ISOVALENT

Network Policies

The Not-So-Hard-Way

Speaker: **Tracy Holmes & Raymond de Jong**

Agenda

- Cilium & eBPF Introduction
- Security
- Observability
- Cilium Network Policies Fundamentals
- Strategies for Designing Network Policies
- Observability as the Network Policy Superpower
- Getting it done! - Demo
- Isovalent Lab - Isovalent Cilium Enterprise: Network Policies

Cilium & eBPF

Introduction

- Open Source Projects

ISOVALENT

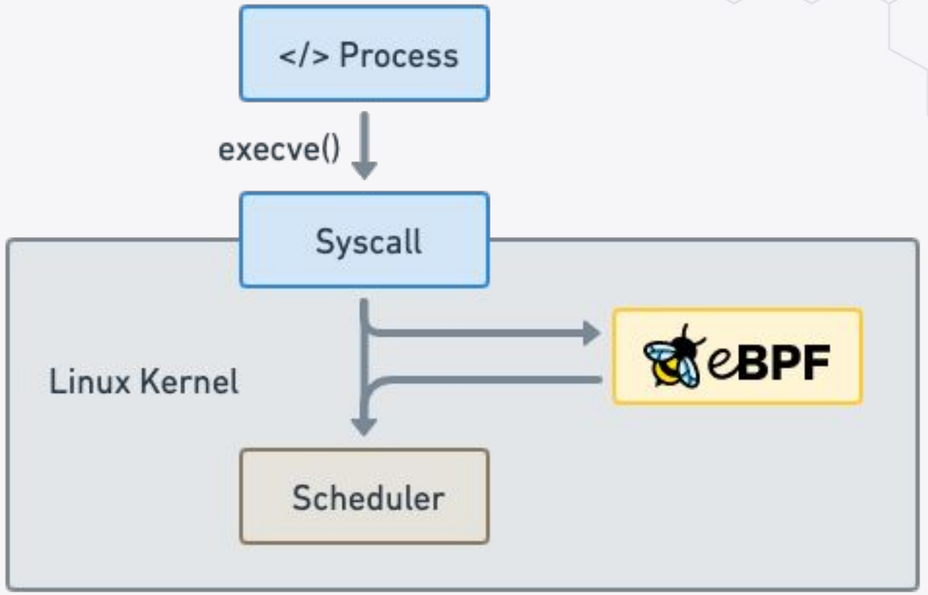
- Company behind Cilium
- Provides Cilium Enterprise


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”

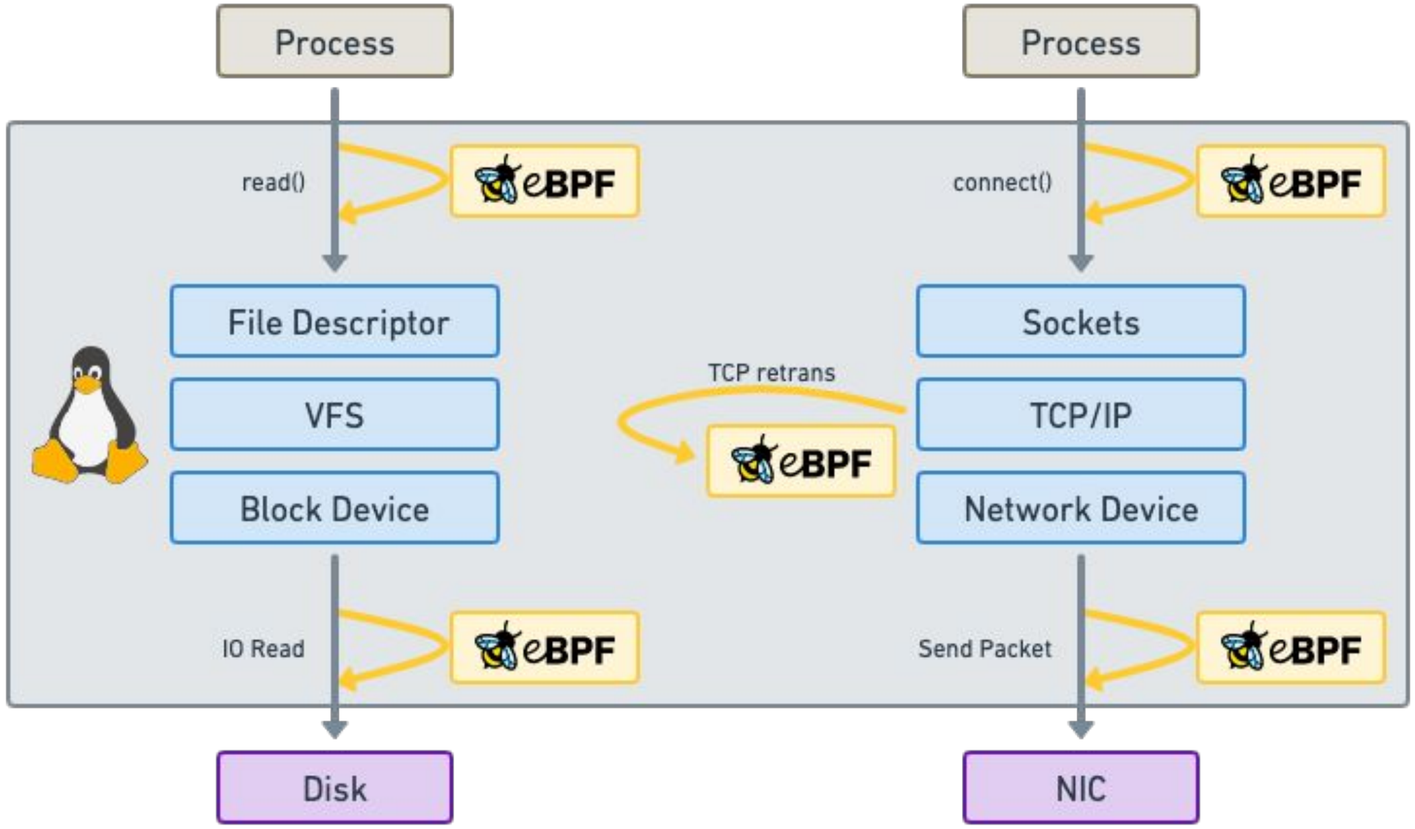



```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

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

        return 0;
}
```

Run eBPF programs on events



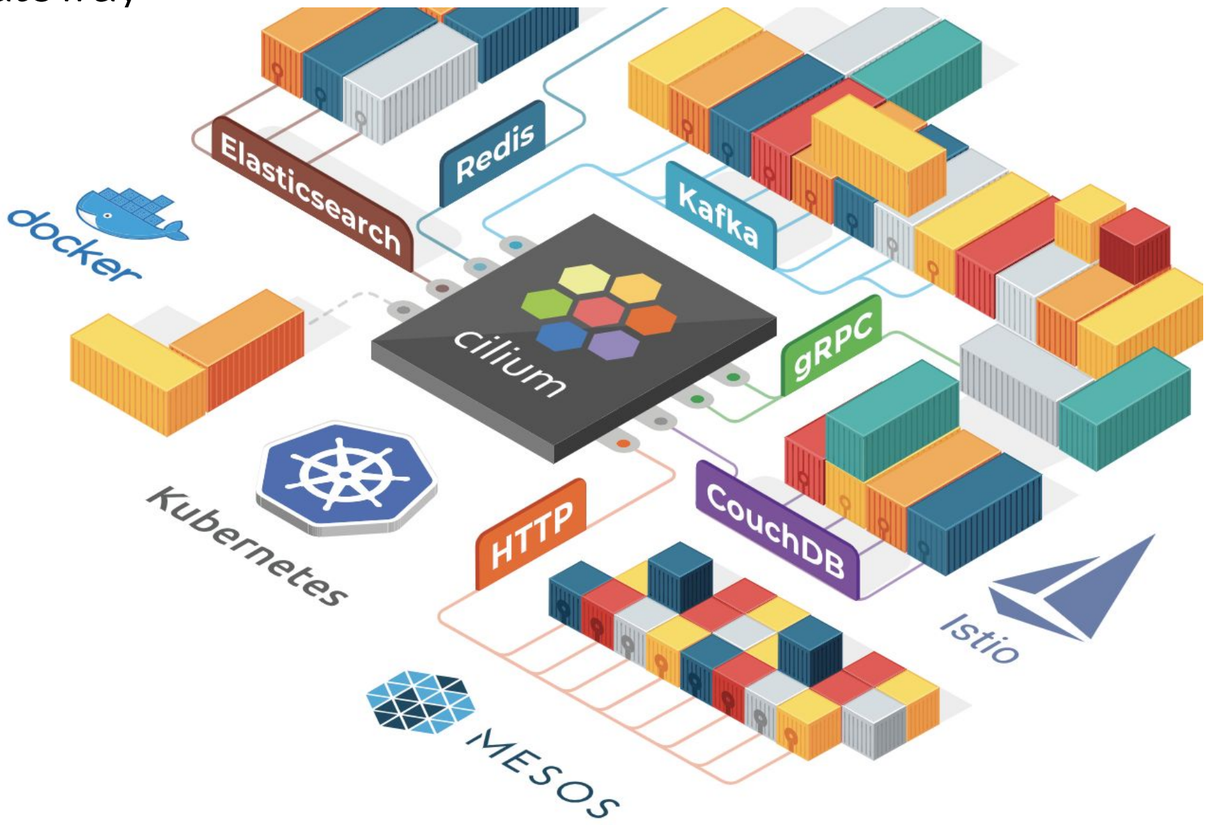
Attachment points

- 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



cilium

Created by ISOVALENT

eBPF-based:

- Networking
- Security
- Observability
- Service Mesh & Ingress

Foundation

CLOUD NATIVE COMPUTING FOUNDATION

Technology

eBPF envoy

Building a Global Multi Cluster Gaming Infrastructure with Cilium

---

What Makes a Good Multi-tenant Kubernetes Solution

Building a Secure and Maintainable PaaS

Building High-Performance Cloud-Native Pod Networks

Scaling a Multi-Tenant k8s Cluster in a Telco

First step towards cloud native networking

Cloud Native Networking with eBPF

Managed Kubernetes: 1.5 Years of Cilium Usage at DigitalOcean

---

Google chooses Cilium for Google Kubernetes Engine (GKE) networking

---

Why eBPF is changing the Telco networking space?

Kubernetes Network Policies in Action with Cilium

AWS picks Cilium for Networking & Security on EKS Anywhere

Scaleway uses Cilium as the default CNI for Kubernetes Kapsule

---

Sportradar is using Cilium as their main CNI plugin in AWS (using kops)

Utmost is using Cilium in all tiers of its Kubernetes ecosystem to implement zero trust

---

Yahoo is using Cilium for L4 North-South Load Balancing for Kubernetes Services

ISOVALENT

cilium Service Mesh

Service Mesh

Ingress | Authentication | Traffic Management | spiffe | Gateway API

cilium hubble

Observability

Metrics | Tracing | Service Map | Logs | SIEM | fluentd | Grafana | OpenTelemetry

cilium CNI

Networking

Network Policy: DNS | L3/L4 | L7

Encryption: IPsec | Wireguard

Load-Balancing: K8s | Maglev | DSR

Networking: Multi-Cluster | NAT46 | IPv4 | IPv6 | Cloud SDN | BGP | Overlay | SRv6 | Egress Gateway

Runtime Security

Tetragon

SIEM | fluentd | Grafana

Observability

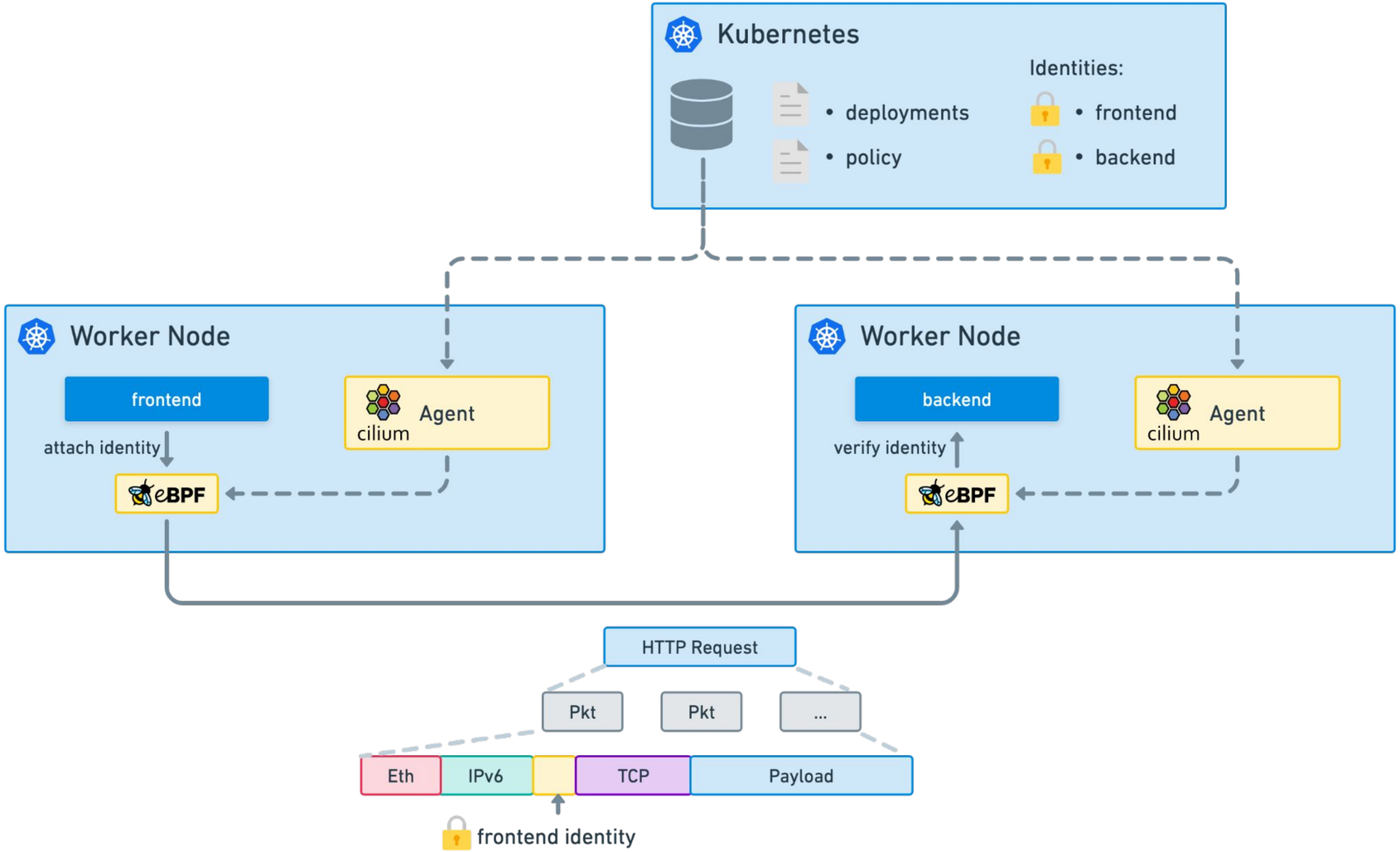
Enforcement

Kubernetes | Container | VM | Metal

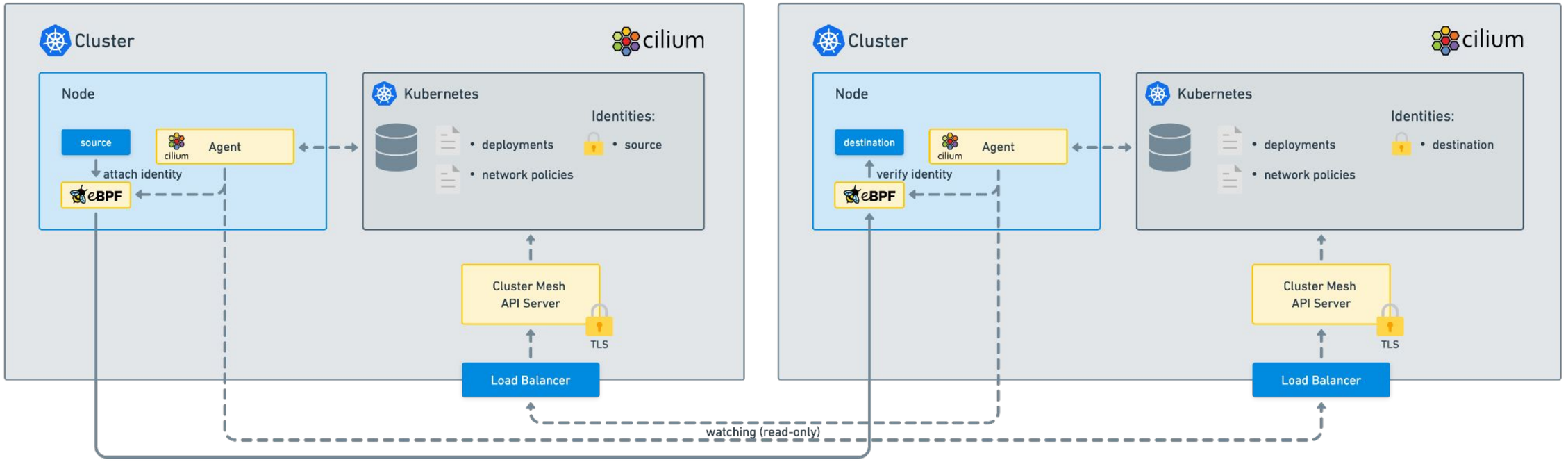
aws | Google Cloud | Azure | Alibaba Cloud | RED HAT OPENSHIFT | vmware

Security

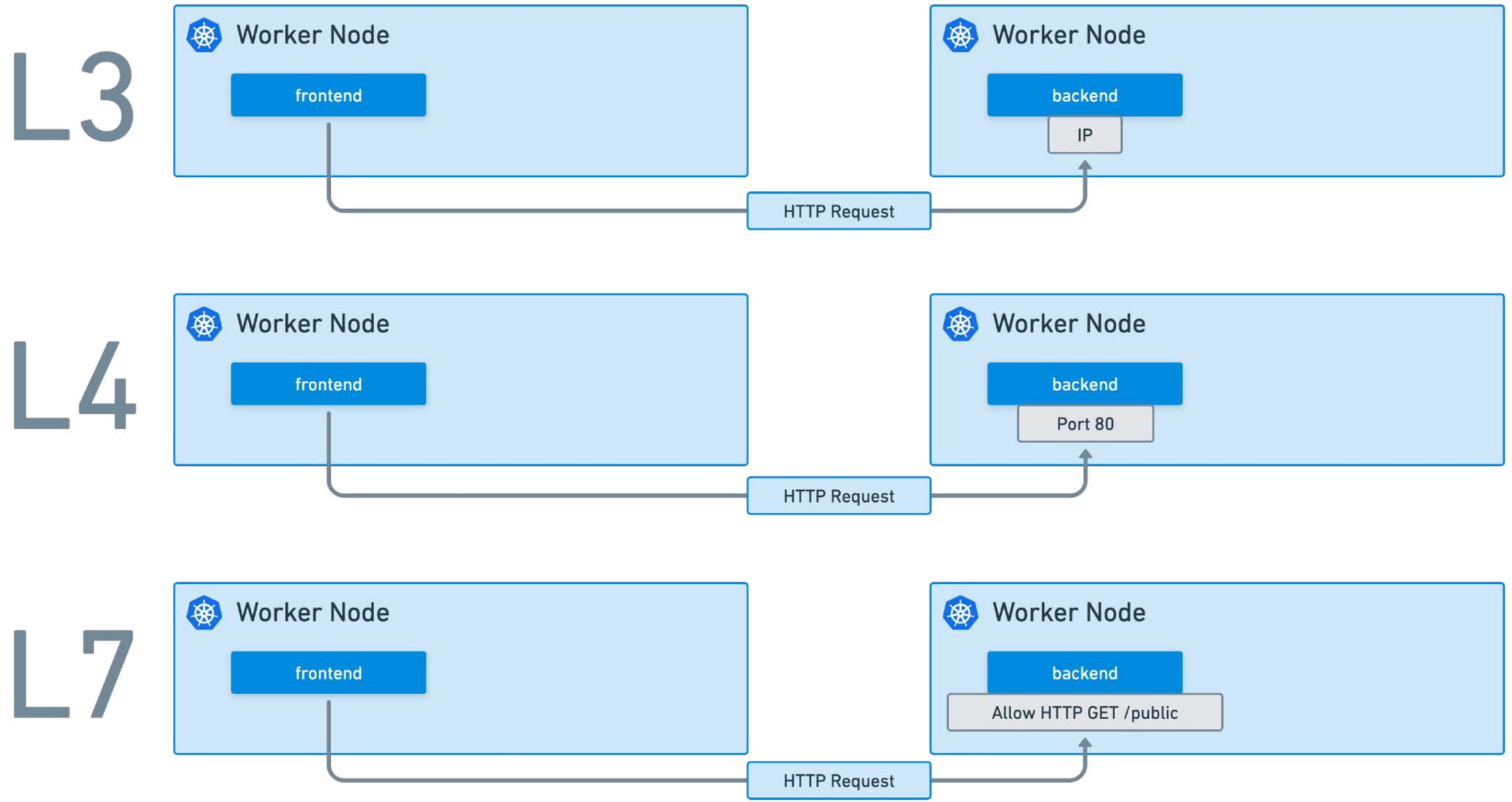
Identity-based Security



Cluster Mesh - Identity Aware Security



API-aware Authorization



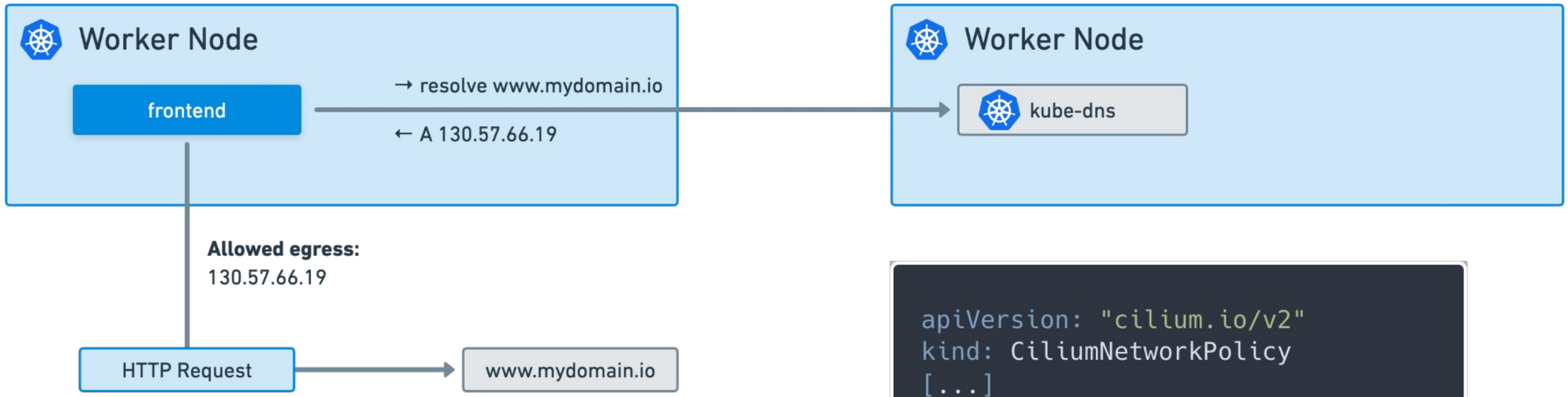
HTTP-Aware Cilium Network Policy


```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"
```

Cassandra Cilium Network Policy Example

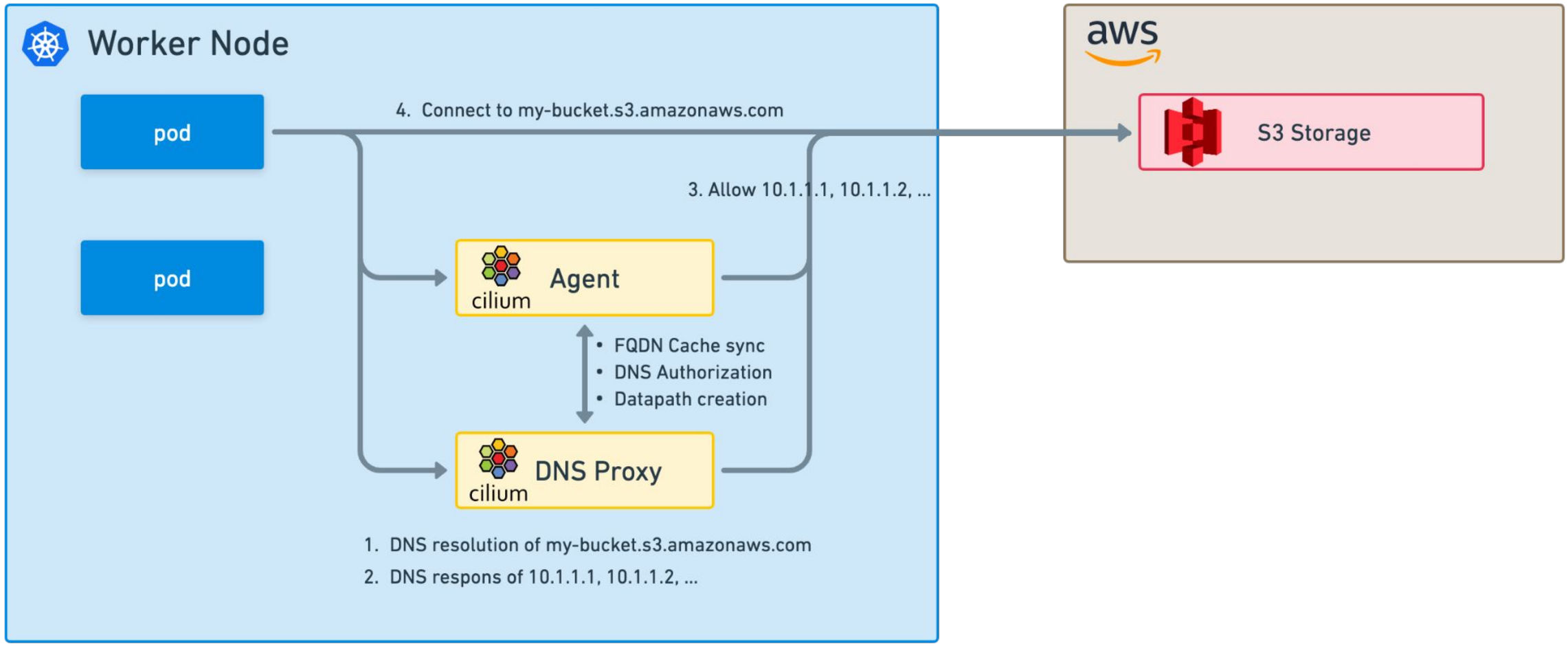
```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
[...]
specs:
- endpointSelector:
  matchLabels:
    app: cassandra
  ingress:
  - toPorts:
    - ports:
      - port: "9042"
        protocol: TCP
        l7proto: cassandra
        l7:
        - query_action: "select"
          query_table: "myTable"
```

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
```

DNS Proxy HA


Worker Node
pod
pod
4. Connect to my-bucket.s3.amazonaws.com
3. Allow 10.1.1.1, 10.1.1.2, ...
cilium
Agent
• FQDN Cache sync
• DNS Authorization
• Datapath creation
cilium
DNS Proxy
1. DNS resolution of my-bucket.s3.amazonaws.com
2. DNS respons of 10.1.1.1, 10.1.1.2, ...
aws
S3 Storage

L3 Matching Capabilities

Kubernetes

- Pod labels
- Namespace name & labels
- ServiceAccount name
- Service names
- Cluster names

DNS Names

- FQDN and regular expression

CIDR

- CIDR blocks with exceptions

Cloud Providers

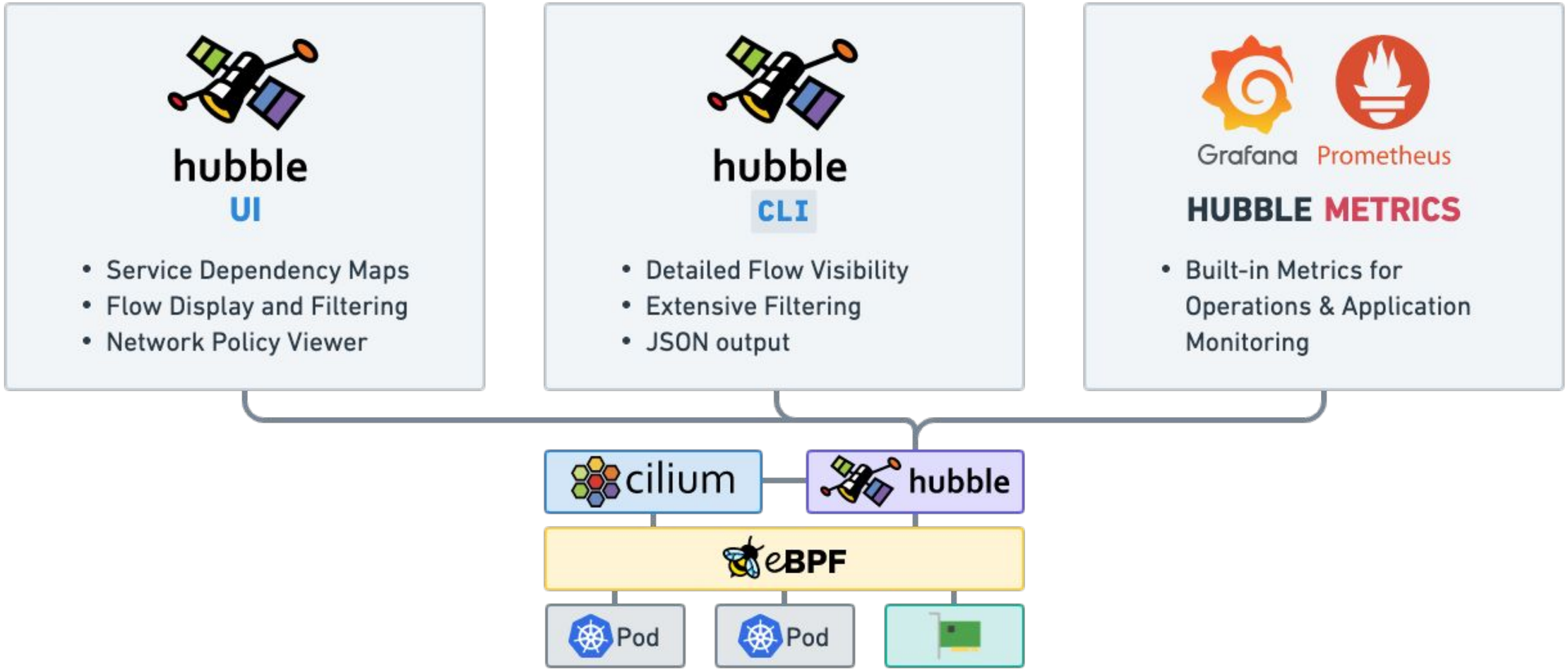
- Instance labels
- VPC/Subnet name/tags
- Security group name

Logical Entities

- Everything inside cluster
- Everything outside cluster
- Local host
- ...

Observability

What is Hubble?



Flow Visibility

```
$ kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
tiefighter                      1/1     Running   0          2m34s
xwing                           1/1     Running   0          2m34s
deathstar-5b7489bc84-crlxh      1/1     Running   0          2m34s
deathstar-5b7489bc84-j7qwq      1/1     Running   0          2m34s

$ hubble observe --follow -l class=xwing
# DNS lookup to coredns
default/xwing:41391 -> kube-system/coredns-66bff467f8-28dgp:53 to-proxy FORWARDED (UDP)
kube-system/coredns-66bff467f8-28dgp:53 -> default/xwing:41391 to-endpoint FORWARDED (UDP)
# ...
# Successful HTTPS request to www.disney.com
default/xwing:37836 -> www.disney.com:443 to-stack FORWARDED (TCP Flags: SYN)
www.disney.com:443 -> default/xwing:37836 to-endpoint FORWARDED (TCP Flags: SYN, ACK)
www.disney.com:443 -> default/xwing:37836 to-endpoint FORWARDED (TCP Flags: ACK, FIN)
default/xwing:37836 -> www.disney.com:443 to-stack FORWARDED (TCP Flags: RST)
# ...
# Blocked HTTP request to deathstar backend
default/xwing:49610 -> default/deathstar:80 Policy denied DROPPED (TCP Flags: SYN)
```

Flow Metadata

- Ethernet headers
- IP & ICMP headers
- UDP/TCP ports, TCP flags
- HTTP, DNS, Kafka, ...

Kubernetes

- Pod names and labels
- Service names
- Worker node names

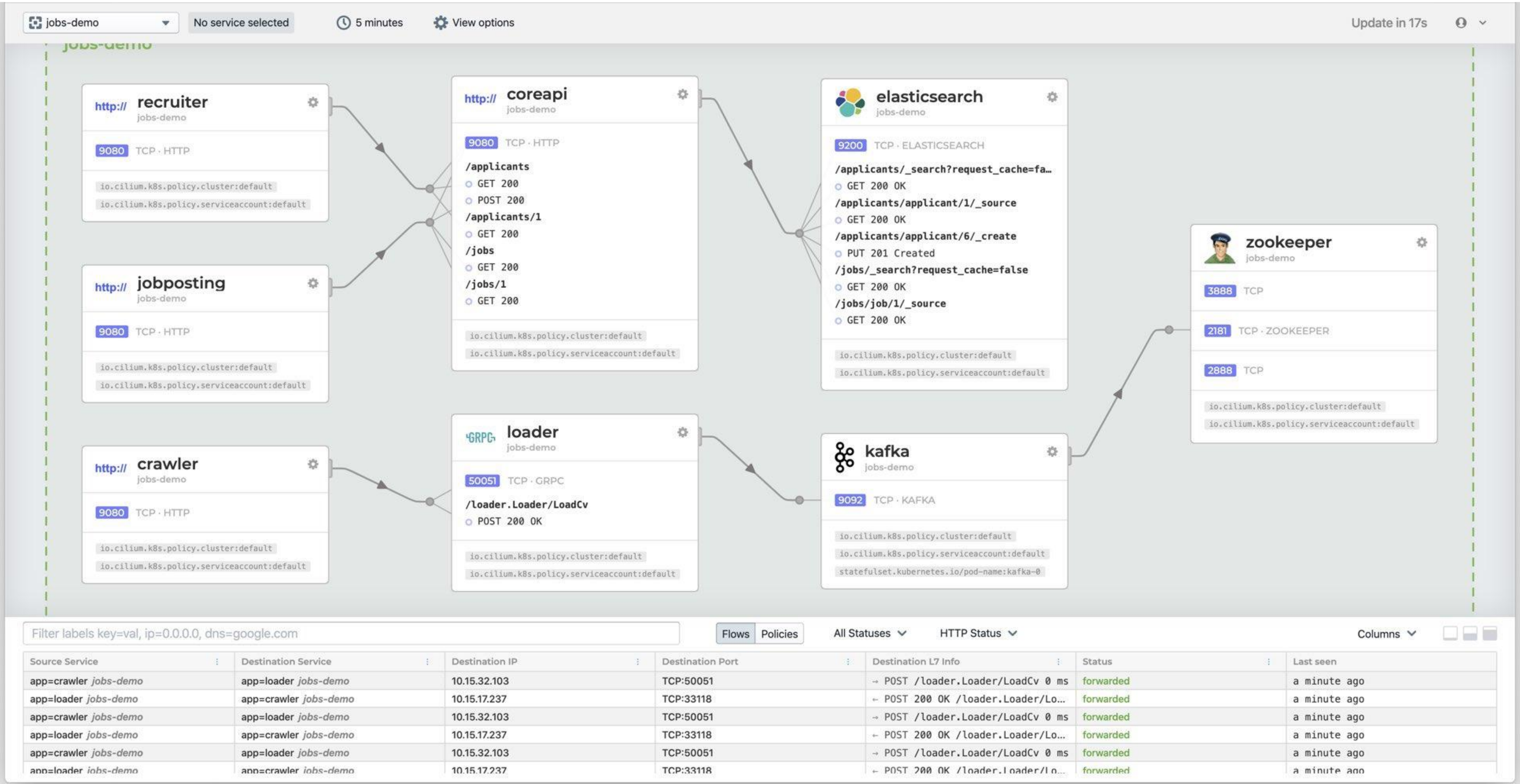
DNS (if available)

- FQDN for source and destination

Cilium

- Security identities and endpoints
- Drop reasons
- Policy verdict matches

Service Map



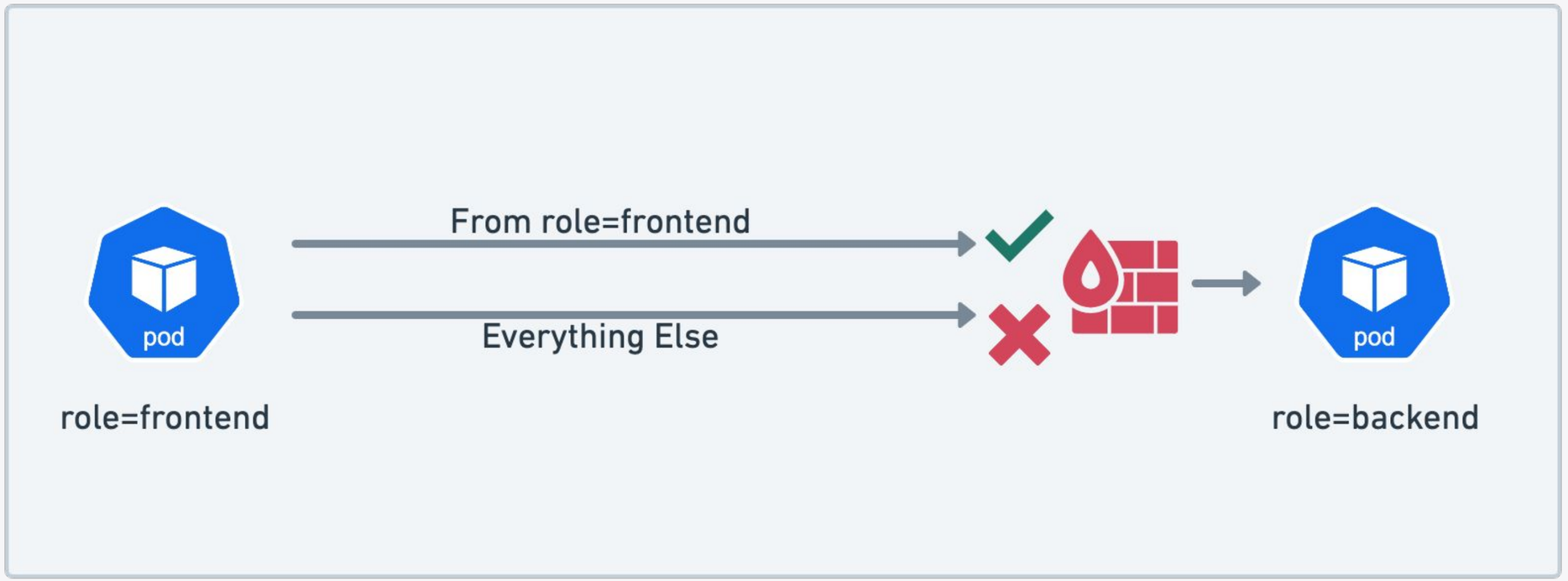
cilium

Hubble Metrics

Cilium Network Policy Fundamentals

What is a Network Policy?

A declarative pod-level network firewall abstraction that understands Kubernetes identity.



Core Concepts

Declarative Policy

- Allowed Connectivity is defined using declarative Kubernetes resources.
  - NetworkPolicy: Built-in k8s abstraction for basic policies.
  - CiliumNetworkPolicy: Custom Resource Definition (CRD) that is super-set of NetworkPolicy, includes advanced Cilium features.
- Each network policy resources defines one or more “rules”.

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l3-rule"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  ingress:
  - fromEndpoints:
    - matchLabels:
        role: frontend
```

Core Concepts

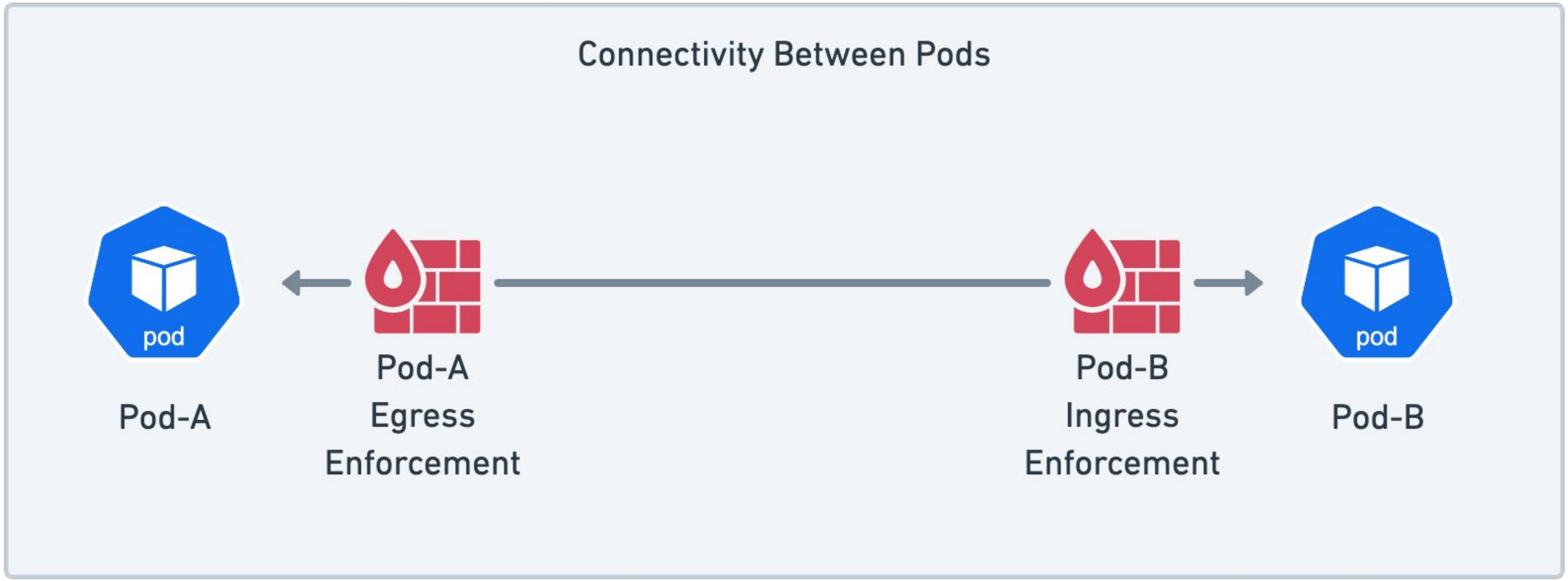
Endpoint Selection

- Each Policy exists in a namespace.
- Endpoint selectors match pods within that namespace.
- A policy matches 0 or more pods in a namespace based on a label-match:
  - PodSelector field in NetworkPolicy
  - EndpointSelector field in CiliumNetworkPolicy
- Subset Match:
  - Pod must have the labels specified in ‘matchLabels’, but may have other labels as well.
- Each rule is applied to all pods that match.

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l3-rule"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  ingress:
  - fromEndpoints:
    - matchLabels:
        role: frontend
```

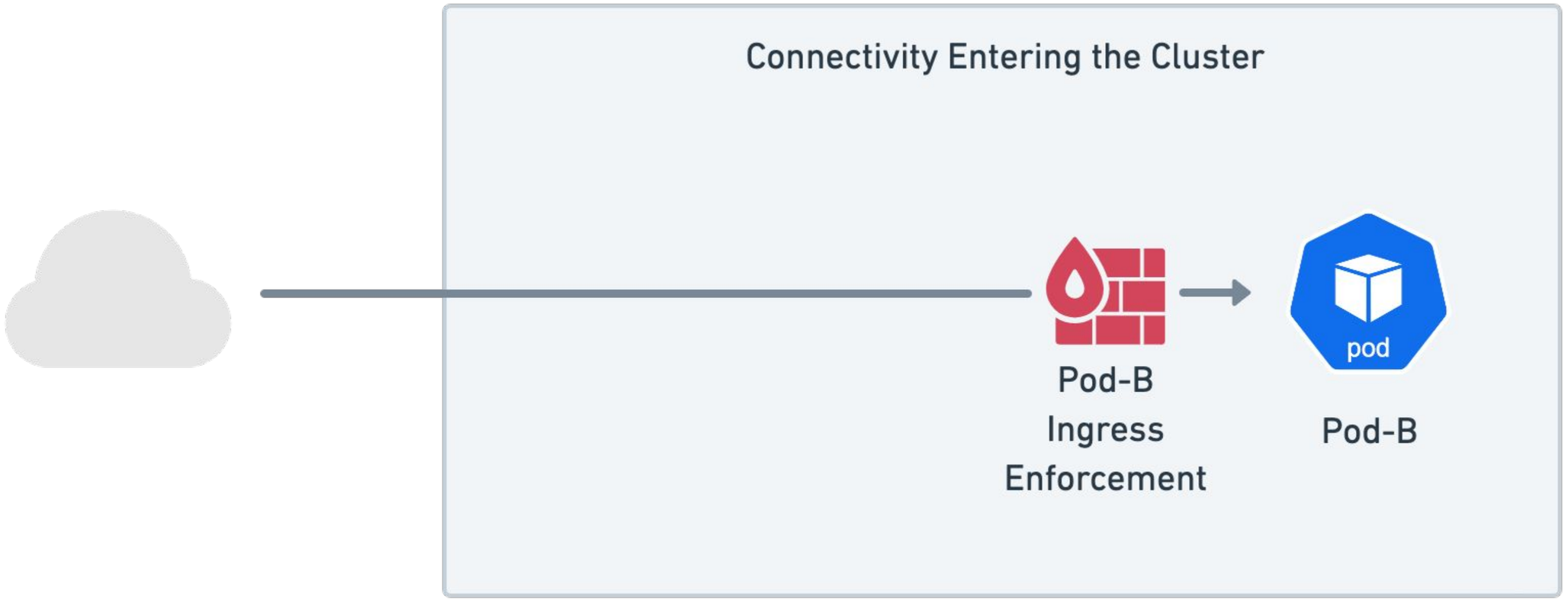
Core Concepts

Ingress & Egress Rules



Core Concepts

Ingress & Egress Rules



Core Concepts

Ingress & Egress Rules

Core Concepts

Ingress & Egress Rules

Core Concepts

Ingress & Egress Rules

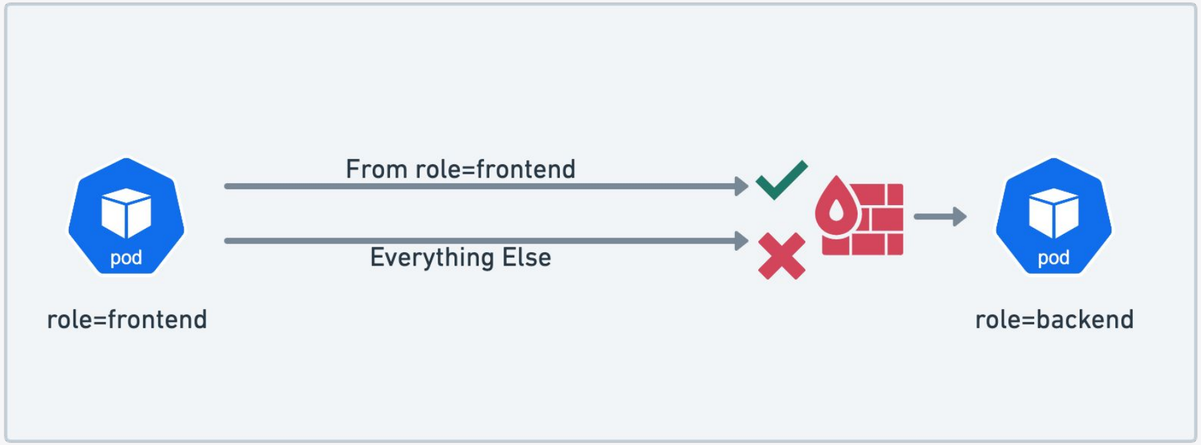
- Ingress Rules: traffic entering a pod
- Egress Rules: traffic leaving a pod.
- Policies may have any combination of ingress & egress rules.

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l3-rule"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  ingress:
  - fromEndpoints:
    - matchLabels:
        role: frontend
egress:
 - toEntities:
   - world
```

Core Concepts

Default Deny Model

- Each rule describes an allowed type of connectivity.
- Total allowed connectivity is the logical OR of all rules applied to a pod.
- Connectivity not explicitly allowed is implicitly denied.




```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l3-rule"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  ingress:
  - fromEndpoints:
    - matchLabels:
        role: frontend
```

Core Concepts

Automatically Allowed Connections

If NetworkPolicy is default deny, why is traffic forwarded in a cluster with no policies defined?

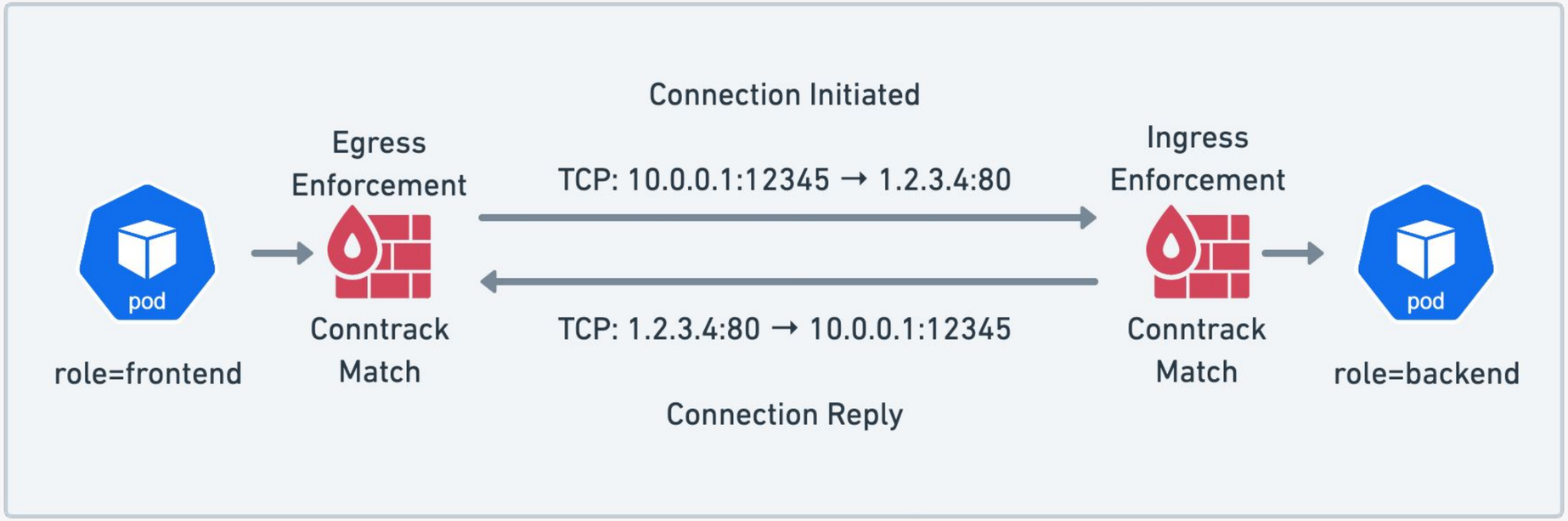
- Default deny model does NOT apply:
  - On ingress for a pod if no ingress rules matches the pod.
  - On egress for a pod if no egress rules matches the pod.

- All ingress connectivity from K8s worker nodes is automatically allowed.
  - Avoid unintentional impact on network liveness/readiness probes from kubelet.
  - Allows connectivity from host-network pods. Host-network == trusted.

Core Concepts

Stateful Connection Tracking

- Connection reply traffic is automatically allowed.
- Applies for TCP & UDP connections.



Core Concepts

Policy Layers

- L3: Network Endpoints:
  - toEndpoints / fromEndpoints
  - toCIDR / fromCIDR
  - toFQDN
- L4:
  - toPort / fromPort:
- L7:
  - DNS
  - HTTP, gPRC
  - Kafka, Memcache, Cassandra

Policy Definition

Policy Layers

Within a rule, each layer is logically AND with the other layers.

Unspecified layers are wildcards.

L3-only
(all L4 ports)

```
ingress:
  - fromEndpoints:
    - matchLabels:
      role: frontend
```

L4-only
(all L3 remote endpoints)

```
ingress:
- toPorts:
  - ports:
      - port: "80"
        protocol: TCP
```

L3 + L4

```
ingress:
- fromEndpoints:
  - matchLabels:
      role: frontend
  toPorts:
  - ports:
      - port: "80"
        protocol: TCP
```

Policy Definition

Multiple Rules per Policy

2 Rules (logical OR)

```
ingress:
- fromEndpoints:
  - matchLabels:
      role: frontend
- toPorts:
  - ports:
    - port: "80"
      protocol: TCP
```

1 Rule (logical AND)

```
ingress:
- fromEndpoints:
  - matchLabels:
      role: frontend
  toPorts:
  - ports:
    - port: "80"
      protocol: TCP
```

Policy Definition

Advanced Endpoint Selectors

Multi-label Match
Pod must have all labels to match
(logical AND)

```
spec:
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  egress:
...
```

Wildcards
Match all endpoints
in namespace

```
spec:
  endpointSelector: {}
  egress:
...
```

Set-Based Label Selectors
More complex
more powerful matching

```
spec:
  endpointSelector:
    matchExpressions:
    - key: k8s-app
      Operator: NotIn
      Values:
      - app1
      - app2
  egress:
...
```

More Details: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Policy Definition

toEndpoints/fromEndpoints

- Subset match: pod must have all labels specified in ‘matchLabels’ in order to be allowed by the rule, but may have additional labels as well.
- Namespaces are the key consideration when matching use labels.

Implicit Local Namespace Binding

```
...
  ingress:
  - fromEndpoints:
    - matchLabels:
        role: frontend
```

Explicit Namespace Binding

```
...
  ingress:
  - fromEndpoints:
    - matchLabels:
        role: frontend
        k8s:io.kubernetes.pod.namespace: foo
```

Policy Definition

toEndpoints/fromEndpoints

Match all endpoints in local namespace

```
...
  ingress:
  - fromEndpoints:
    - matchLabels: {}
```

Wildcarding Namespace

```
...
  ingress:
  - fromEndpoints:
    - matchExpressions:
      - key: k8s:io.kubernetes.pod.namespace
        operator: Exists
      - key: app
         operator: In
         values: [public-service]
```

Policy Definition

toCIDR/fromCIDR and toCIDRSet/fromCIDRSet

```
...
  egress:
  - toCIDR:
    - 10.0.0.0/8
```

```
...
  egress:
  - toCIDRSet:
    - cidr: 0.0.0.0/0
      except:
      - 169.254.169.254/32
```

NOTE: CIDR and CIDR-based policies apply ONLY to IPs that are “outside” the cluster. Do not use them to try and match on Pod, Service, or Worker Node IPs.

Policy Definition

toFQDN L3 Policies

Similar to toCIDR, but automatically translated from DNS names.

```
...
  egress:
  - toFQDNs:
    - matchName: "www.domain.net"
```

Allows regex-based matching.

```
...
  egress:
  - toFQDNs:
    - matchPattern: "*.company.com"
```

Policy Definition

toFQDNs DNS proxy requirement

- toFQDNs rules can only be applied to pods that also have a DNS proxy rule that intercepts all DNS lookups from that pod.
- For security reasons, this rule should be limited to DNS servers that you trust.

```
...
   - toEndpoints:
      - matchLabels:
          io.kubernetes.pod.namespace: kube-system
          k8s-app: kube-dns
      toPorts:
        - ports:
           - port: "53"
             protocol: ANY
          rules:
            dns:
              - matchPattern: "*"
```

Policy Definition

toEntities/fromEntities + Reserved L3 Identities

Special reserved **identities** that correspond to a group of **endpoints**:

- world: all endpoints outside the cluster
- cluster: all managed pod endpoints
- host: the LOCAL k8s worker node.
- remote-node: non-local k8s worker nodes.

```
...
  ingress:
  - fromEntities:
    - world
```

```
...
  egress:
  - toEntities:
    - cluster
    - host
    - remote-node
```

Policy Definition

toPorts

L4-only policies implicitly allow from all L3 entities

```
...
  ingress:
  - toPorts:
     - ports:
         - port: "80"
           protocol: ANY
```

L3+L4 policies allow only connections that match at both layers.

```
...
  egress:
  - toPorts:
     - ports:
         - port: "80"
           protocol: ANY
    fromEndpoints:
     - matchLabels: {}
```

Policy Definition

L7 Policy Example (HTTP)

L4+L7 policies implicitly allow from all L3 entities

```
...
  ingress:
  - toPorts:
     - ports:
         - port: "80"
           protocol: TCP
       rules:
         http:
         - method: GET
           path: “/path”
```

L3+L4+L7 policies allow only connections that match at both layers.

```
...
  egress:
  - fromEndpoints:
      - matchLabels:
        app: frontend
    toPorts:
     - ports:
         - port: "80"
           protocol: TCP
       rules:
         http:
         - method: GET
           path: “/path”
```

Common Patterns

Default Deny Policy

- Select all endpoints in namespace, specifying only an “empty” rule for ingress + egress.
- Subverts the “automatic allow” by ensuring that all endpoints have at least one ingress + egress rule, but doesn’t permit any traffic.
- Can be installed by default in a namespace, requiring app team to add explicit “allow” exceptions.

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "default deny"
spec:
  endpointSelector: {}
  ingress:
  - {}
 egress:
 - {}
```

Common Patterns

Default DNS Proxy Rule

After a default deny it is recommended tp Apply DNS proxy rule per namespace or as Cilium ClusterWide Network Policy to observe all DNS lookups.

```
...
   - toEndpoints:
      - matchLabels:
          io.kubernetes.pod.namespace: kube-system
          k8s-app: kube-dns
      toPorts:
        - ports:
           - port: "53"
             protocol: ANY
          rules:
            dns:
              - matchPattern: "*"
```

Common Patterns

Baseline Policies

- Puts standard “baseline” security posture in place for most/all namespaces.
- Often applied during namespace creation or using CiliumClusterwideNetworkPolicy.
- App teams define additional “allowed” exception on a per-namespace basis.
- Security team can monitor/approve these additional allow rules.

```
spec:
  endpointSelector: {}
  ingress:
  - fromEndpoints: {}
 - fromEntities:
     - host
     - remote-node
 egress:
- toEndpoints: {}
- toCIDRSet:
       - cidr: 0.0.0.0/0
       except:
       - 169.254.169.254
       - 10.0.2.0/24
```

Common Patterns

L7 Visibility Policies

- Provides visibility into API-layer requests, even if no policy is being enforced.
- For L7 policy, and “empty rule” matches any L7 requests, resulting in a wildcard behavior.
- Visibility policies can be ingress/egress or both.
- Must remember to also have rule to allow all traffic on other ports as well.

```yaml
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "http-ingress-visibility"
spec:
  endpointSelector:{}
  ingress:
  - fromEntities:
    - all
 - toPorts:
     - ports:
         - port: "80"
           protocol: TCP
       rules:
         http:
         - {}
```

Common “Gotchas”

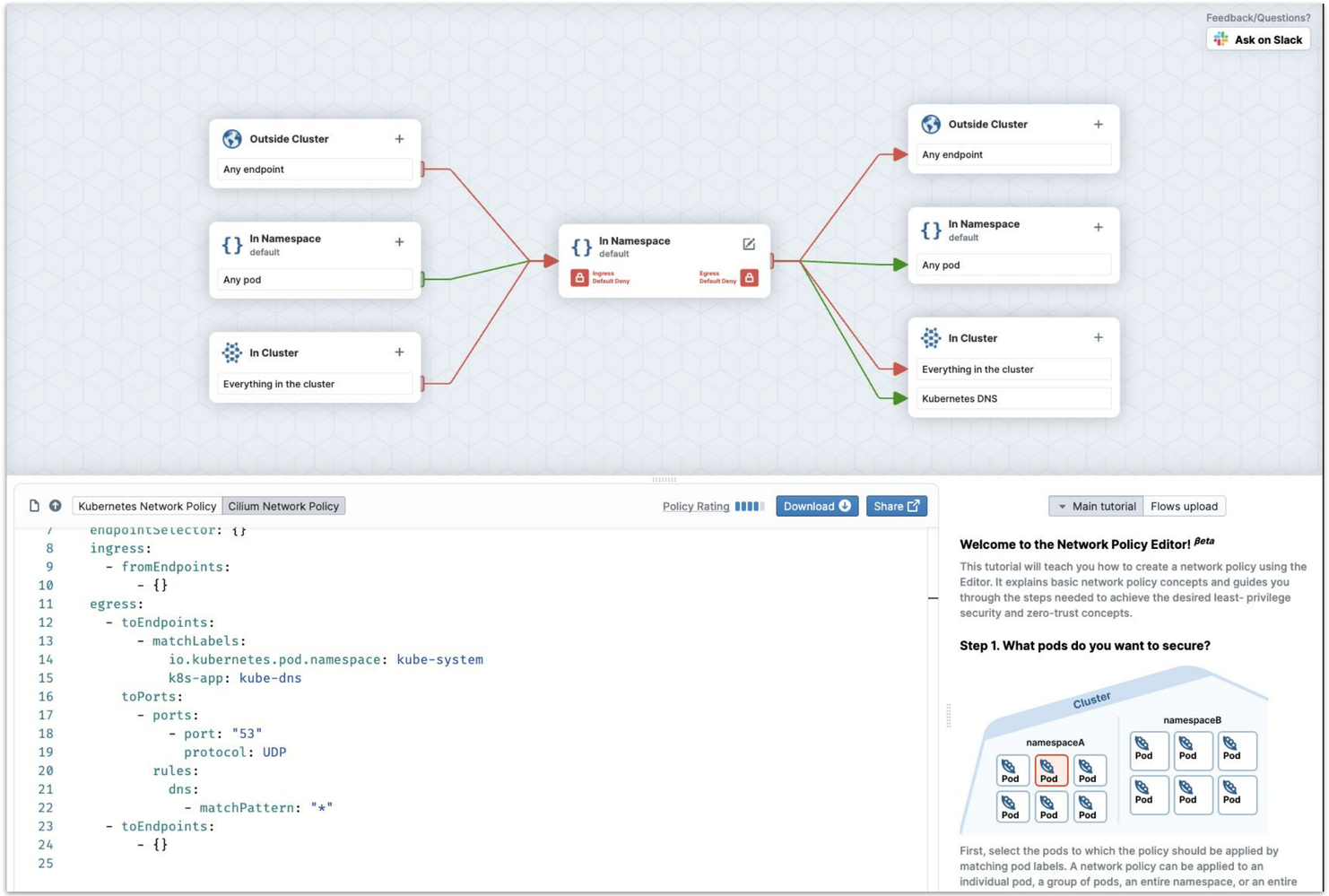
- Enabling egress enforcement rules, but not also allowing connectivity to CoreDNS pods.
- Creating toFQDNs rules without corresponding DNS proxy rule.
- Creating rules based on the label-identity of unmanaged pods.
- Creating rules to allow connections to pods based on IPs.
  - Use `toEndpoints` instead.
- Creating rules to allow connections to K8s worker nodes based on IPs.
  - Use `toEntities` with `host` and `remote-node` instead.
- Creating rules to filter traffic arriving via NodePort/Loadbalancer Services.
  - Using `fromCIDR` requires the use of `externalTrafficPolicy=local` or BPF-based NodePort in DSR mode. Otherwise source IP is obfuscated by NodePort SNAT.
  - Use back-end service port, not NodePort port.
- Creating an FQDN policy for internal k8s services.
- Creating policy for `toCIDR` or `fromCIDR` for pod CIDRs in the cluster.

Overlapping / Conflicting Policies

- Standard network policies all specify what traffic to allow. So order or precedence is not critical here.
- But:
    - Deny policies always take precedence over an allow.
    - Any policy that allows a connection AND has a L7 rules associated will take precedence over a rule that simply allows that traffic at L3/L4, so that L7 inspection occurs. However, if the L7 match in that rule does not match the connection, the connection will still be allowed, due to an automatic L7 wildcard match added due to the L3/L4 rule.

Tools

https://editor.cilium.io



Strategies for Designing Network Policies

Challenges for Adopting Network Policies

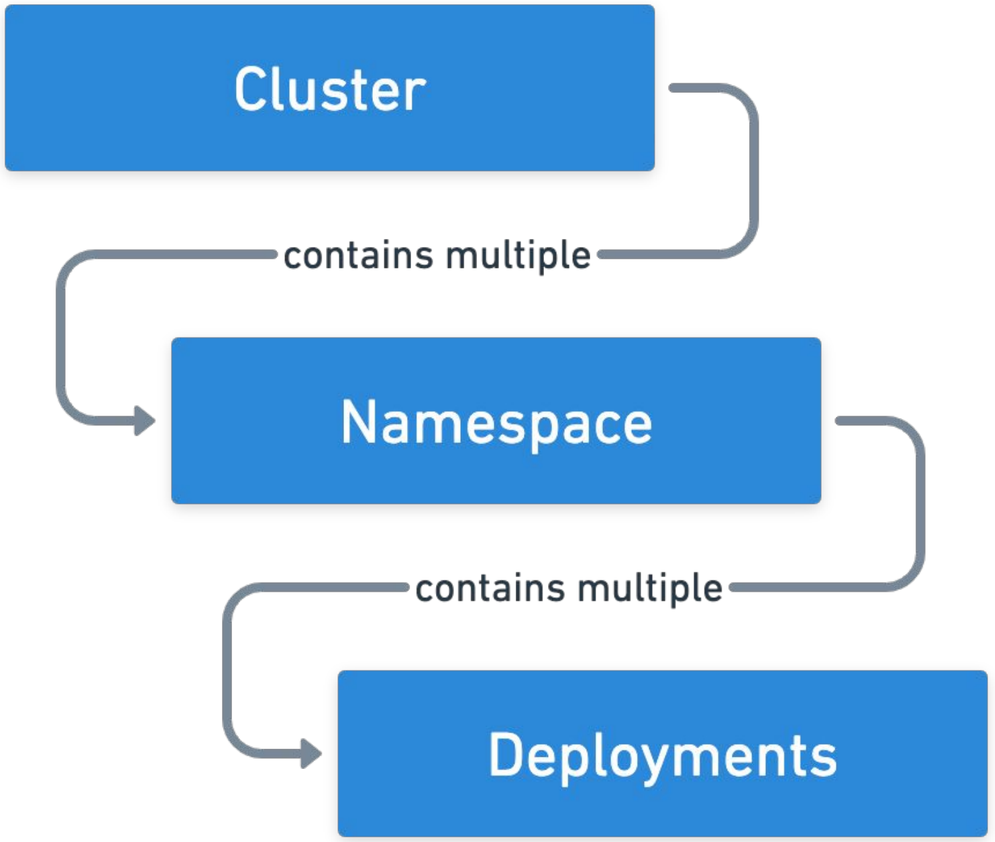
- Where to start? What policies reduce risk with least friction?
- How to troubleshoot Network Policies?
- How to keep Network Policies up-to-date as applications evolve?
- How to prevent applications teams from simply “allowing everything”
- How can security teams prove “default deny” policies are always enforced?
- How can security teams alert 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.
  - However, trying to enumerate all valid connectivity, with high chance of misconfiguration will result in application unavailability and high 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 and reduce granularity of wildcard rules.

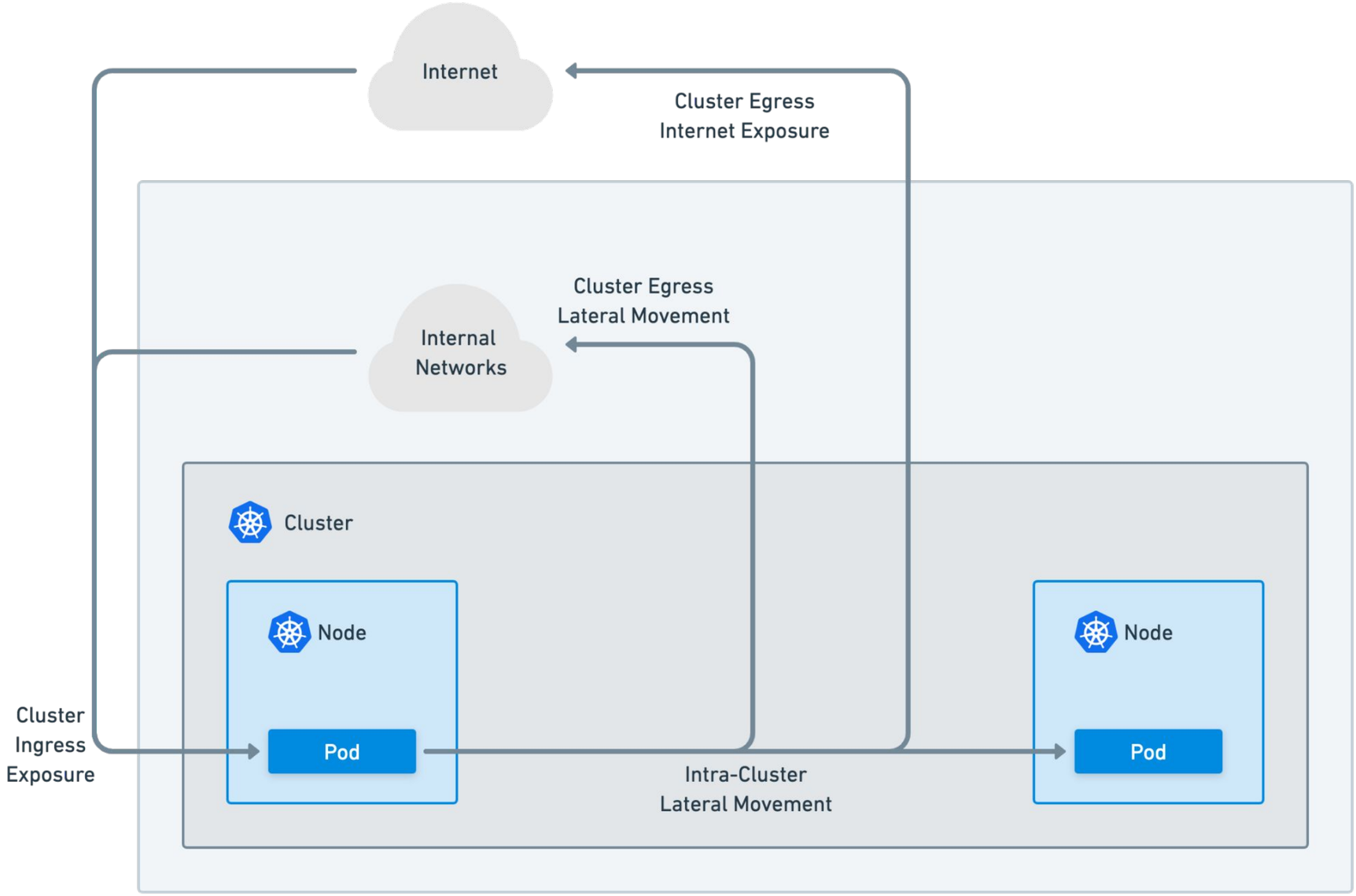
Application Teams & Multi-tenancy

- 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



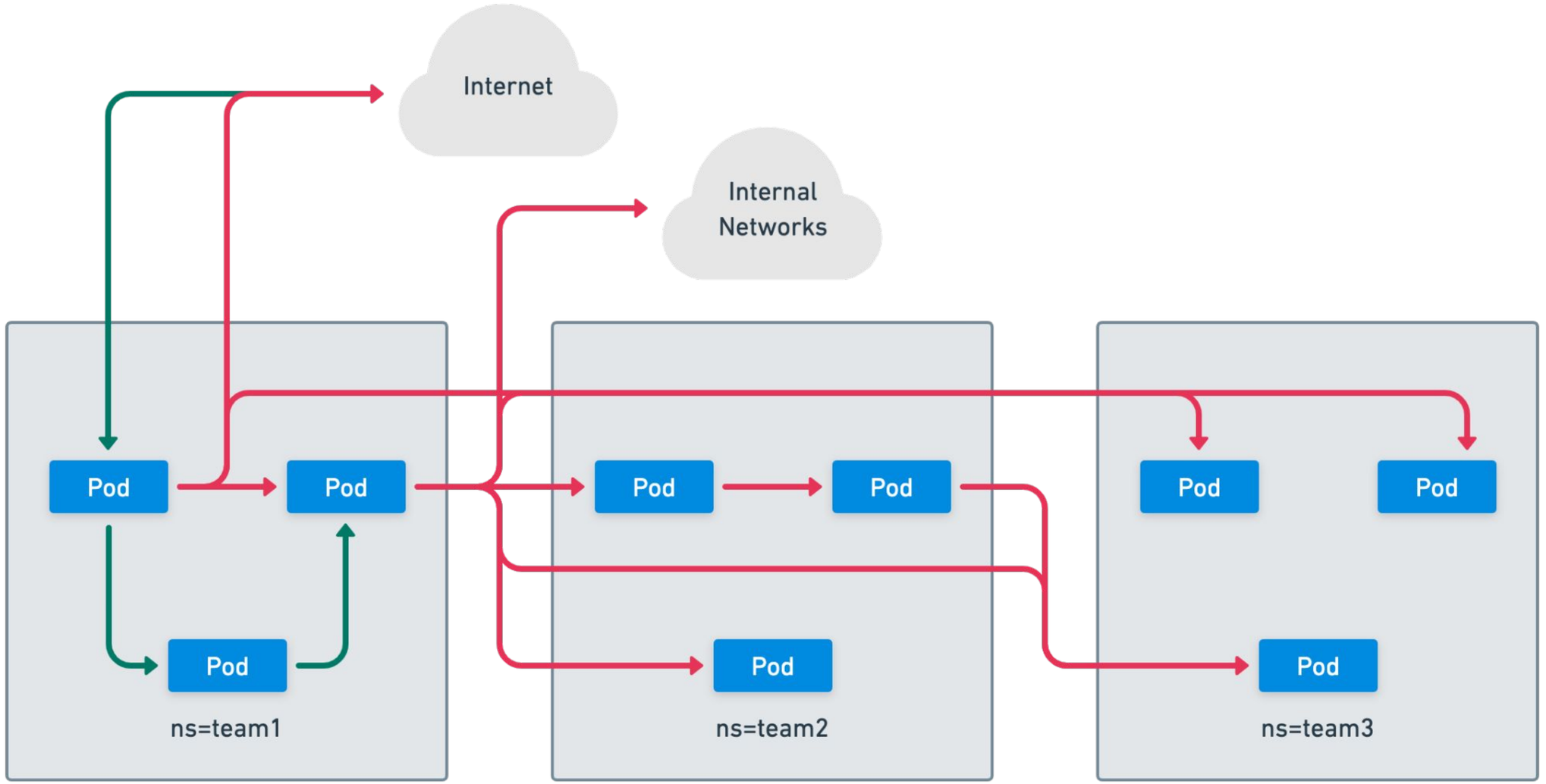
Examples of each Exposure Type

Exposure Type	Example
Cluster Ingress	• Service Exposed to the Internet • Service Exposed to the External Network • Overlay Routing: Internal service un-intentionally exposed as Type LoadBalancer or NodePort rather than ClusterIP. • Direct Routing: All pods are reachable from other workloads that can reach worker nodes.
Intra-Cluster Lateral Movement	By default, any pod to pod connectivity within a cluster is allowed.
Cluster Egress Lateral Movement	Egress connections to other VMs/bare-metal servers within corporate network, as well as cloud metadata, K8s worker nodes, K8s API server.
Cluster Egress to Internet	Access to Internet destinations for data exfiltration, access to command & control servers (C2), cryptomining, etc.

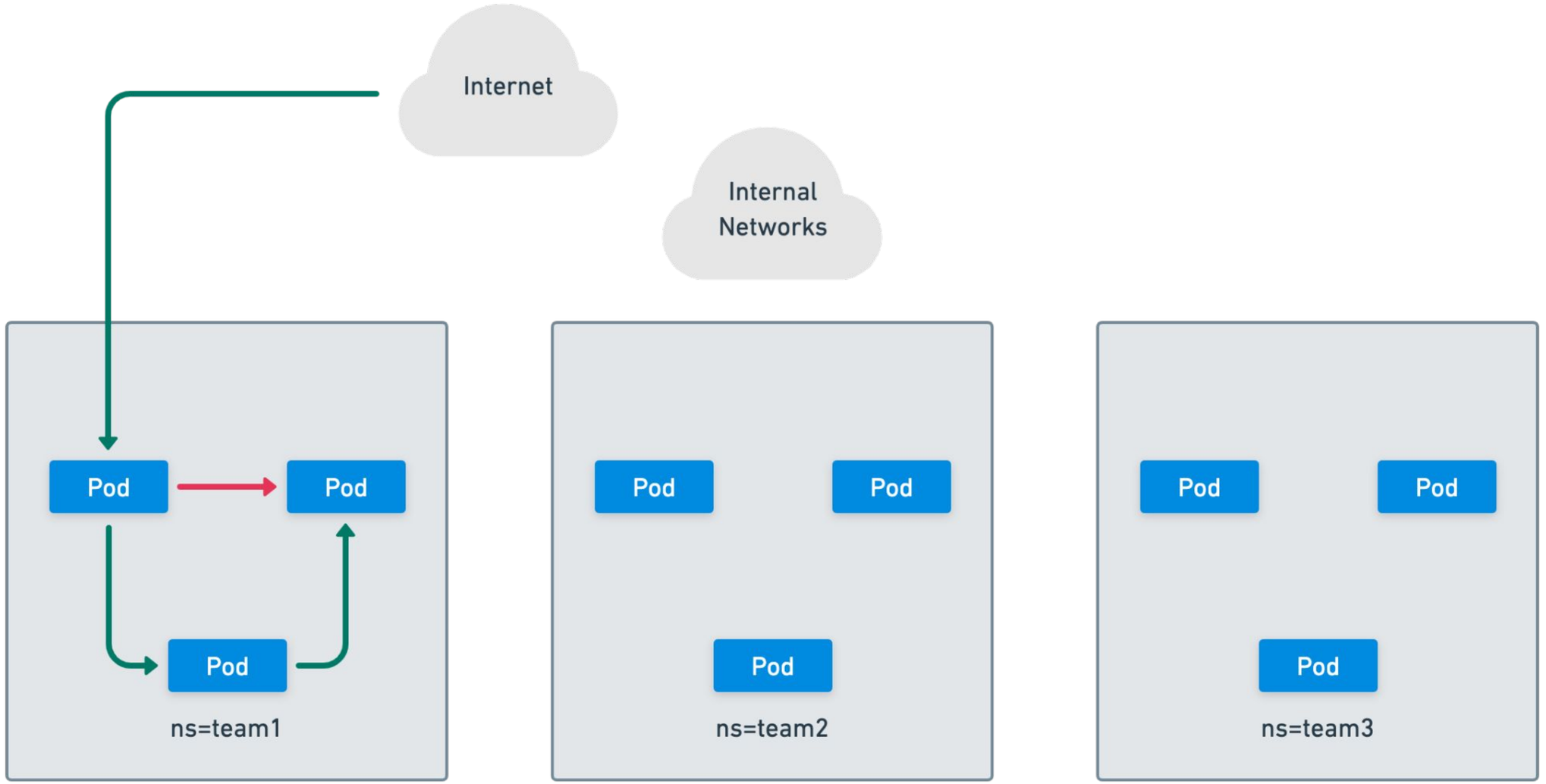
Minimizing Risk: Reduce Unused Access

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 coarse-grained 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

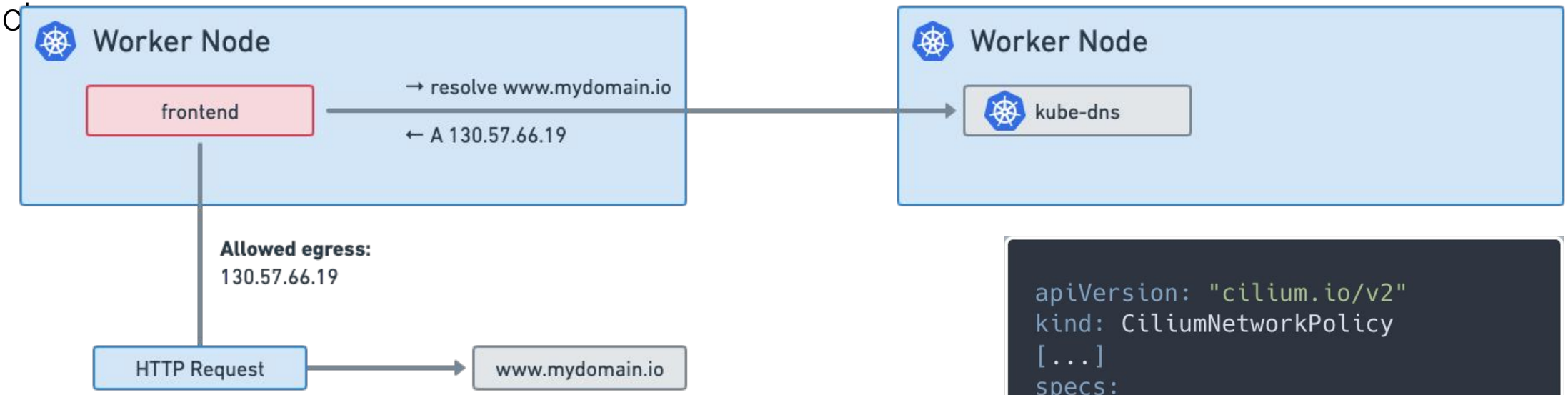
- # of services reachable via Cluster Ingress with no matching traffic.
- # of services reachable from other namespaces with no matching traffic.
- # of services with access to External Networks or Internet with no matching connections.

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.

Best Practices - DNS Rules

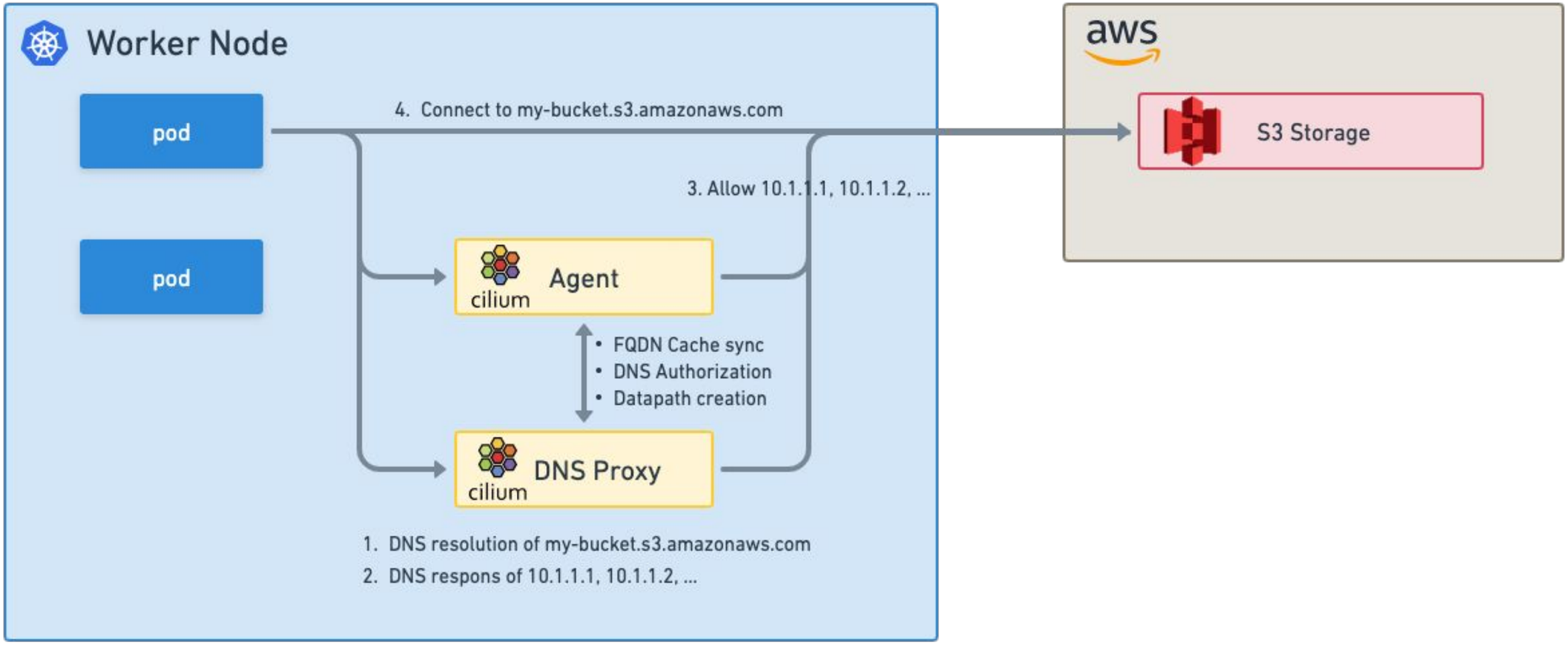
DNS-aware Cilium Network Policies for External Services which IPs frequently c




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

Best Practices - DNS Proxy HA

Solution for Enterprises with Mission Critical External Services



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 namespace.
  - 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

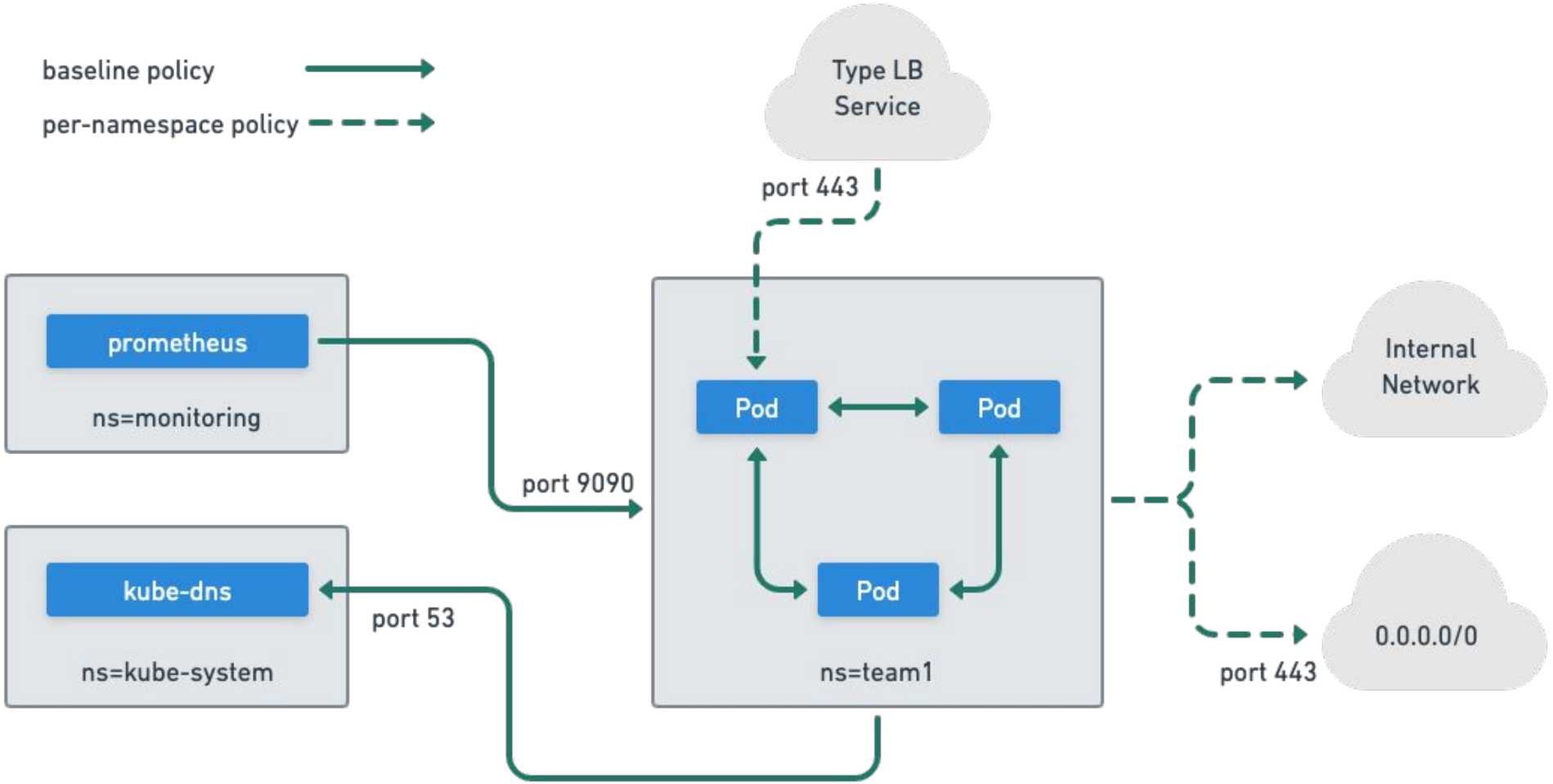
Global Baseline Policies

- 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 or Load-Balancer/NodePort.
- Namespace-specific egress:
  - No access.
  - Egress to specific CIDR/DNS + port
  - Unrestricted access on specific ports (fallback).

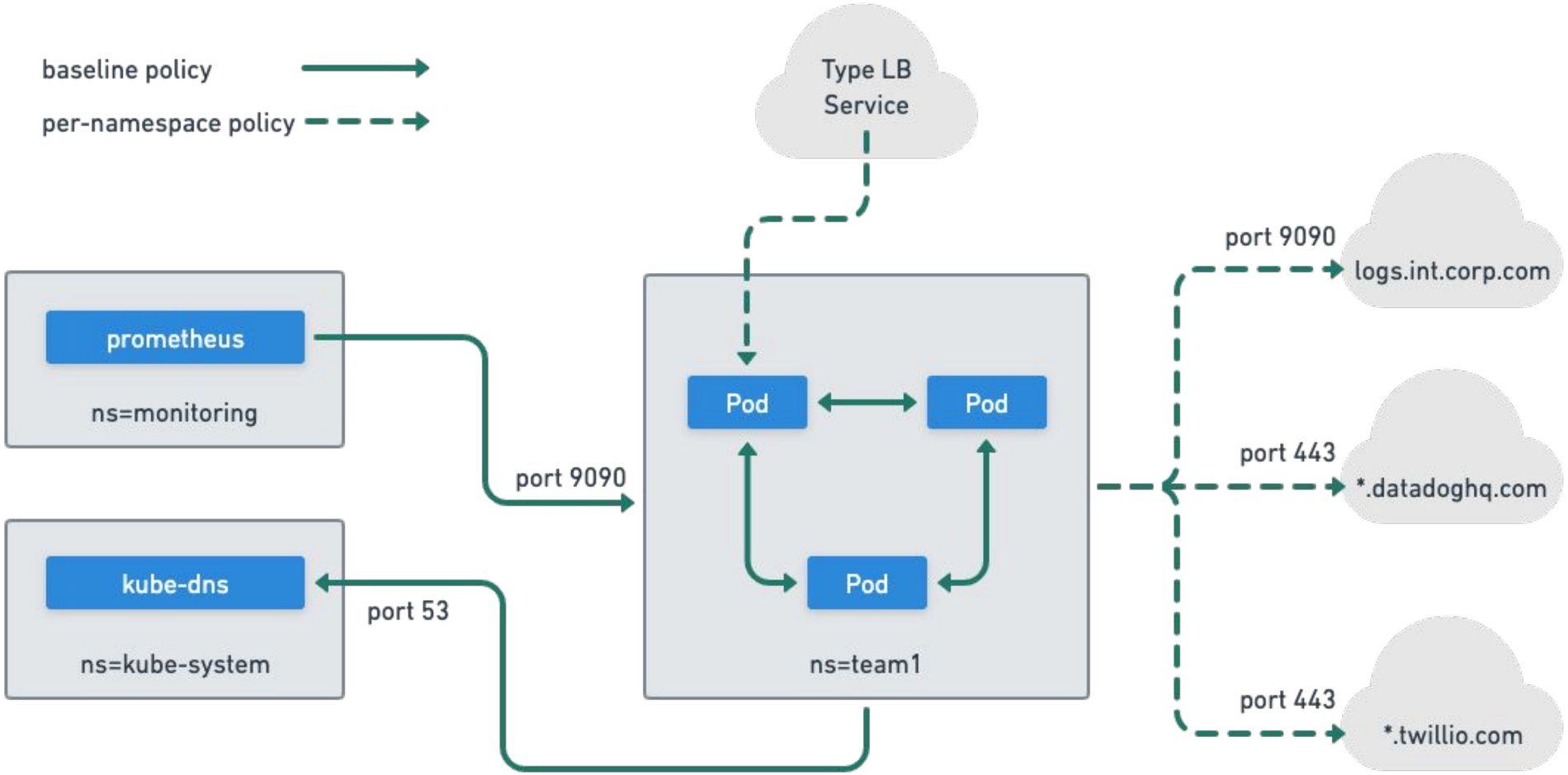
Example Baseline + Coarse-grained Per Namespace Policies



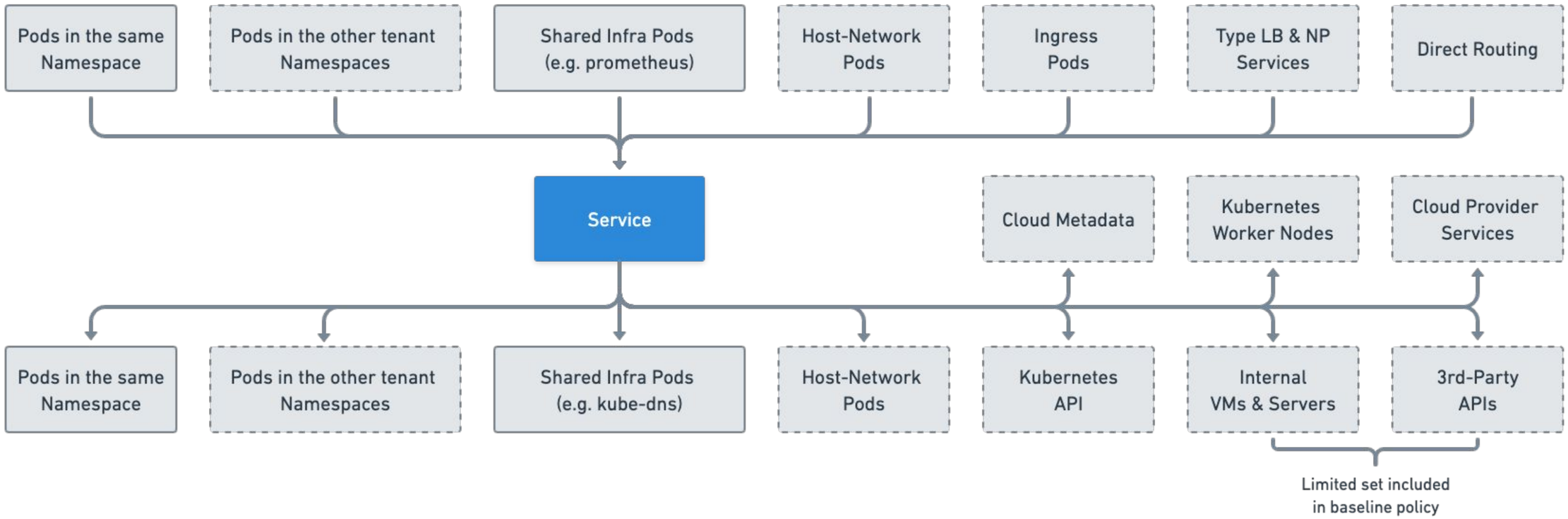
Transitioning from “Coarse” to “Fine-Grained” Policies

- 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

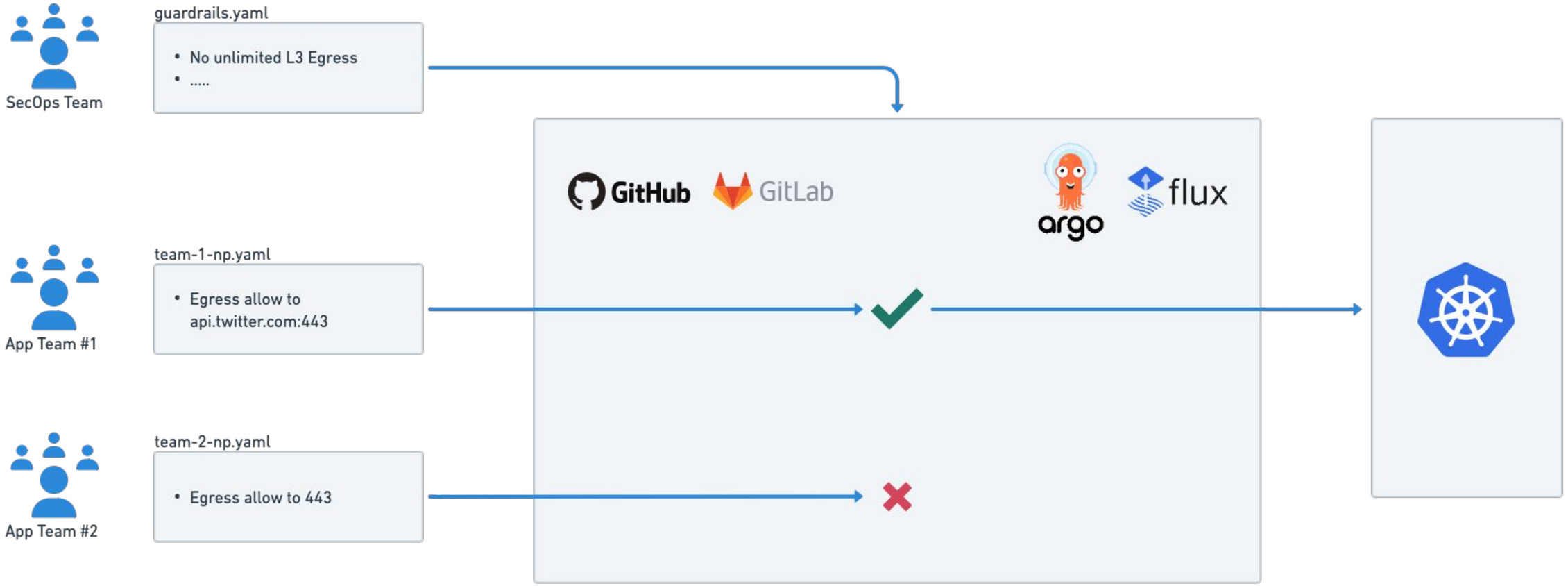


Delegating Network Policy Creation

- At scale, Platform teams cannot manage policies for each service:
  - Solution: delegate per-namespace policies to app teams.
  - Challenge: how to ensure teams do not simply “allow all” traffic and undermining security properties.
- Several Options:
  - ClusterWide deny policies:
    - Prevents known bad specific behavior e.g. egress to 0.0.0.0/0 on port 22.
    - Does not prevent aggressive wildcarding (e.g., allow egress 0.0.0.0/0 on port 443)
  - Indirect Access:
    - App teams cannot create arbitrary KNP/CNP via K8s API.
    - Use portal, or other indirect mechanism to allow user to accept inputs (e.g., list of DNS names) that populate a KNP/CNP template that is sent to K8s API.
  - Direct Access with Validation:
    - App teams can create their own KNP/CNP, but policies are validated against set of acceptable rule patterns via tools, gitops checks or webhooks.

Network Policy Guardrails

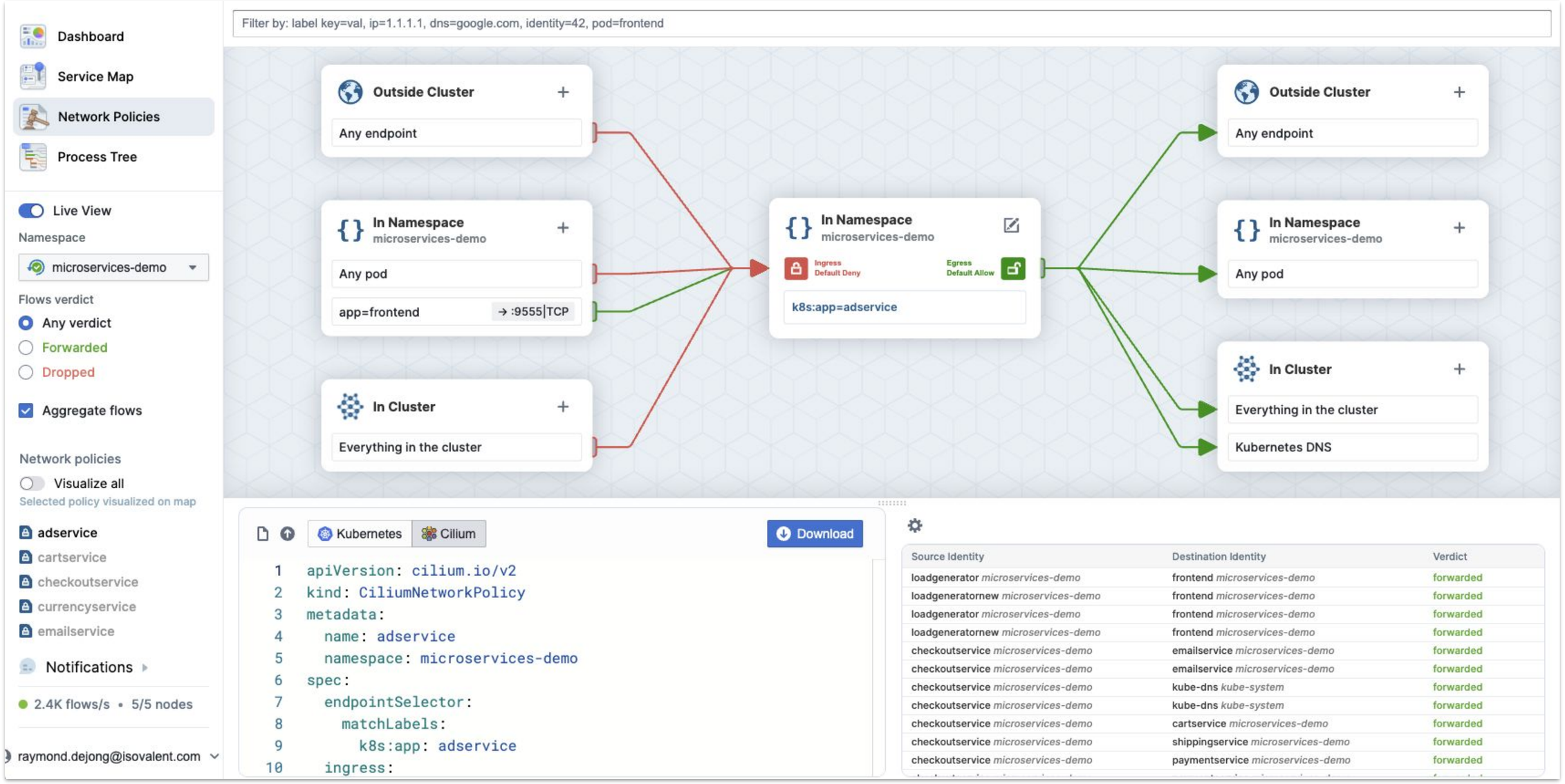
Maintain control while granting application teams self-service management of Network Policy.



GitOps-based Network Policy Approvals or Guardrails

- Example of “Direct Access with Validation” approach:
  - Policies created via Policy Editor or manually.
  - All policies pushed via GitOps-based workflow.
  - Declarative Guardrails config created by SecOps describes acceptable rules.
  - Policies with violations are either rejected or routed for “exception approval”
- Three Primary patterns:
  - Require egress destinations to match a list of “approved” DNS domains or CIDRs.
  - Prevent egress access to a specific sensitive destination or set of destinations. Ex:
    - cloud metadata service.
    - access to k8s worker nodes. .
  - Prevent “aggressive” wildcarding:
    - large CIDR domains (prefix larger than /24)
    - FQDNs with wildcards (*.twitter.com)
    - access to/from entire cluster.
- Be careful: many different ways to write the same “bad” rule. Not as simple as “grepping” for obvious bad rules.

Hubble Enterprise Network Policy Editor



Observability as the Network Policy Superpower

Troubleshooting Network Policies

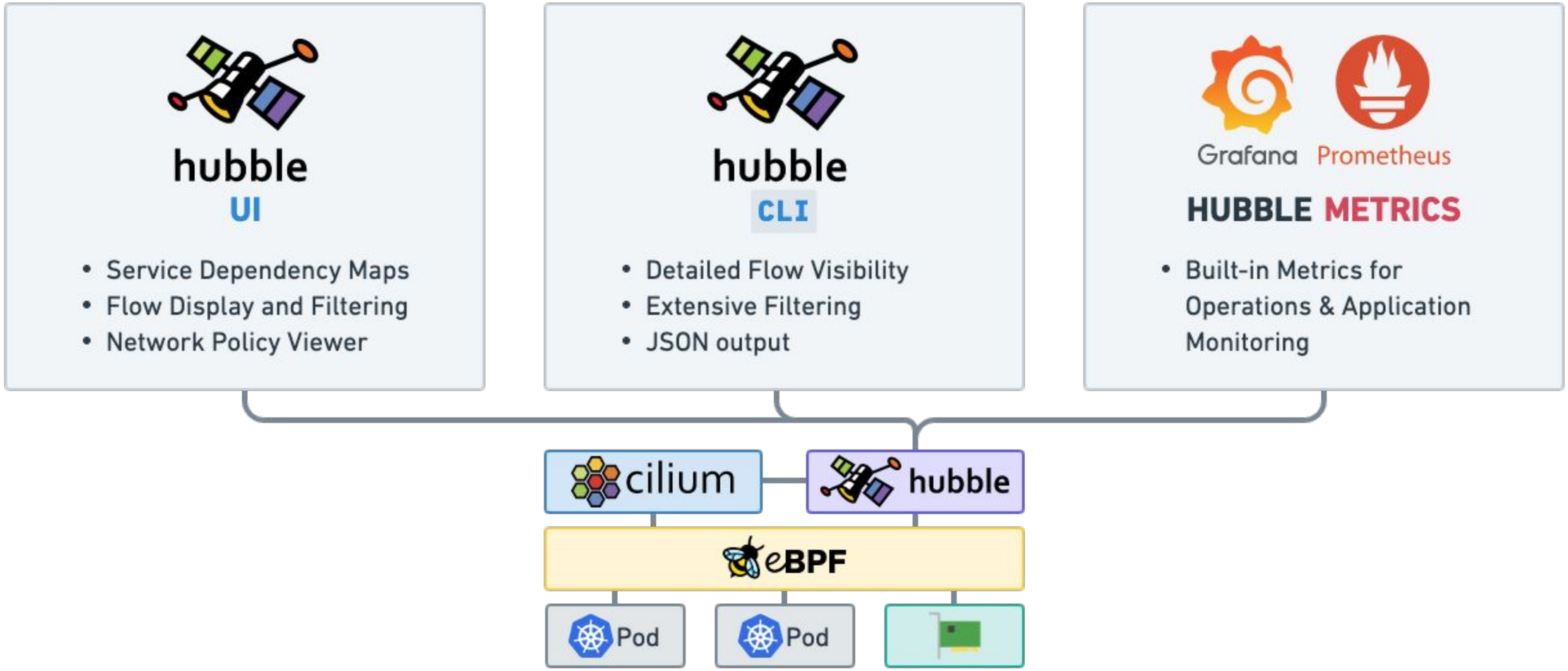
Once enforced, Network Policy is a potential cause of any application outage:

- 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?

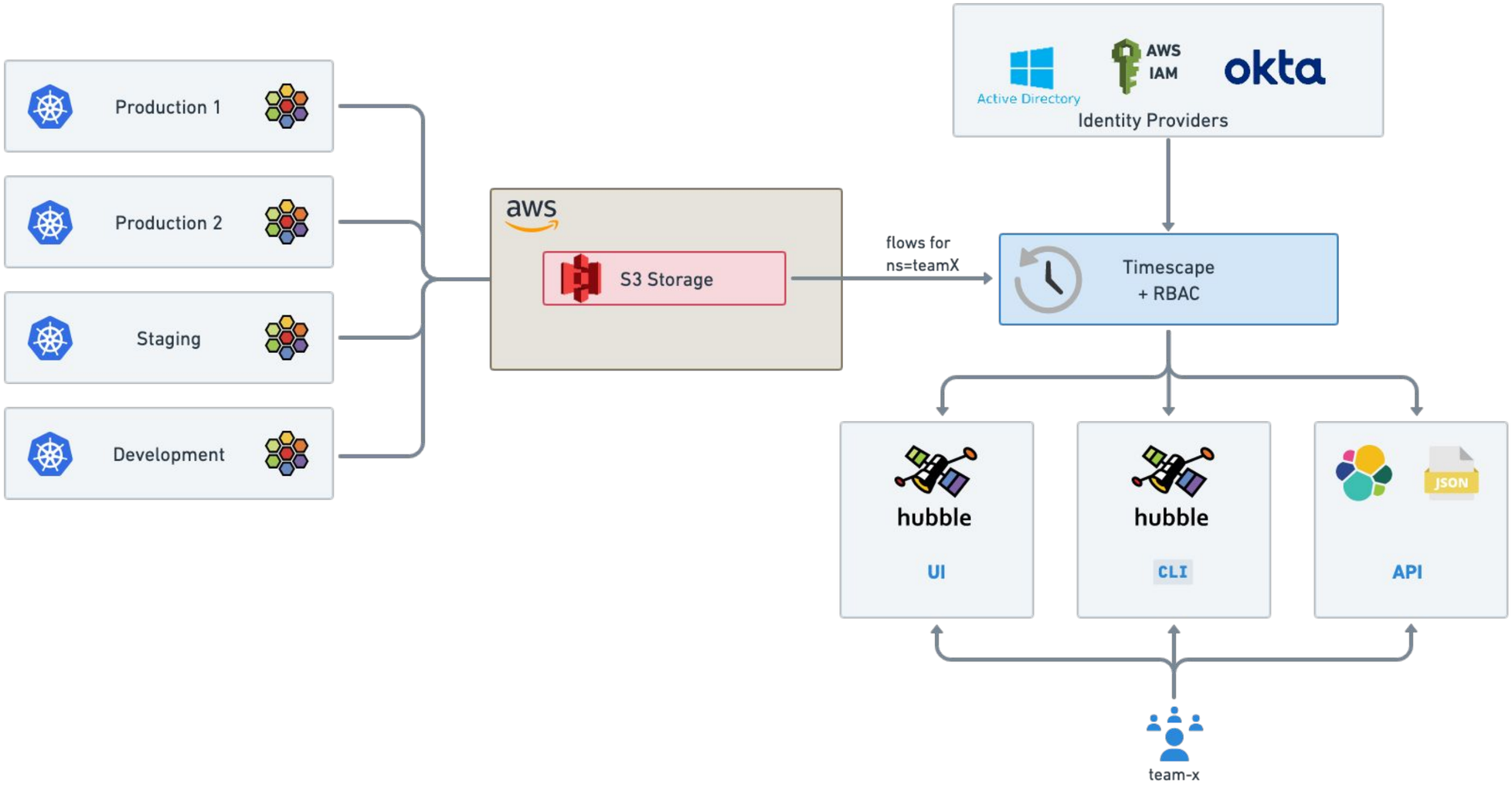
Challenge:

- 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 Troubleshoot Policies



Troubleshoot Network Policy

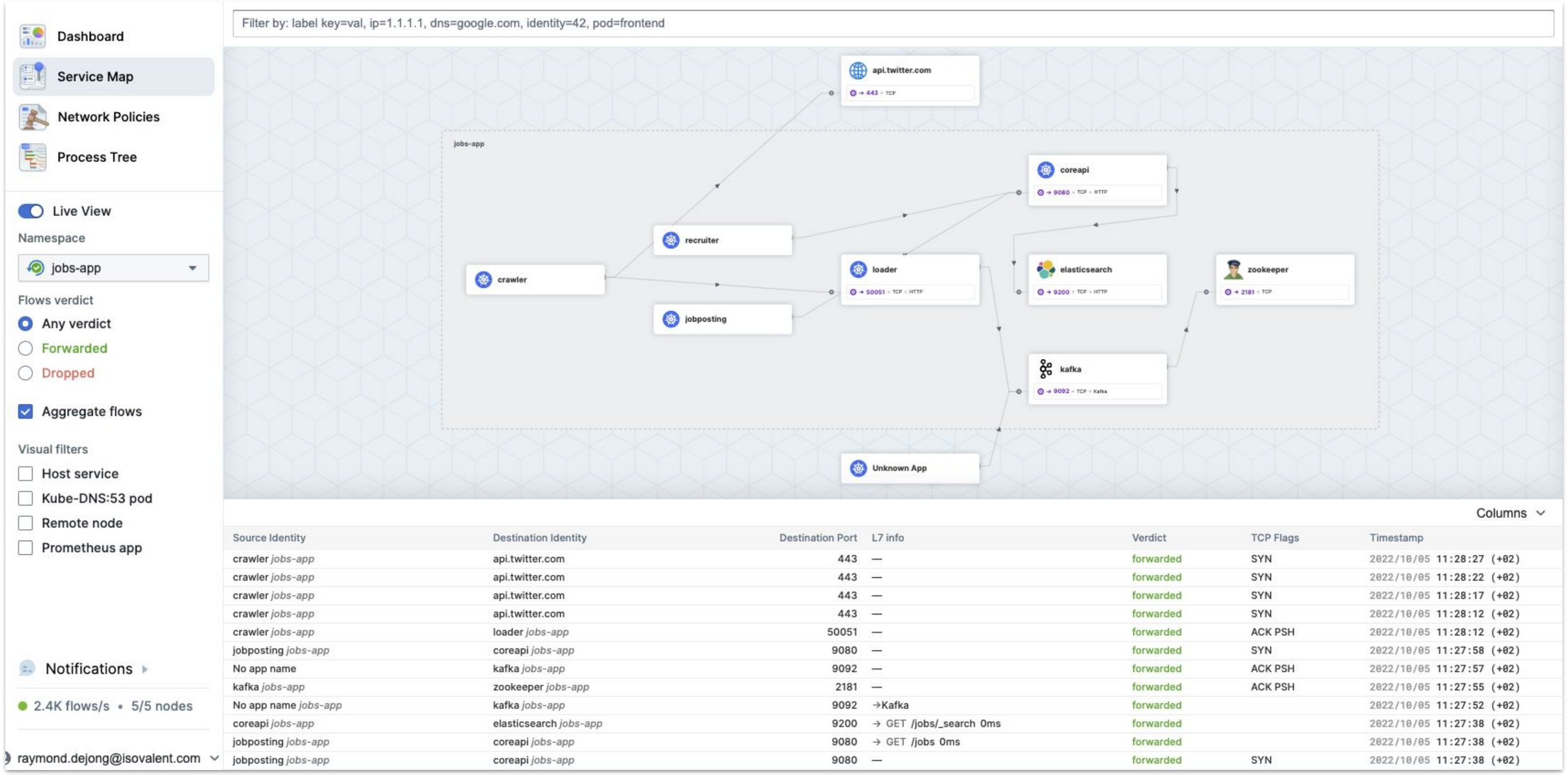
Troubleshooting Data from Hubble Flows:

- Identity labels of source.
- Identity labels of destination
- Verdict.
- “allowed-by” indicatoring policy match (or lack thereof).

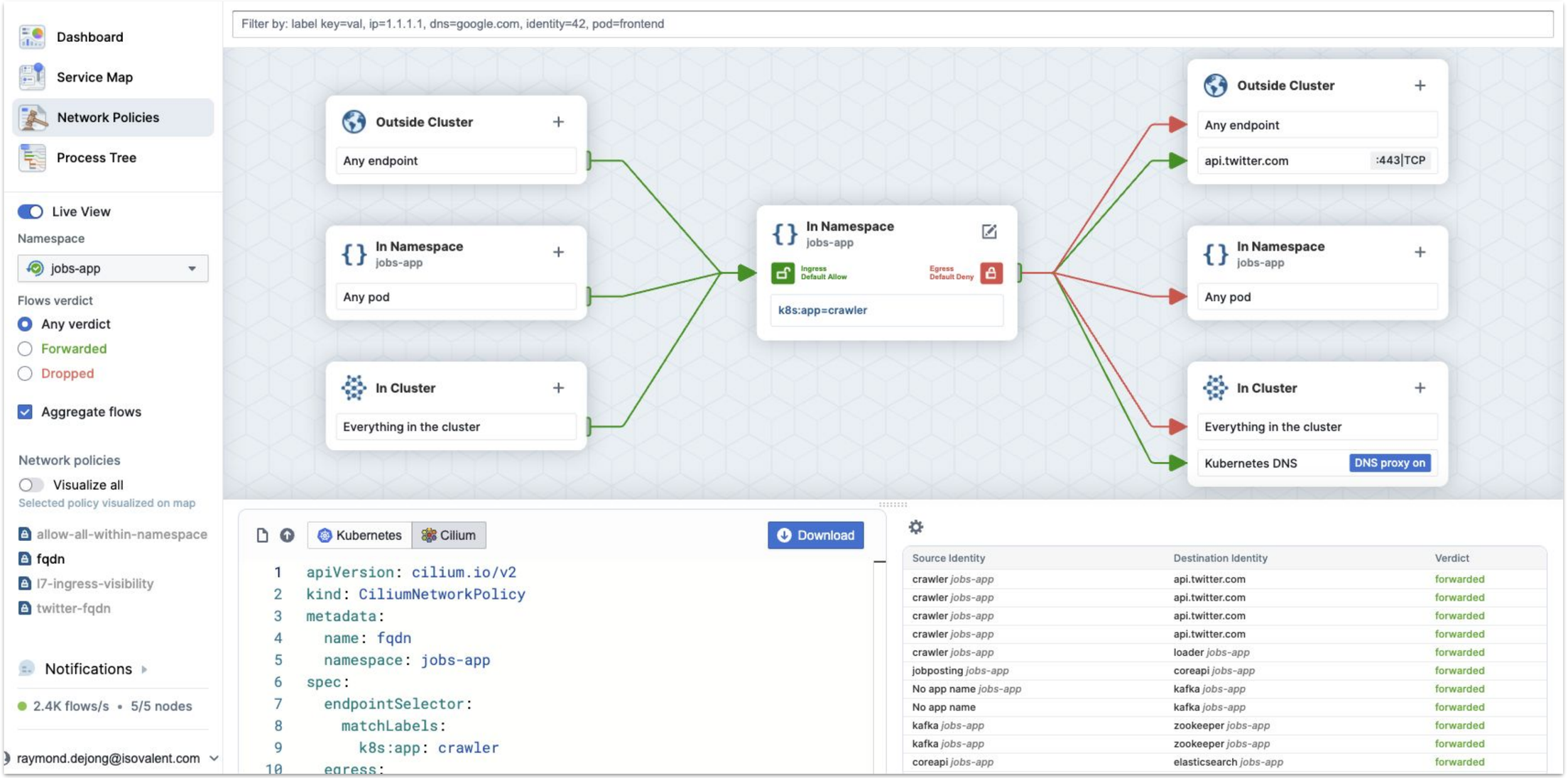
Additional Key Troubleshooting Data:

- Current set of policy rules enforced on both source (egress rules) and destination (ingress rules).
- Date/Time when policy drops started.
- Date/Time of recent changes to policy rules applied to source/destination.
- Date/Time of recent container starts (especially with new image tag) of the source or destination service, which could indicate changed app behavior.
- Date/Time of recent an recent faults on source or destination worker node and associated Cilium instances.

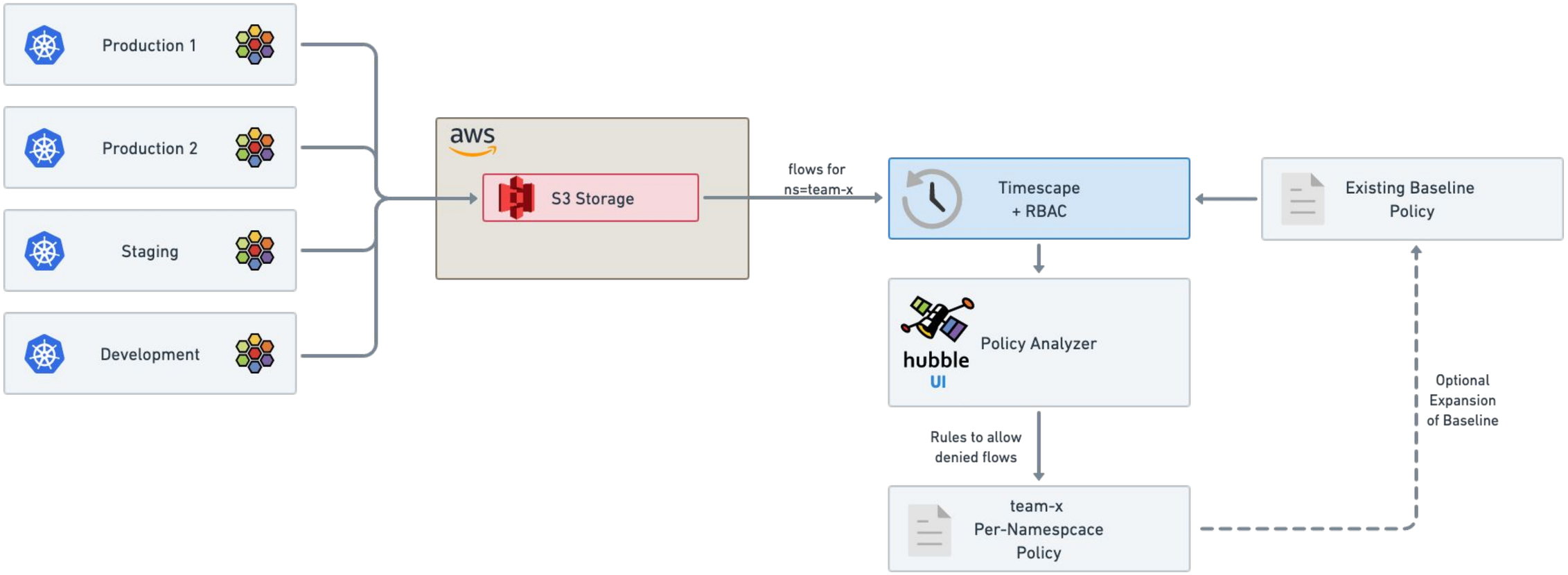
Using Hubble Observability Data to Build Policies



Hubble Network Policy Editor



Using Hubble Observability Data to Build Policies



Using Hubble Observability Data to Build Policies

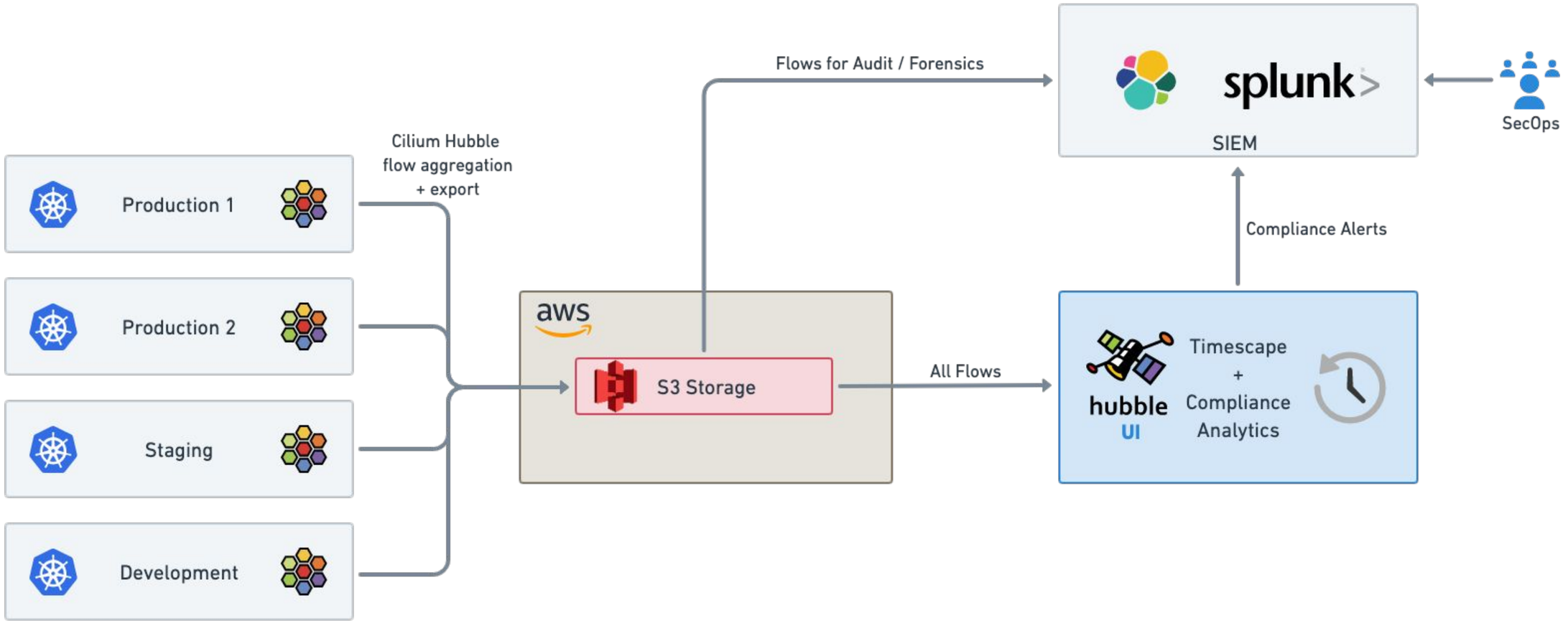
Day 1:

- 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.

Using Hubble Observability Data for Security Compliance



Using Hubble Observability Data for Security Compliance

Requirement to store historical flow data for X days (e.g., 90 days):

- Common requirement to ensure data exists to perform incident investigation or forensics if a potential attack is discovered.
- Data can be stored in S3, or in SIEM like Splunk or ELK.

Prove to security auditors that “default deny” policy has always been in place:

- Compliance analytics alert on any flows that do not have an explicit “allowed-by” field indicating that policy was enforced.

Alerting on network policy drops:

- New sources of denied network connections are potential indicators of compromise.

Alerting on unknown/suspicious destinations as “soft enforcement”:

- Monitor access to “sensitive” addresses (cloud metadata), legacy infra CIDRs if default deny enforcement is not yet “safe”.
- Monitor the set of connections to the public Internet, alerting on unknown/suspicious DNS domains or direct IP access.

Getting it Done!

Transitioning from Observability to Enforcement

Overview

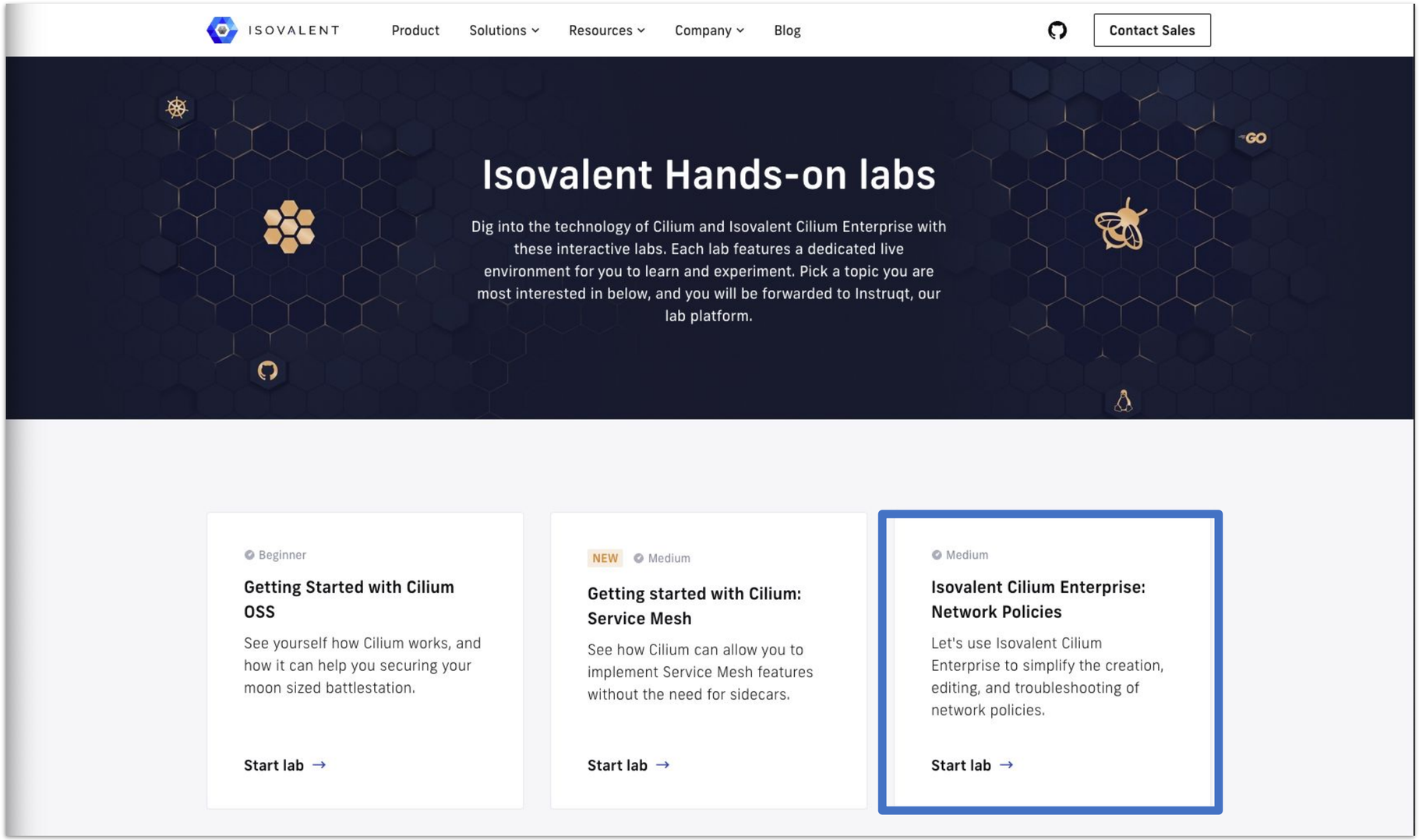
- Step 1: Deploy Namespace with Network Policy
- Step 2: Deploy your Application
- Step 3: Observe Flows with Hubble
- Step 4: Create Required Ingress/Egress Cilium Network Policies
- Step 5: Enforce

Hands-on Lab

Try it out yourself!

Isovalent Hands-on Labs

https://isovalent.com/labs



Learn more!

ISOVALENT

For the Enterprise

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

isovalent.com/product
isovalent.com/labs

OSS Community

eBPF-based Networking, Observability, Security

cilium.io
cilium.slack.com
Regular news

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!