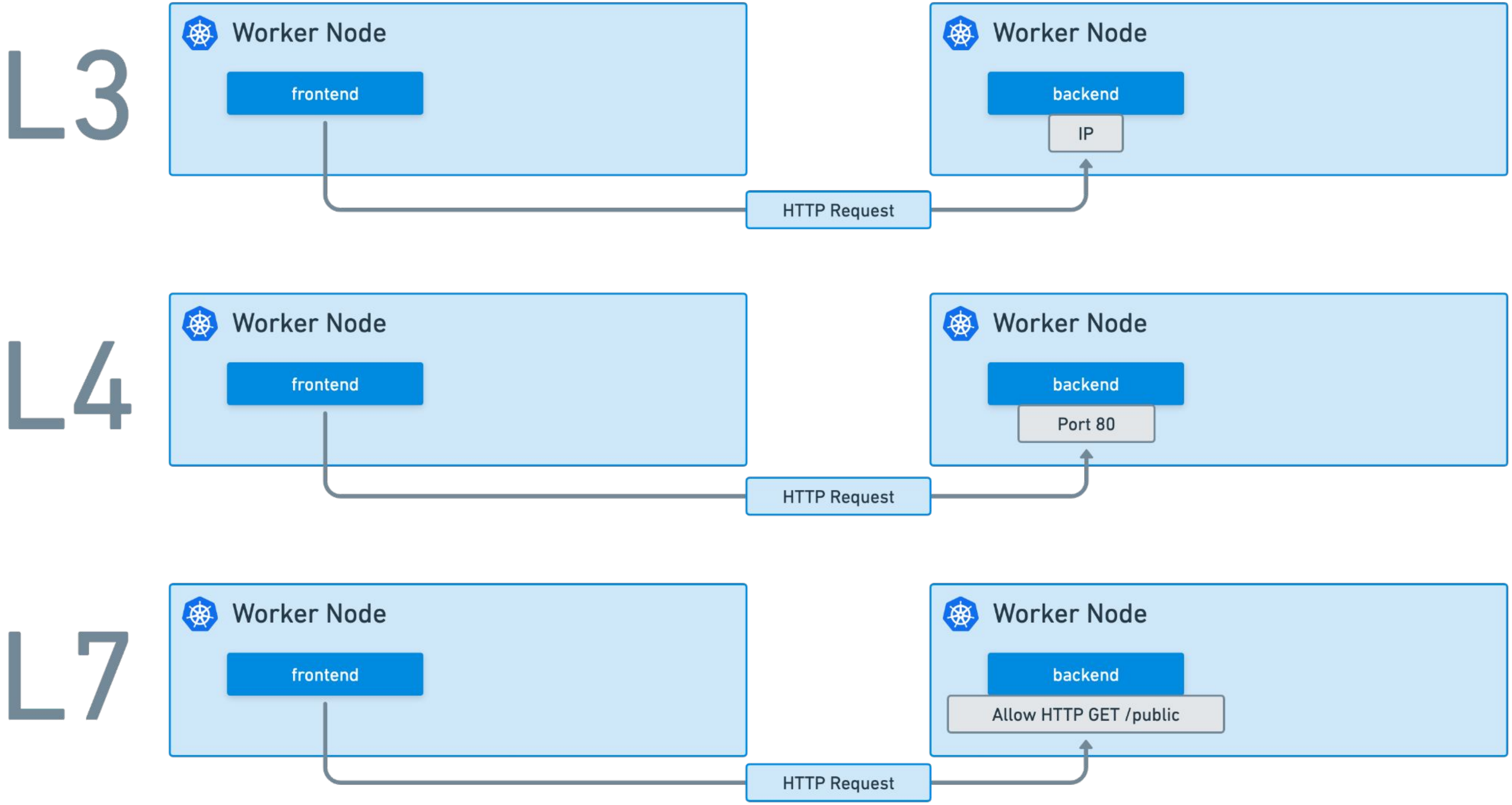
At the foundation of Cilium is the new Linux kernel technology eBPF, which enables the dynamic insertion of powerful security, visibility, and networking control logic within Linux itself. Besides providing traditional network level security, the flexibility of BPF enables security on API and process level to secure communication within a container or pod.
[Read More]


External workloads
On-Premise
Cloud
Edge
cilium
Service Mesh
Service Mesh
Ingress
Authentication
Traffic Management
spiffe
Gateway API
cilium
hubble
Observability
Metrics
Tracing
Service Map
Logs
SIEM
fluentd
cilium
CNI
Networking
Network Policy
DNS
L3/L4
L7
Encryption
IPsec
Wireguard
Load-Balancing
K8s
Maglev
DSR
Multi-Cluster
Networking
NAT46
IPv4
IPv6
Cloud SDN
BGP
Overlay
SRv6
Egress Gateway
Runtime Security
Tetragon
SIEM
fluentd
Observability
Enforcement
Kubernetes
Container
VM
Metal
aws
Google Cloud
Azure
Alibaba Cloud
RED HAT OPENSHIFT
vmware

Identity-based Security



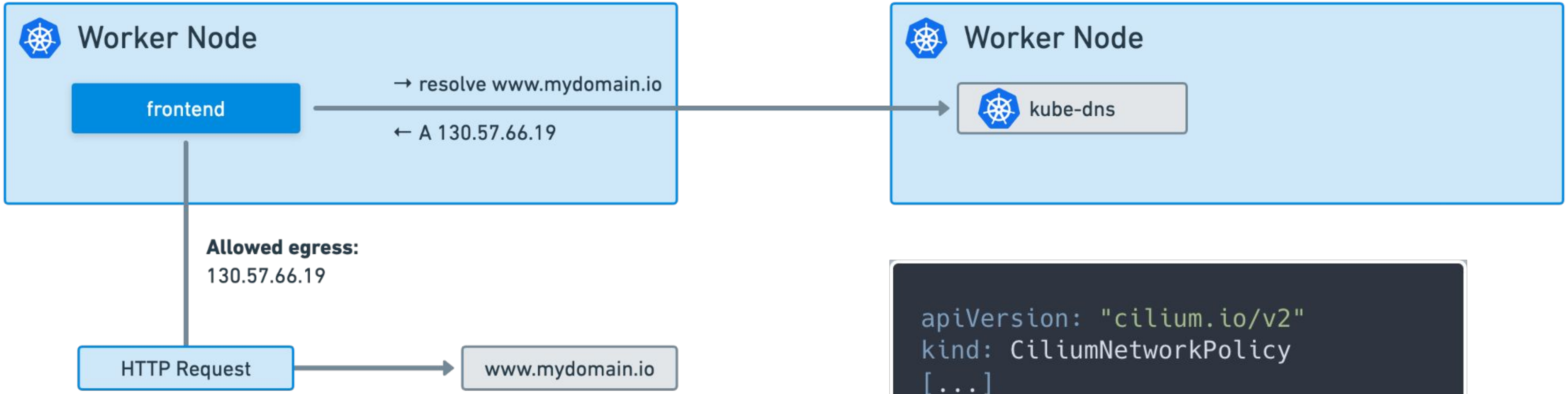
API-aware Authorization



HTTP-Aware Cilium Network Policy

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "http-aware-rule"
spec:
  description: "L7 policy to restrict access to specific HTTP call"
  endpointSelector:
    matchLabels:
      role: frontend
  ingress:
  - fromEndpoints:
    - matchLabels:
        role: frontend
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
      rules:
        http:
        - method: "GET"
          path: "/public"
```

DNS-aware Cilium Network Policy

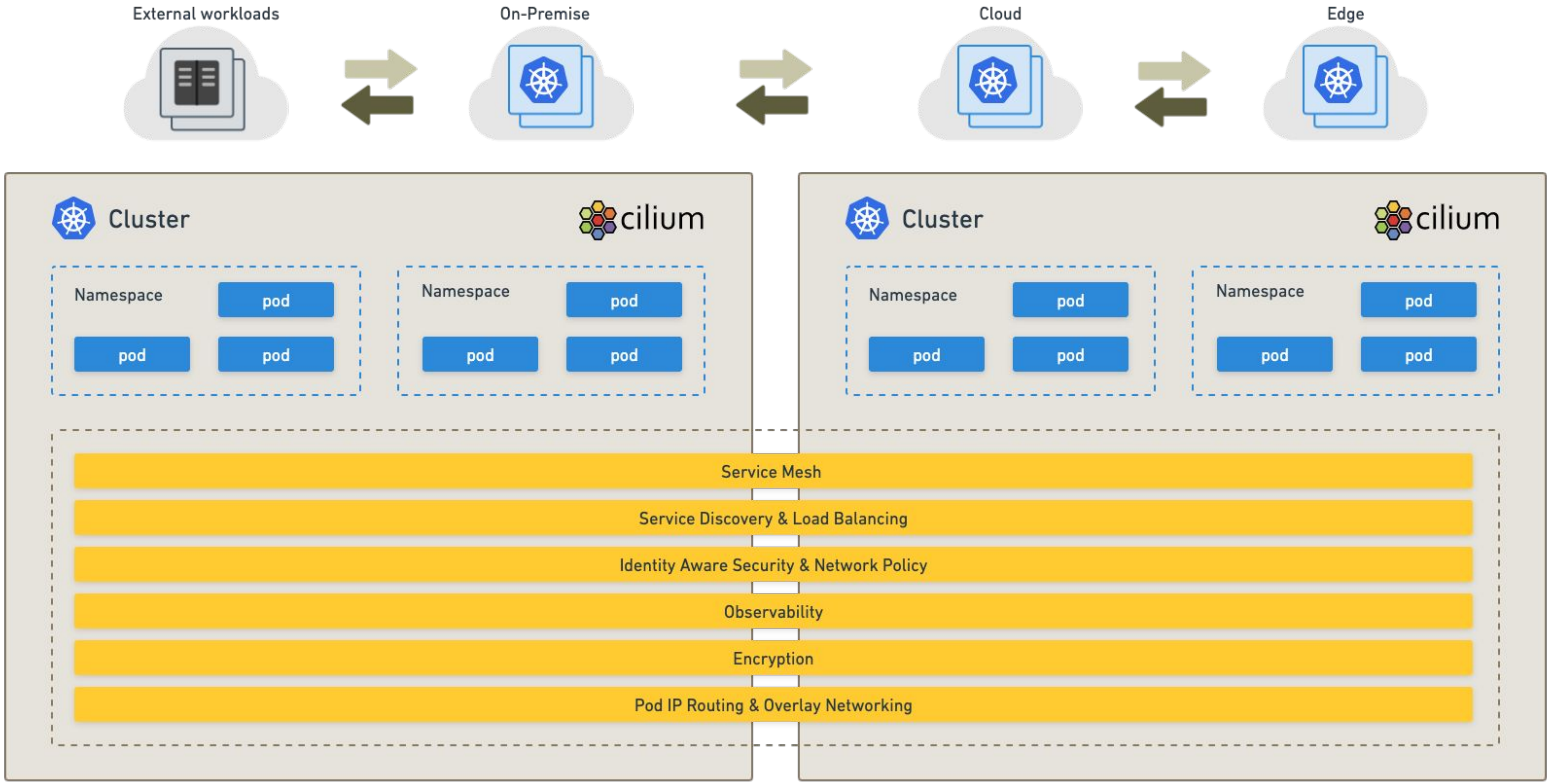



```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
[...]
specs:
- endpointSelector:
    matchLabels:
      app: frontend
    egress:
    - toFQDNs:
      - matchName: "*.mydomain.io"
      toPorts:
      - ports:
        - port: "443"
          protocol: TCP
```

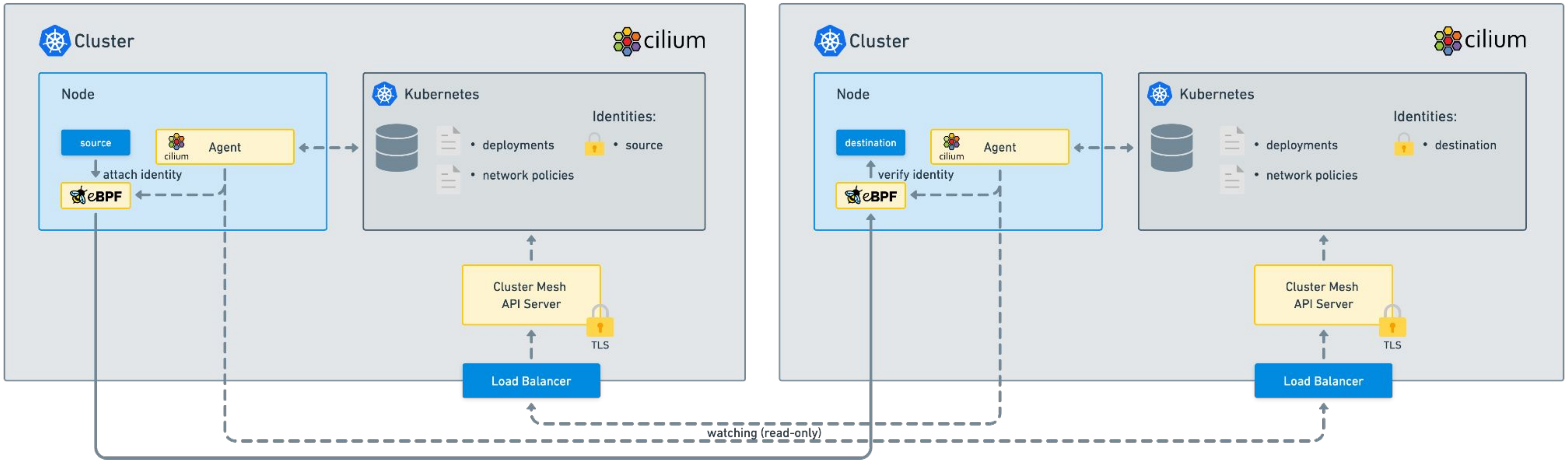
Cilium Cluster Wide Network Policy

```yaml
apiVersion: cilium.io/v2
kind: CiliumClusterwideNetworkPolicy
metadata:
  name: coredns
  namespace: kube-system
spec:
  endpointSelector:
    matchLabels:
      "k8s:k8s-app": kube-dns
  egress:
    - toEntities:
        - world
      toPorts:
        - ports:
            - port: "53"
              protocol: ANY
    - fromEndpoints:
        - matchLabels:
            "k8s:app.kubernetes.io/name": prometheus
            "k8s:io.kubernetes.pod.namespace": monitoring
      toPorts:
        - ports:
            - port: "9153"
              protocol: TCP
```

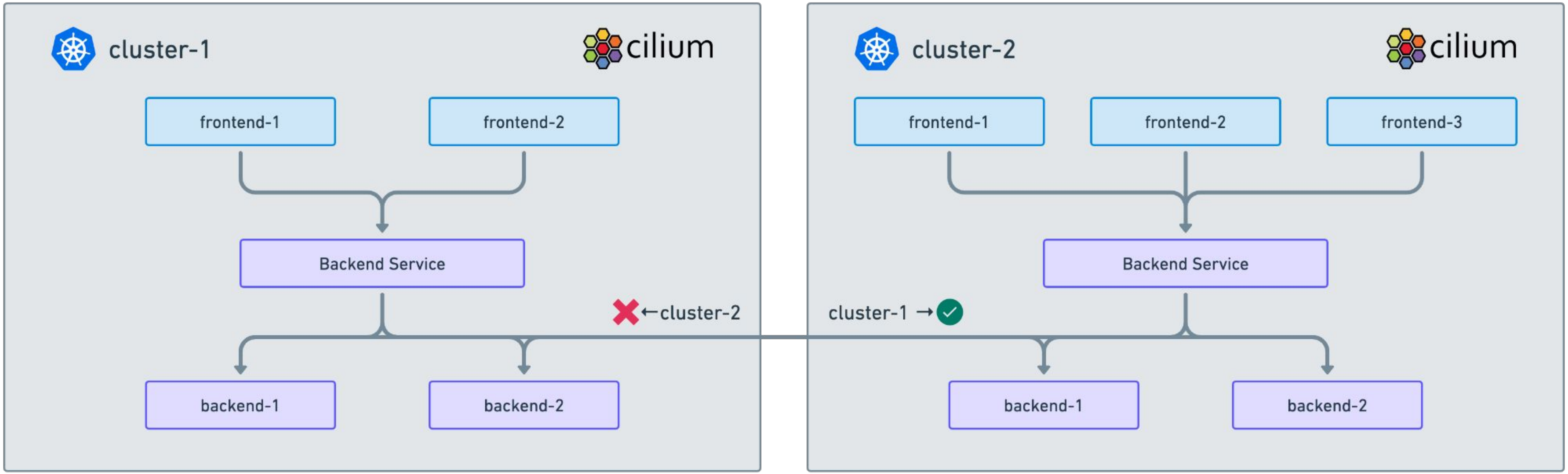
Cluster Mesh - Introduction



Cluster Mesh - Identity Aware Security



Cluster Mesh - Cilium Network Policies



Cluster Mesh - Cilium Network Policies

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "ingress-to-rebel-base"
spec:
  description: "Allow x-wing in cluster-1 to contact rebel-base in cluster2"
  endpointSelector:
    matchLabels:
      name: rebel-base
      io.cilium.k8s.policy.cluster: cluster-2
  ingress:
  - fromEndpoints:
    - matchLabels:
        name: x-wing
        io.cilium.k8s.policy.cluster: cluster-1
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

Observability as the Network Policy Superpower

Troubleshooting Network Policies

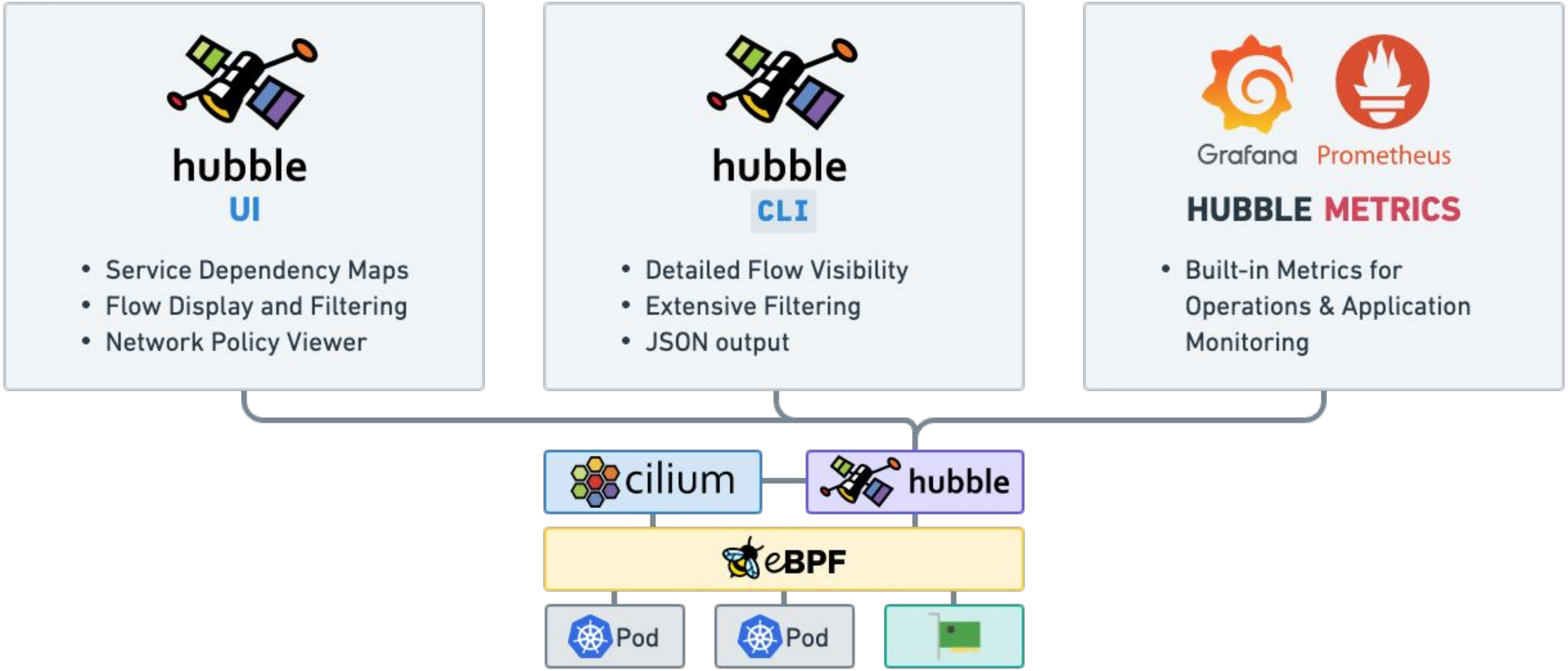
Once enforced, Network Policy is a potential cause of any application outage:

- How to quickly confirm or deny whether this is the case?
- How to enable app teams to achieve this on their own, without taking cycles from the platform team?

Challenge:

- Kubernetes does not provide feedback on allow/denied connections.
- Traditional flow logs from the network:
  - Contain IPs which are ephemeral/meaningless in K8s environments.
  - Only shows allowed traffic (denied traffic is already dropped).
  - Typically these logs are not directly accessible to app teams.

Hubble Overview



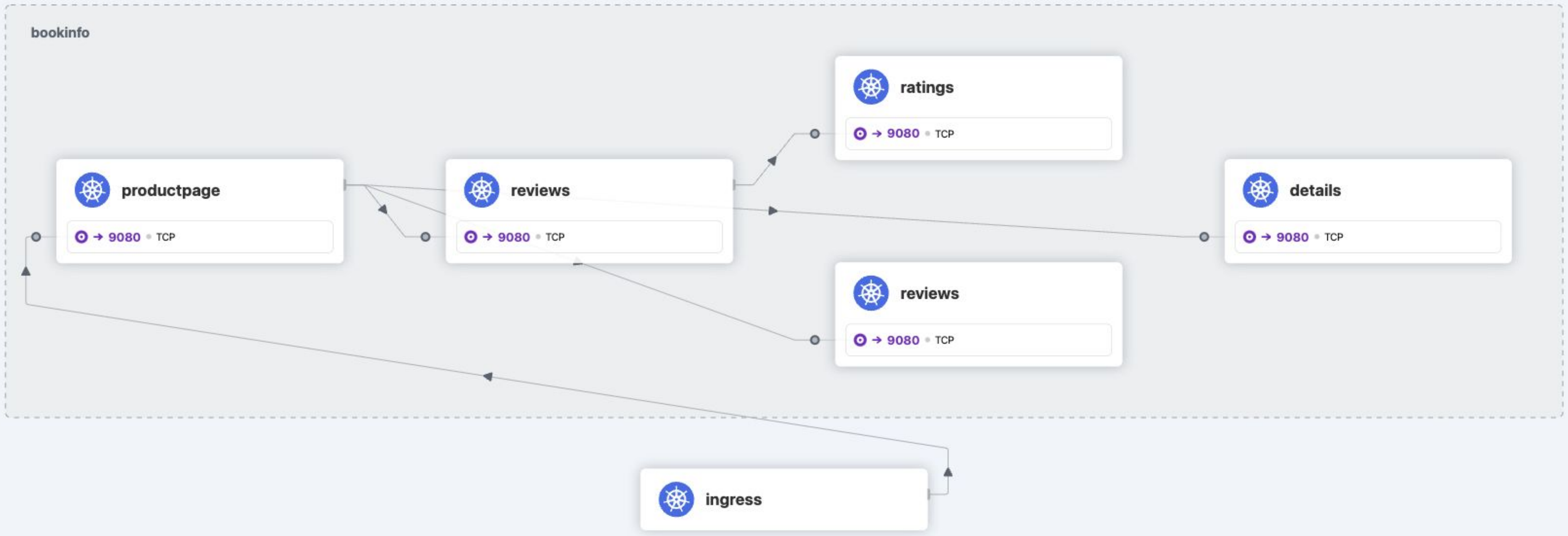
Using Hubble Observability Data to Build Policies

bookinfo

Filter by: label key=val, ip=1.1.1.1, dns=google.com, identity=42, pod=frontend

Any verdict | Visual

167.9 flows/s • 3/3 nodes



Columns

Source Service	Destination Service	Destination Port	L7 info	Verdict	Timestamp
ingress	details *bookinfo*	9080	—	forwarded	half a minute
ingress	details *bookinfo*	9080	—	forwarded	1 minute
ingress	details *bookinfo*	9080	—	forwarded	1 minute
ingress	details *bookinfo*	9080	—	forwarded	1 minute
reviews *bookinfo*	ratings *bookinfo*	9080	—	forwarded	1 minute
reviews *bookinfo*	ratings *bookinfo*	9080	—	forwarded	1 minute
productpage *bookinfo*	reviews *bookinfo*	9080	—	forwarded	1 minute
productpage *bookinfo*	reviews *bookinfo*	9080	—	forwarded	1 minute

Using Hubble Observability Data to Build Policies

Day 1:

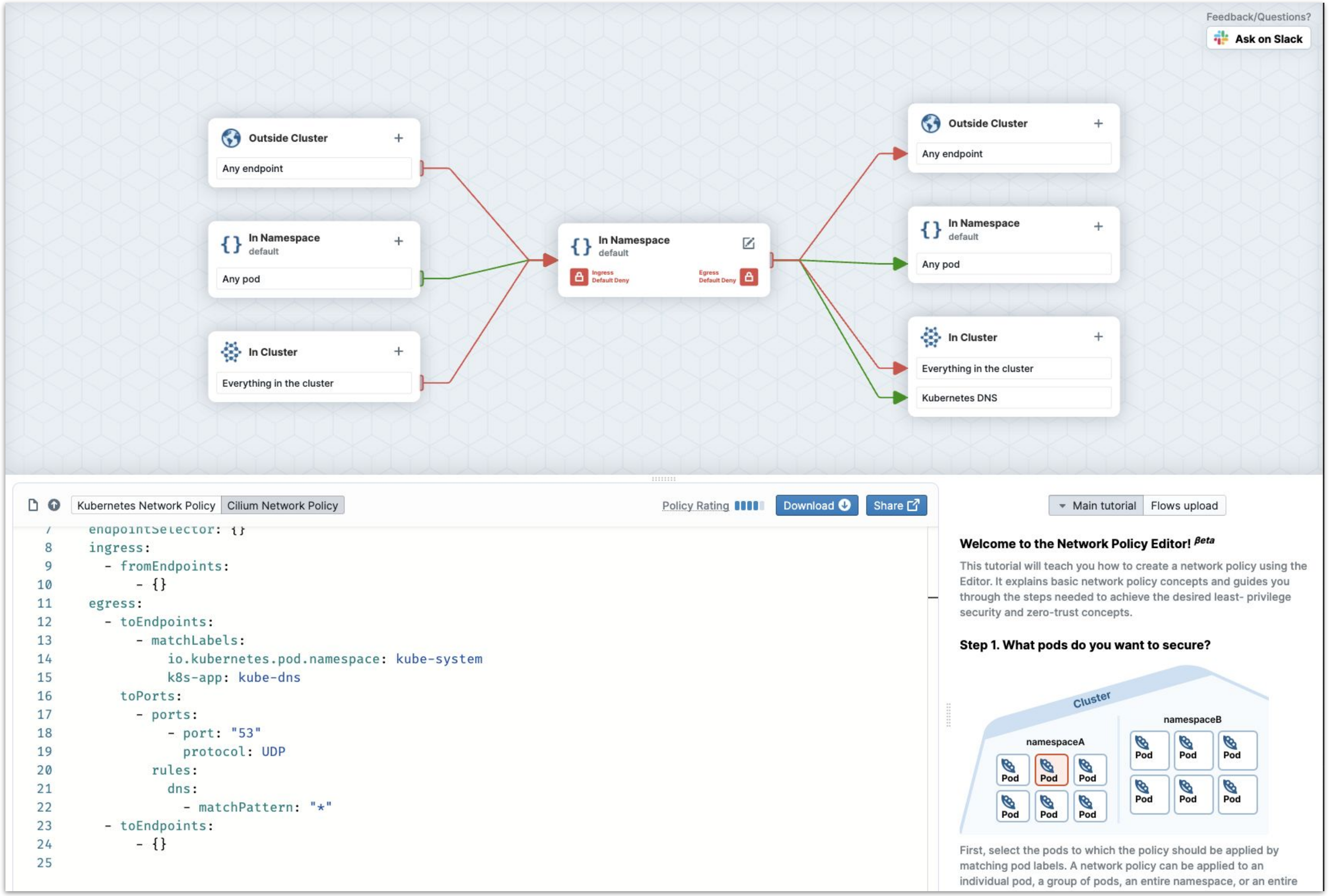
- identify commonly shared services to include in baseline policy.
- identify “low-hanging fruit” namespaces easily locked down at ingress/egress (no per-namespace exceptions required).
- identify the set of “exceptions” required for a given per-namespace policy

Day 2:

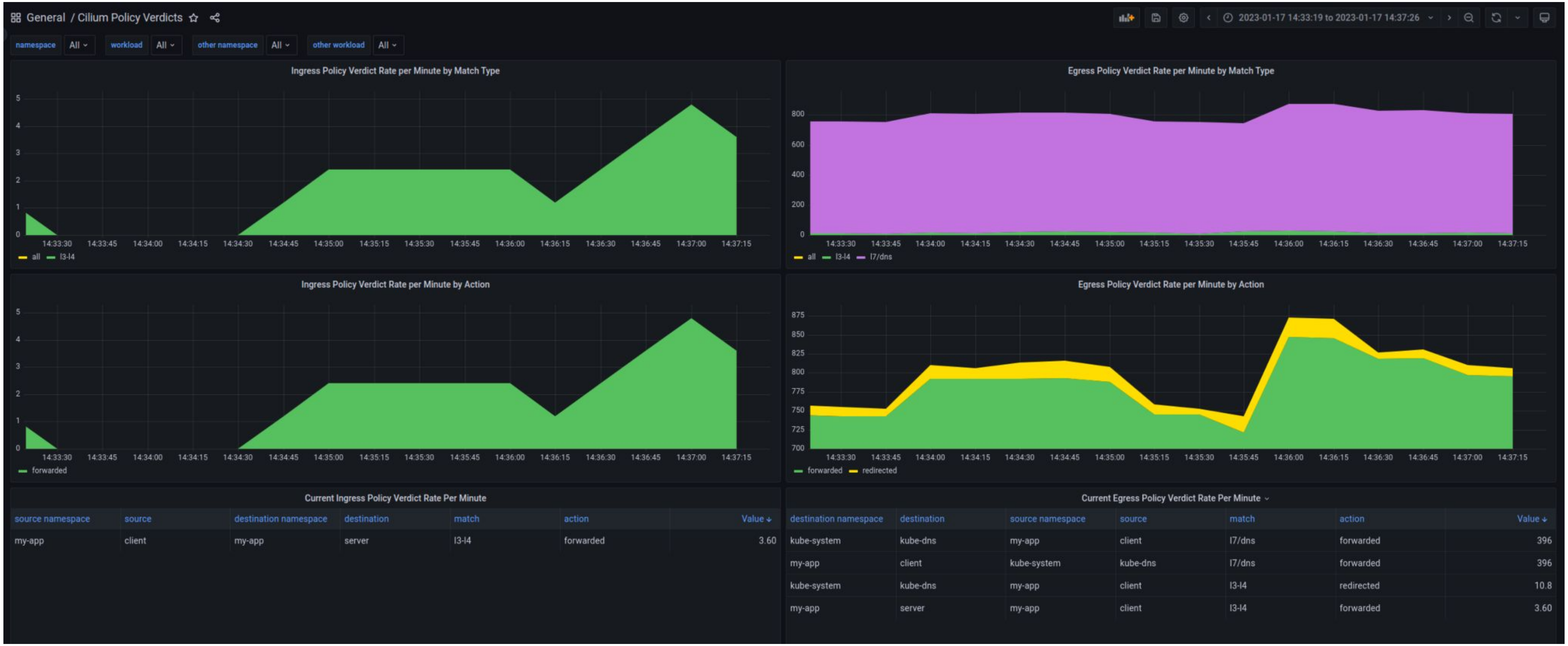
- Flow data can be sourced from dev/staging clusters as well as production to identify the network policy changes required to ship a new app version.
- Historical flow data can be analyzed to predict impact of new more stringent baseline or per-namespace policies.
- Historical flow data combined with enforced policies can be used to detect overly broad rules. For example, “egress allow 0.0.0.0/0 port 443” when a rule to a few specific DNS names would suffice.

Network Policy Editor

https://editor.cilium.io



Hubble & Grafana for Policy Verdicts Metrics



Demo

Learn more!

ISOVALENT

For the Enterprise

Hardened, enterprise-grade eBPF-powered networking, observability, and security.

isovalent.com/product
isovalent.com/labs

cilium

OSS Community

eBPF-based Networking, Observability, Security

cilium.io
cilium.slack.com
Regular news

eBPF

Base technology

The revolution in the Linux kernel, safely and efficiently extending the capabilities of the kernel.

ebpf.io
What is eBPF? - ebook

ISOVALENT

Thank you!