KubeCon | CloudNativeCon

Europe 2019

Kubernetes networking at scale

Bowei Du — Google
Laurent Bernaille — Datadog

Introduction

@lbernail

@boweidu

Scaling Challenges

Scale

- 1k-2k+ nodes clusters
- Dozens of clusters
- 10k-20k pods, ~1k services

Throughput

- Trillions of data points daily
- GB/s

Topology

- Multiple clusters
- VM ⇔ Cluster traffic

Latency

- End-to-end pipeline

Scaling Challenges

Data plane

- Removing inefficiencies

Control plane

- Large N (# of nodes, resources)
- Simplify architecture (dependencies, debugging)

More “native” integration with cloud infrastructure

Scaling Challenges

Let’s see what this means in practice:

- Pod networking
- Service load-balancing
- Ingress (L7)
- DNS

Pod networking

Classic approaches

Static routes

Allocate a CIDR per node
Add CIDR to the VPC route table
Rely on cloud-provider routing

Limits

Number of static routes
Address space efficiency
Pods opaque to infrastructure

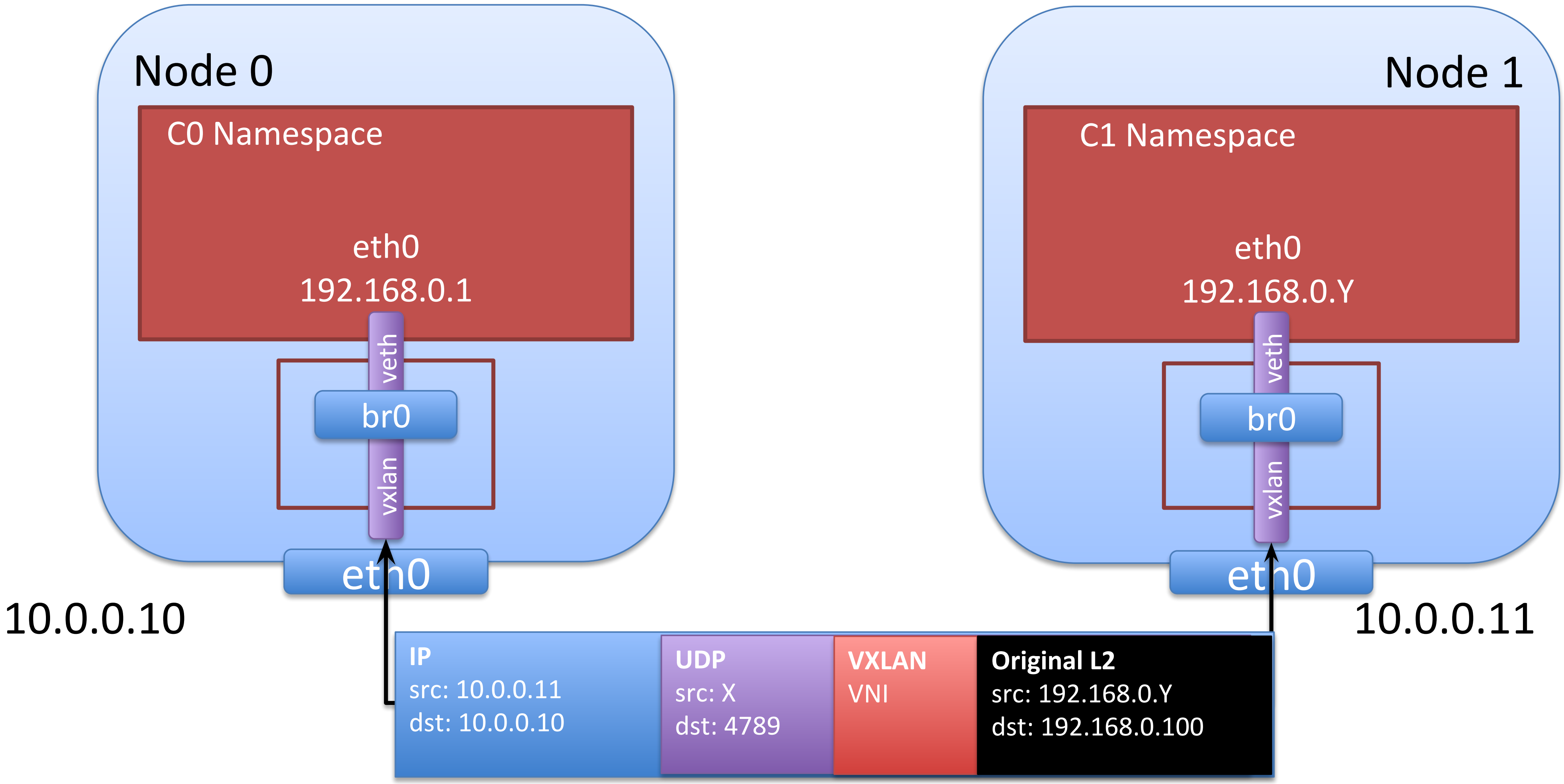
Overlays

Allocate a CIDR per node
Tunnel traffic (IPIP / VXLAN)

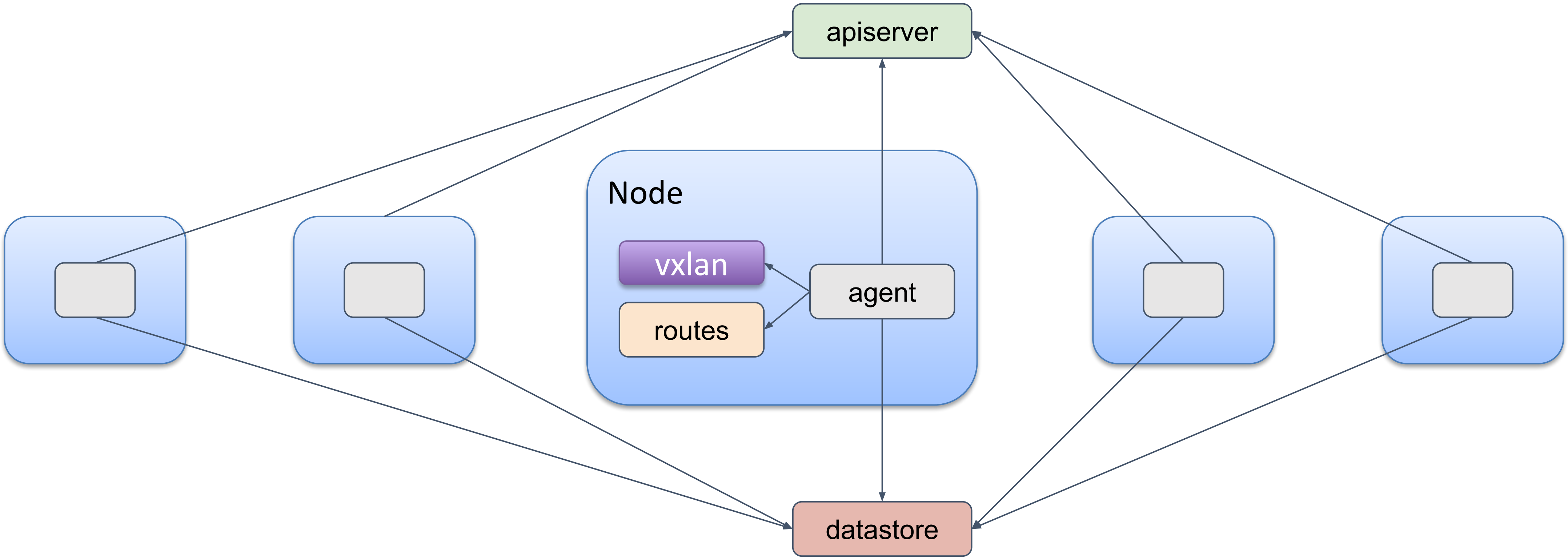
Limits

Tunnel overhead
Route distribution (BGP, custom)

Overlay data plane



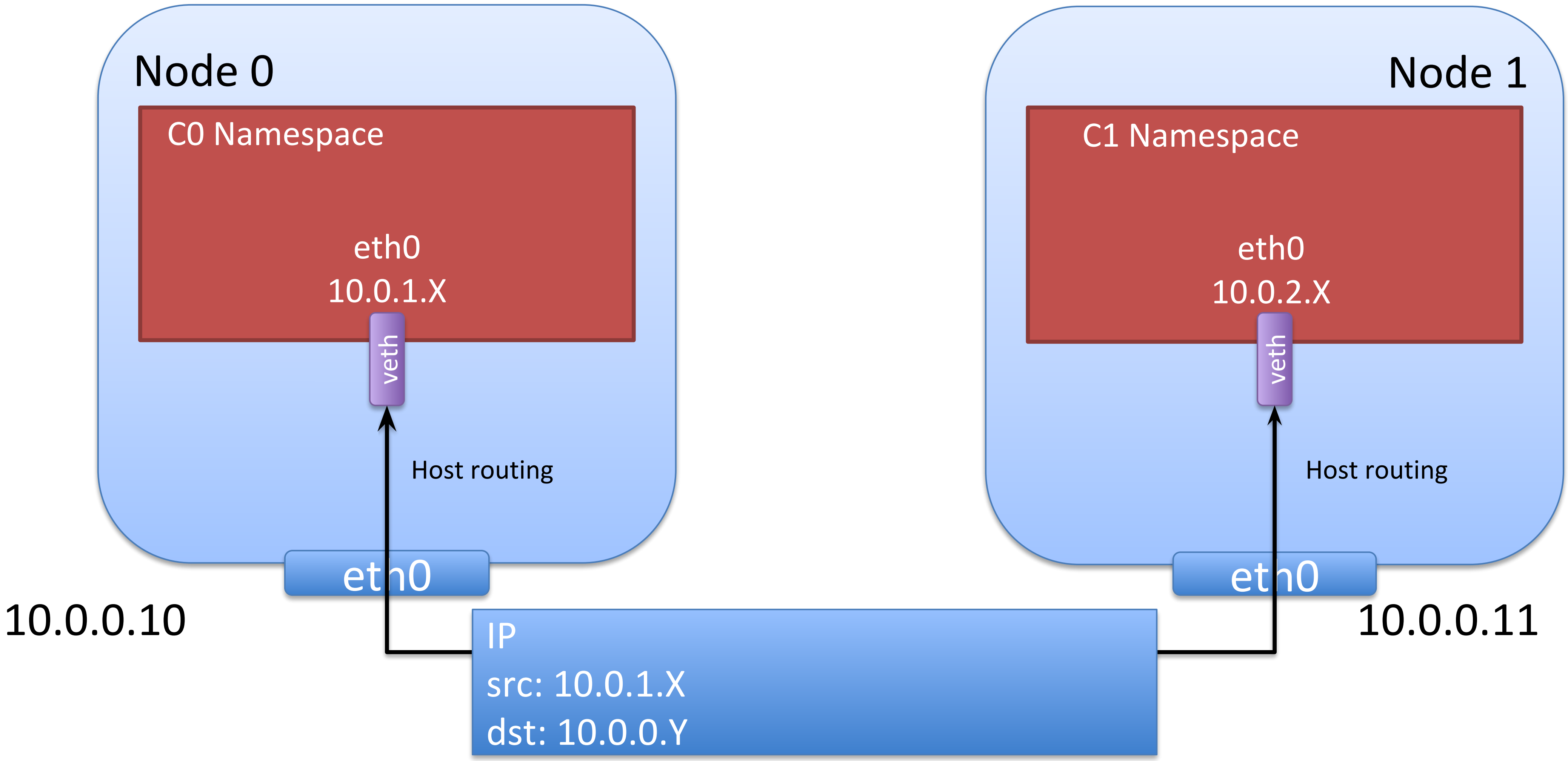
Overlay control plane



Pod routing - best practices

- Avoid overlays
  - Less overhead
  - Simpler and more scalable control plane
  - Easier to debug
- Avoid bridges (PTP or IPVLAN)
  - Better latency
  - Less CPU usage
- "Flat network"
  - Route from non-Kubernetes hosts/between clusters, simpler connectivity

Pod routing


Node 0
C0 Namespace
eth0
10.0.1.X
veth
Host routing
eth0
10.0.0.10
Node 1
C1 Namespace
eth0
10.0.2.X
veth
Host routing
eth0
10.0.0.11
IP
src: 10.0.1.X
dst: 10.0.0.Y

Pod routing

How is this done in practice?

Non-cloud (e.g. on-prem)

Cloud

- Google Cloud GCP
- Amazon AWS

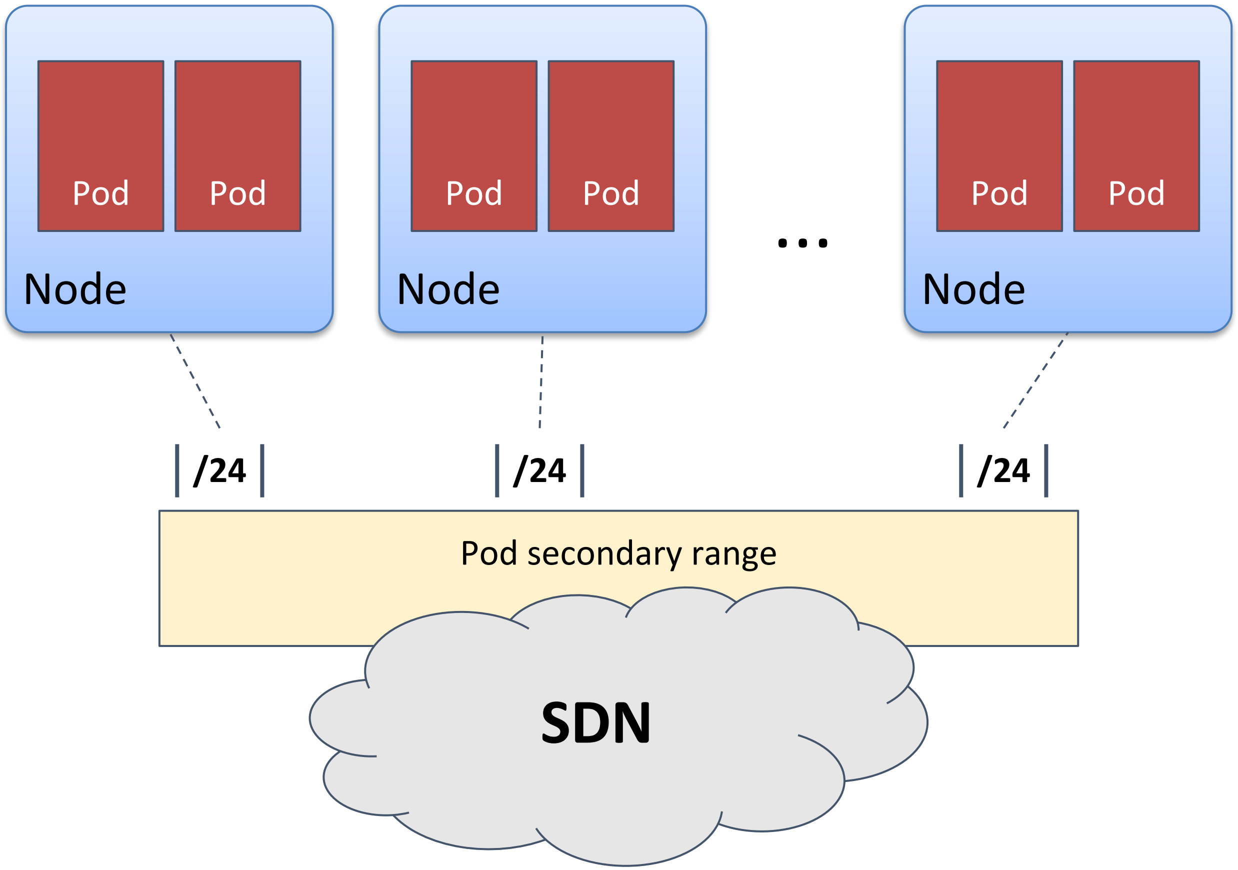
Pod routing (GCP)

Old way

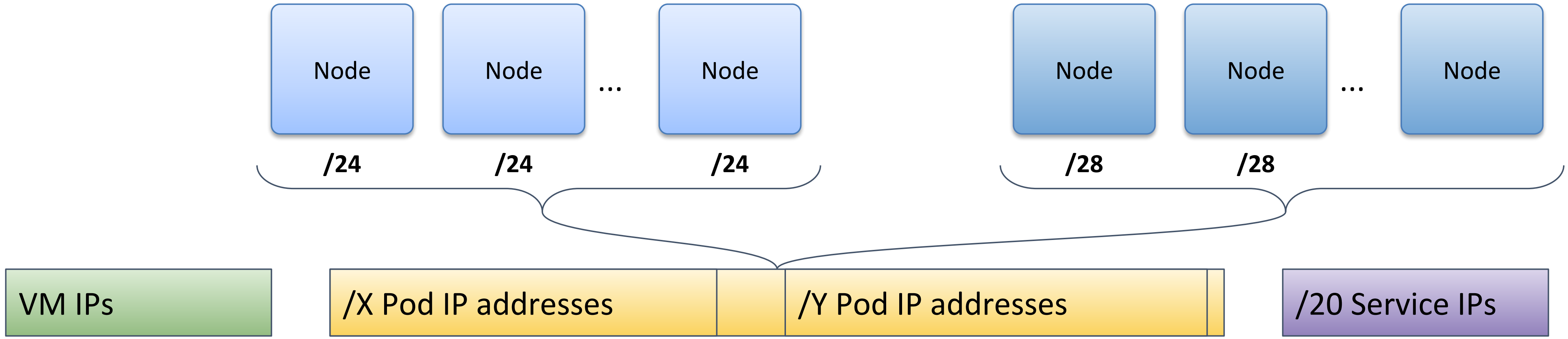
- Static routes
- May run into quota issues
- Complex and interacts with other constructs on the network

VPC-Native

- Pre-allocated range for Pods (secondary range)
- Cloud infrastructure manages IP assignment to nodes (aliases)
- More semantic and efficient for underlying SDN



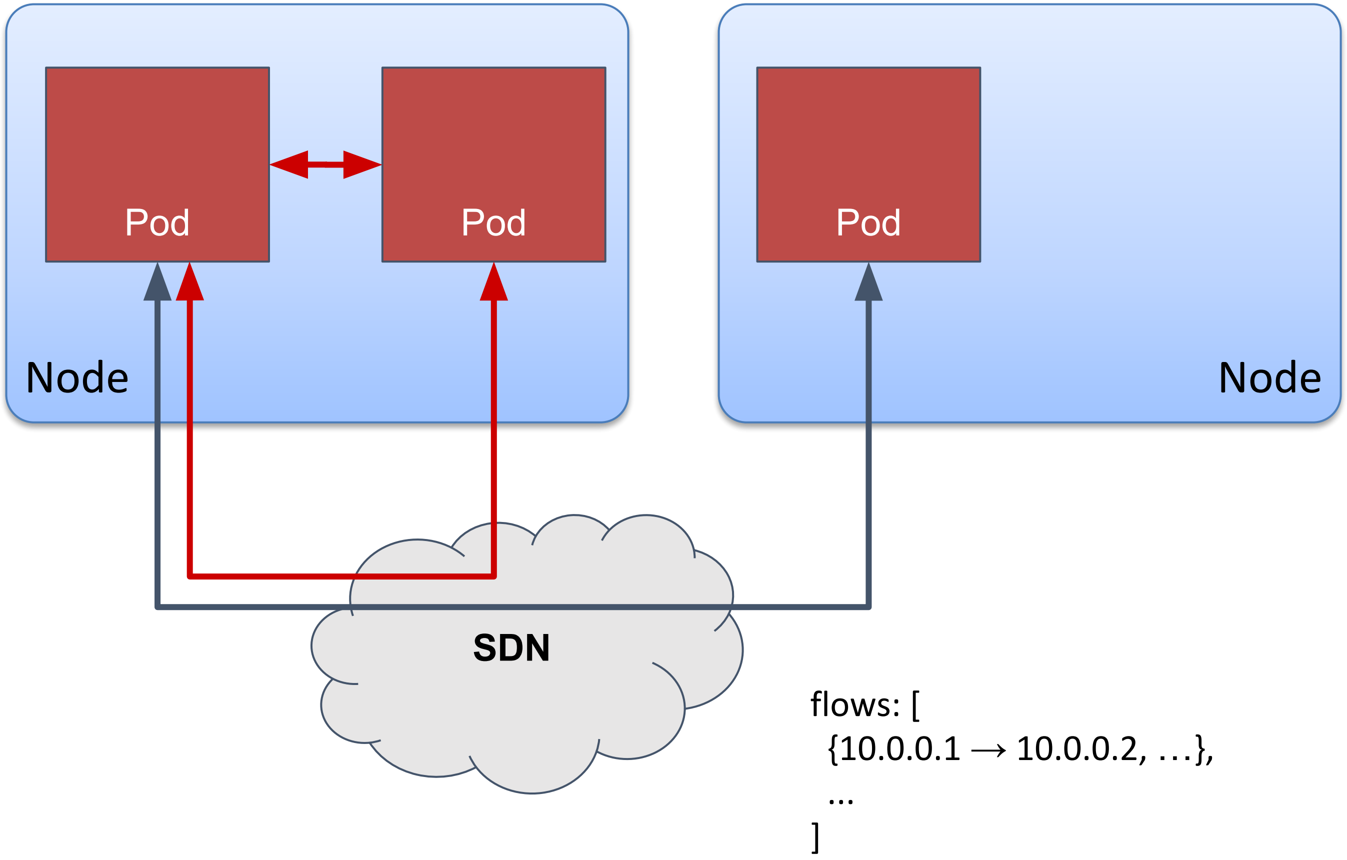
Pod routing (GCP)



Future work: don't make assumptions about contiguous ranges

1k nodes × /24	= 1000 × 256 IPs	= 256000 IPs
1k nodes × /8	= 1000 × 16	= 16000 IPs

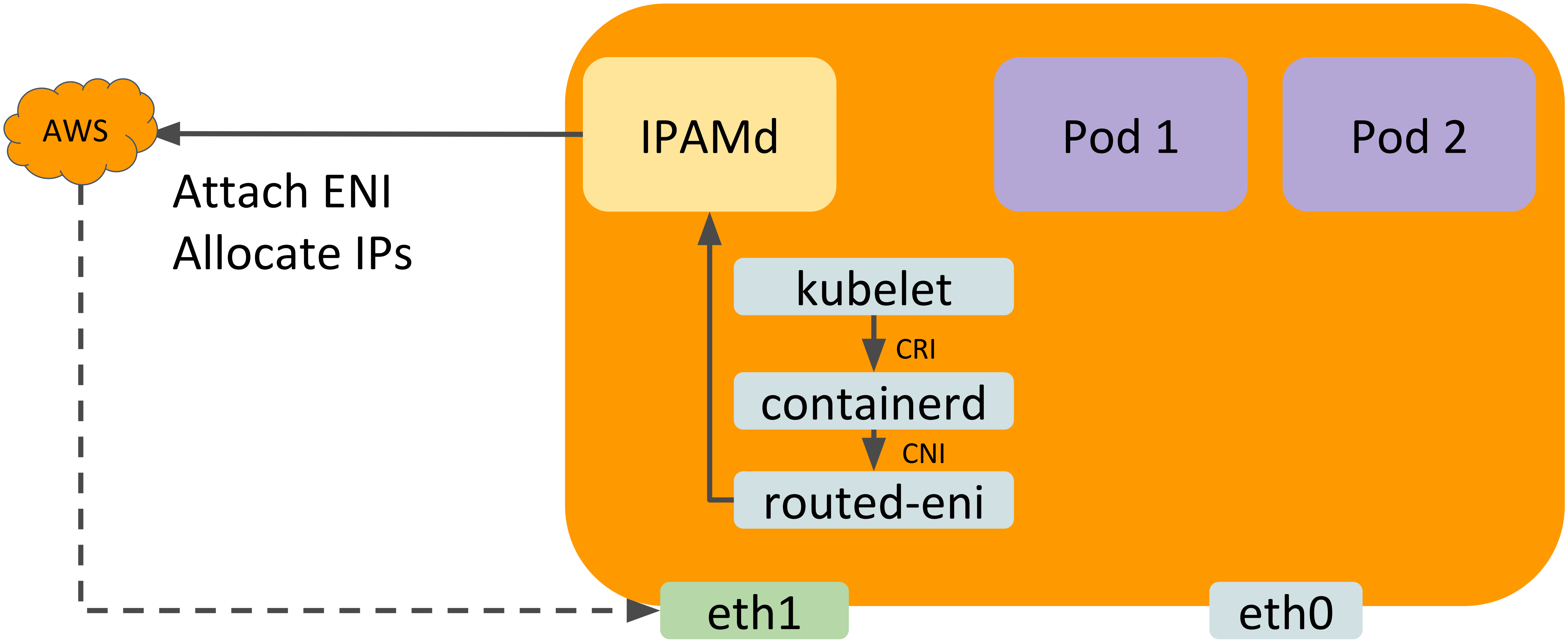
Pod routing (GCP)



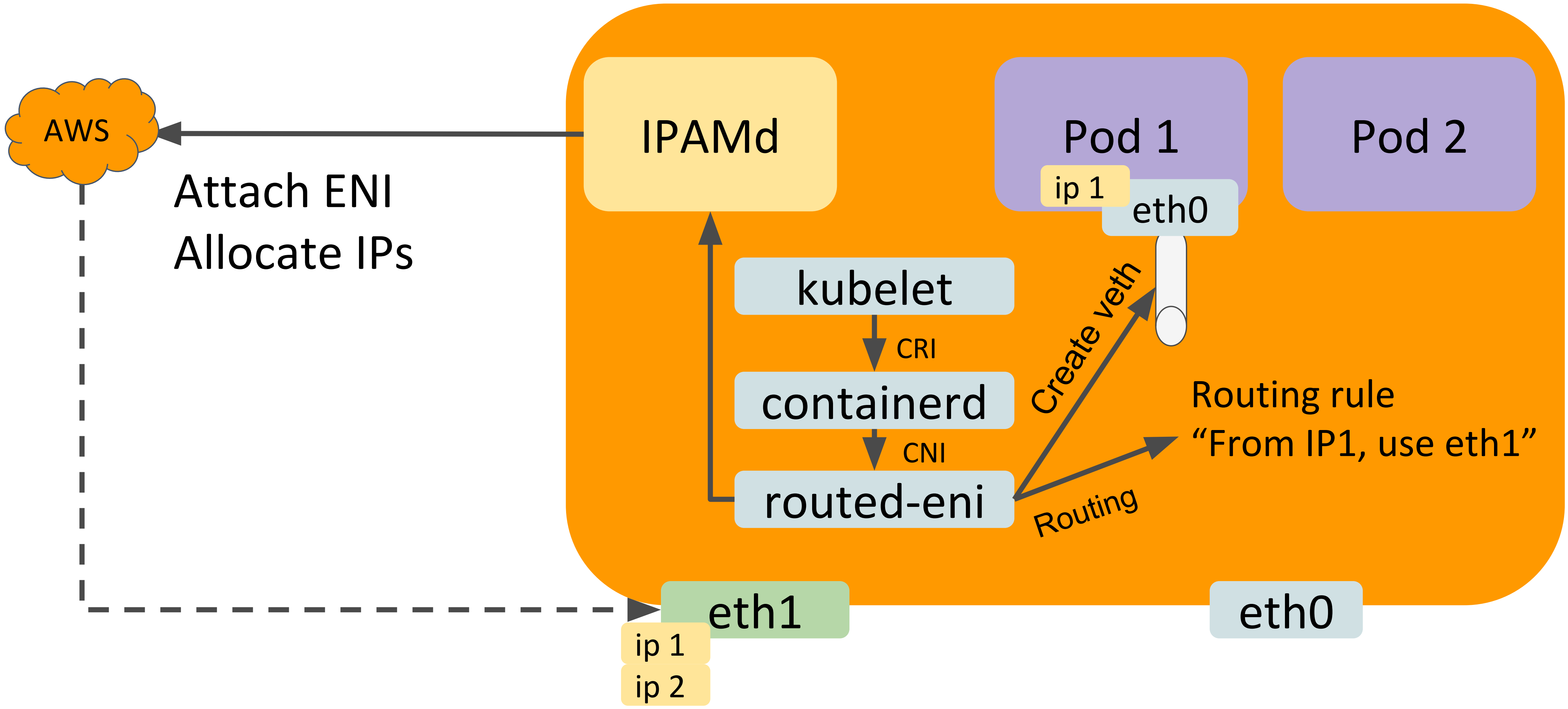
Integration with SDN for visibility

Make sure all traffic is seen by the SDN
-- CNI hairpins pod-to-pod traffic out of the Node to the fabric.

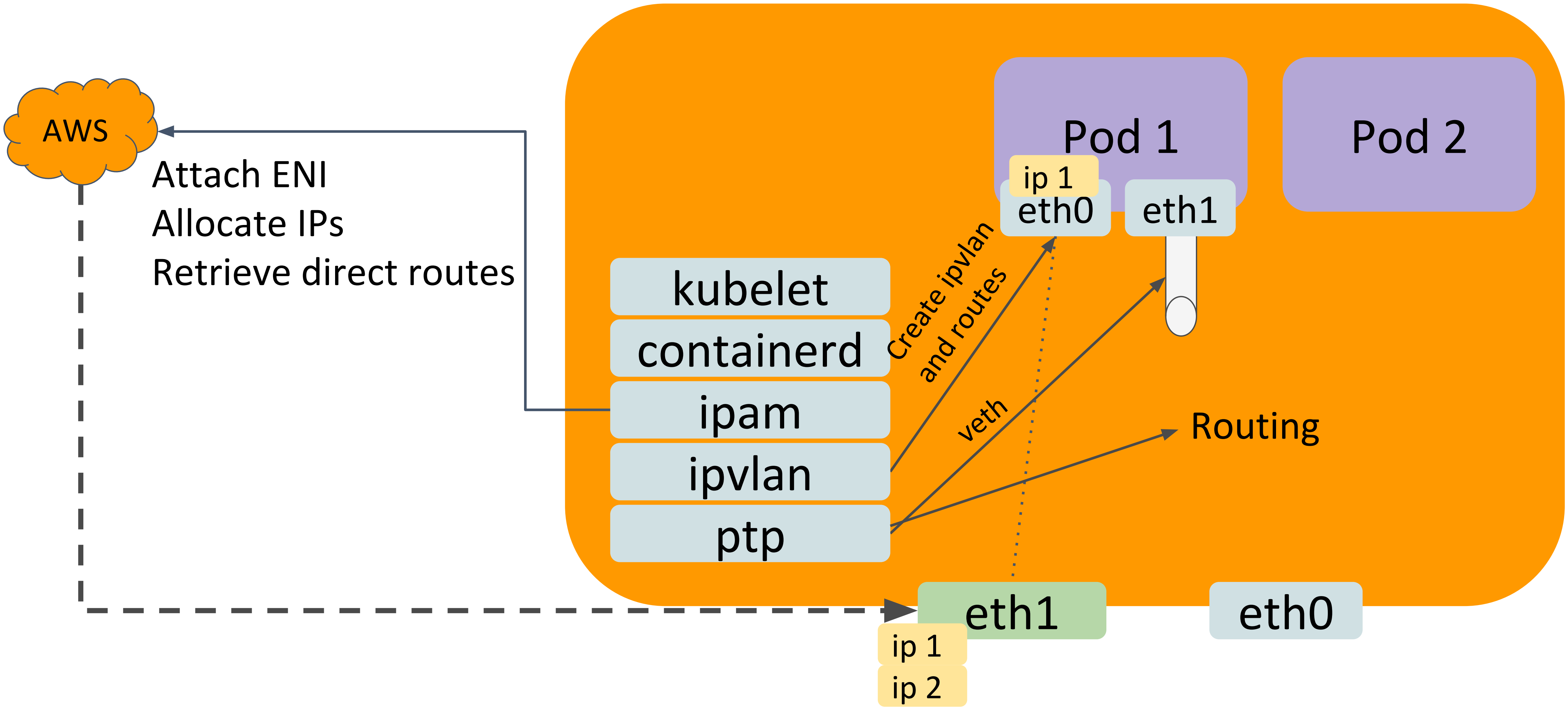
AWS EKS CNI plugin



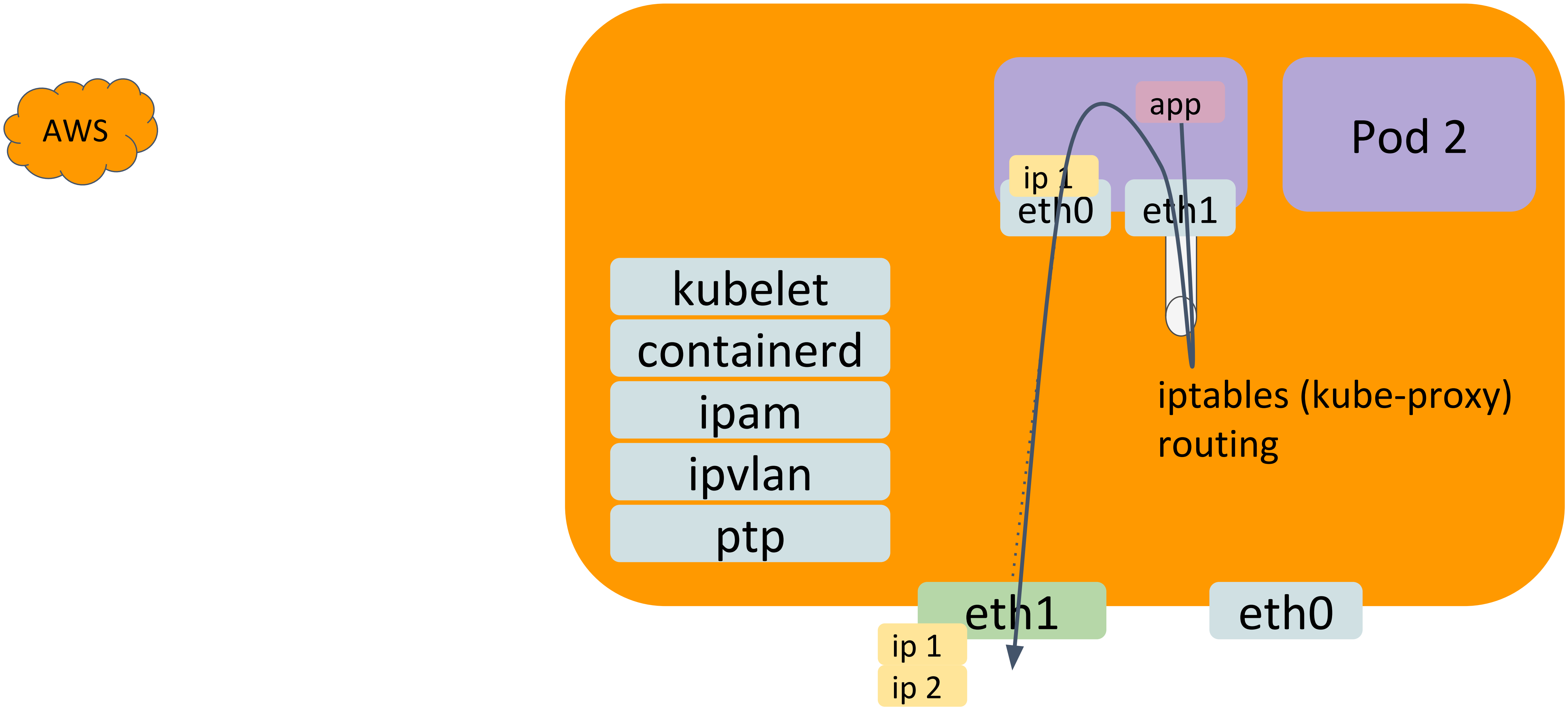
AWS EKS CNI plugin



Lyft AWS CNI plugin



Lyft AWS CNI plugin: ClusterIP



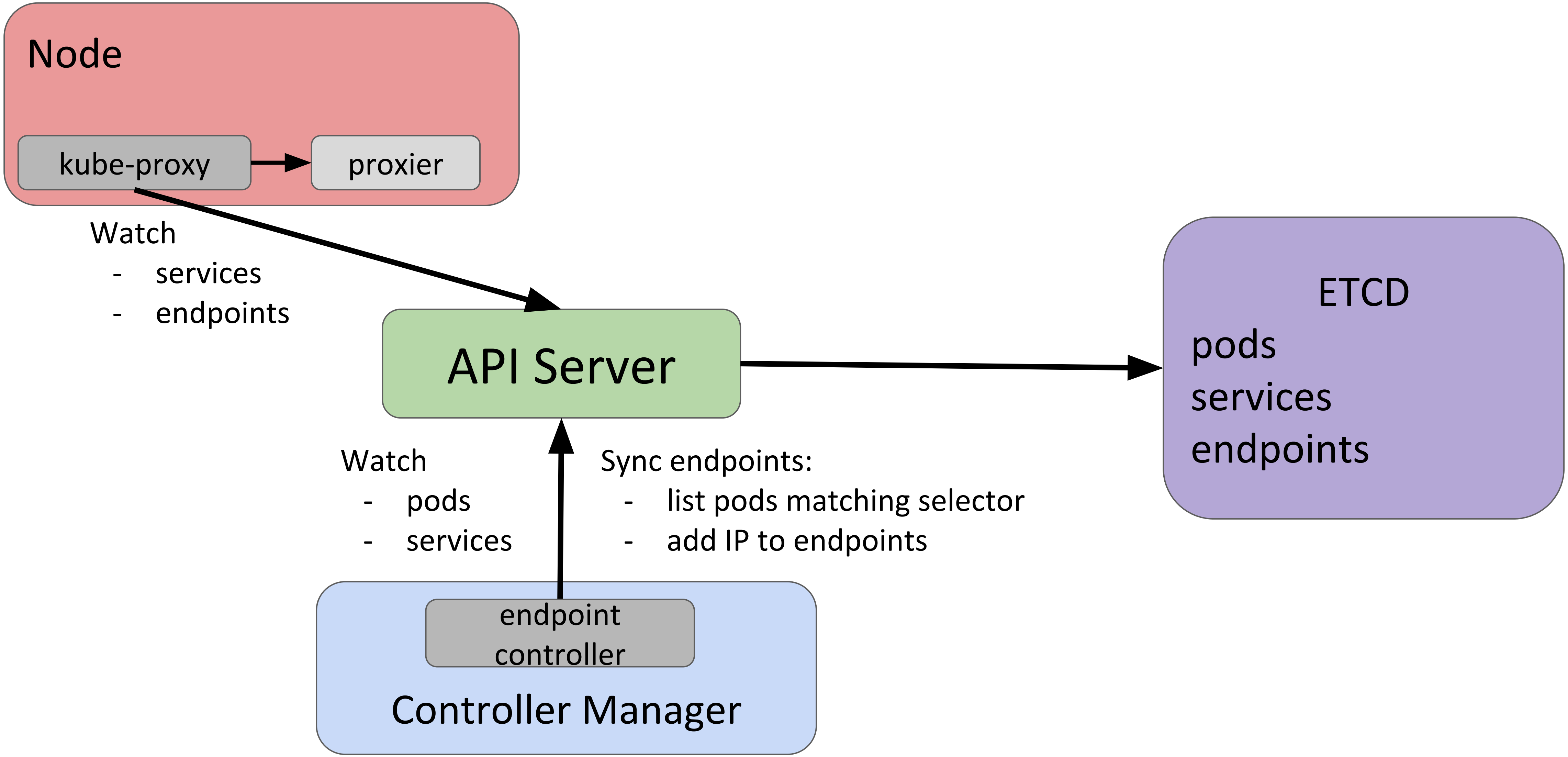
Native pod routing

- Don’t pay the cost of the overlay
- CNI plugins are a bit young but improving fast
- Allows for cross-cluster traffic in the same VPC
- Simplify traffic for higher layers (e.g. Ingresses)

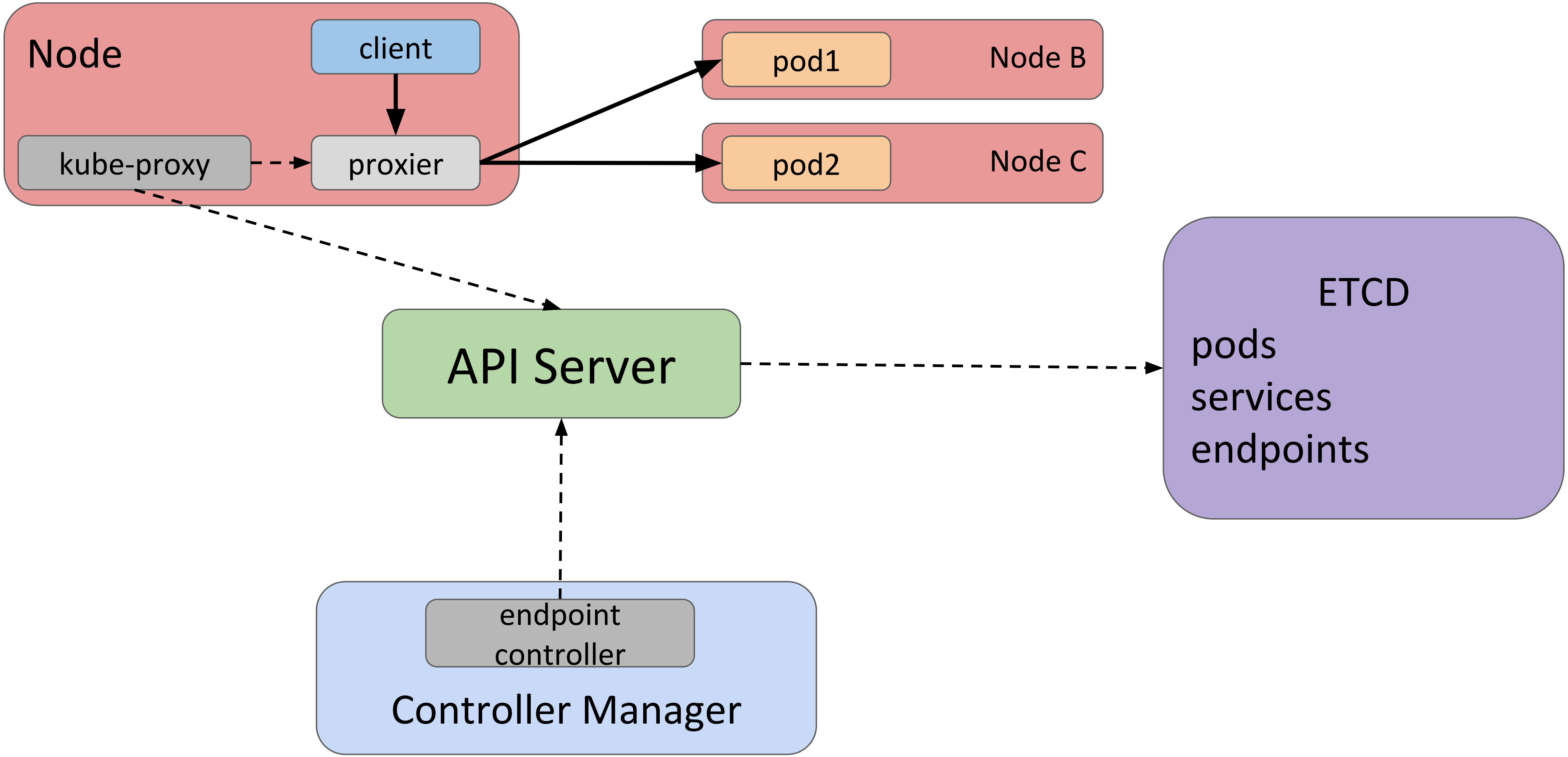
KubeCon | CloudNativeCon
Europe 2019

Service Load-Balancing

Role of kube-proxy



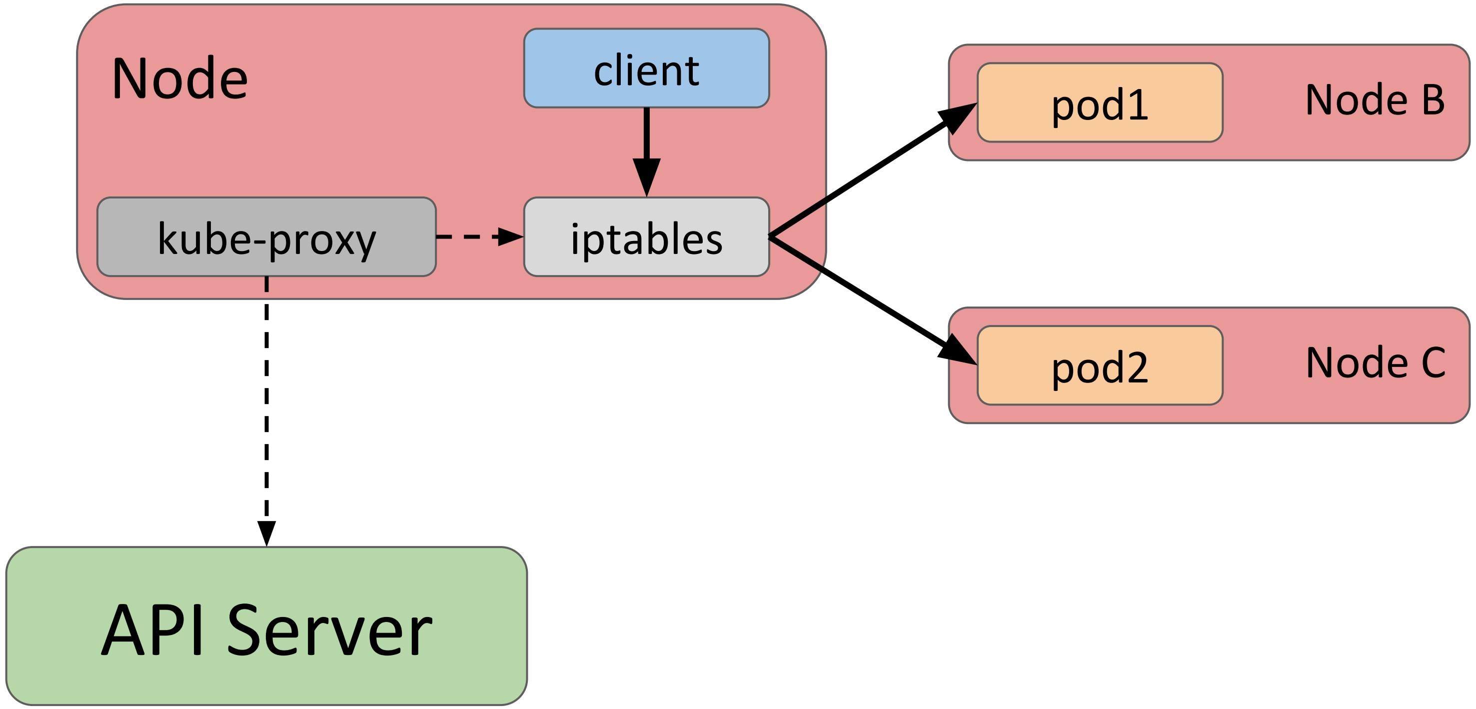
Role of kube-proxy



kube-proxy modes

- **Userspace**
  - Original implementation
  - Userland TCP/UDP proxy
- **IPtables**
  - Default since Kubernetes 1.2
  - Use iptables to load-balance traffic
  - Faster than userspace
- **IPVS**
  - GA since Kubernetes 1.11
  - Use Kernel load-balancing (LVS)
  - Still relies on iptables for some NAT rules
  - Faster than iptables, scales better with large number of services/endpoints

iptables overview



Outgoing traffic

1. Client to Service IP
2. DNAT: Client to Pod1 IP

Reverse path

1. Pod1 IP to Client
2. Reverse NAT: Service IP to client

iptables: how it works

PREROUTING / OUTPUT
any / any => KUBE-SERVICES

All traffic is processed by kube chains

KUBE-SERVICES
any / VIP:PORT => KUBE-SVC-XXX

Global Service chain
Identify service and jump to appropriate service chain

KUBE-SVC-XXX
any / any proba 33% => KUBE-SEP-AAA
any / any proba 50% => KUBE-SEP-BBB
any / any => KUBE-SEP-CCC

Service chain (one per service)
Use **statistic** iptables module (*probability of rule being applied*)
Rules are evaluated **sequentially** (hence the 33%, 50%, 100%)

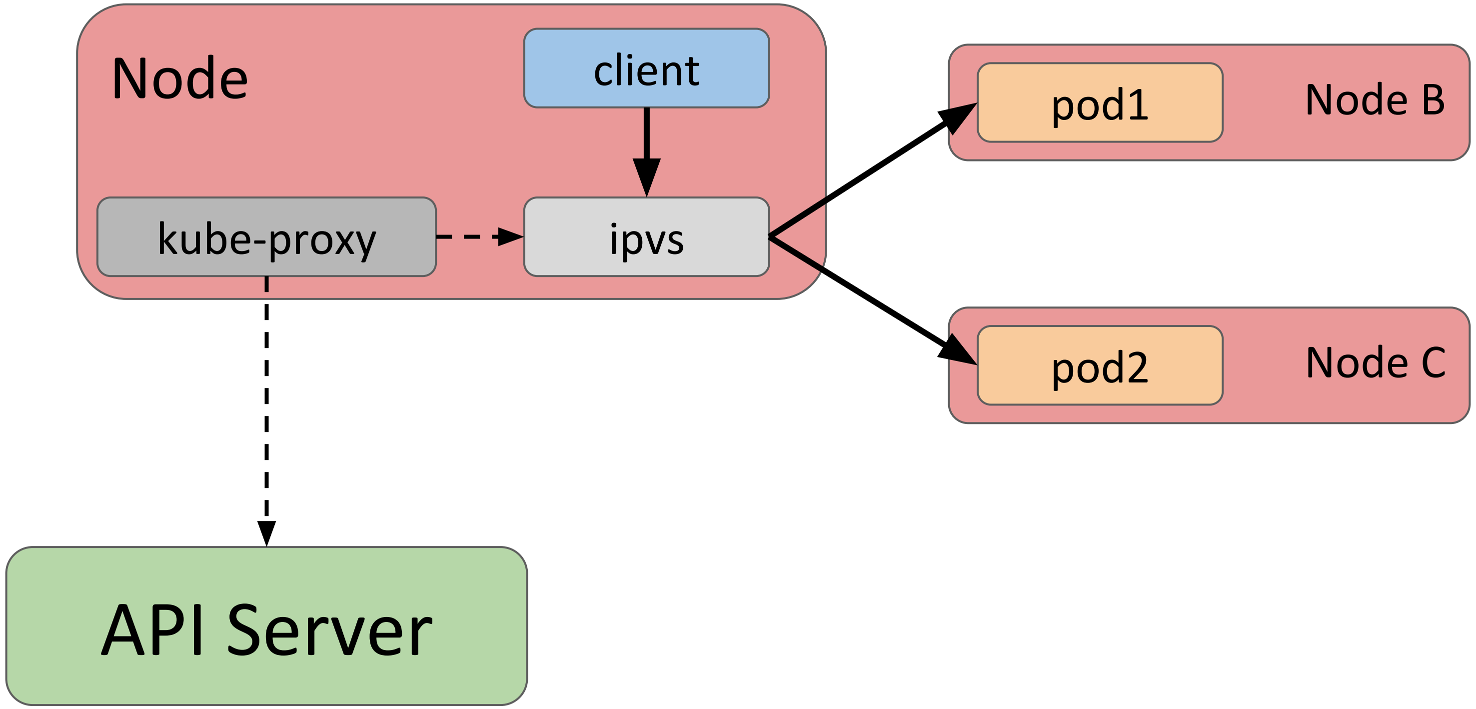
KUBE-SEP-AAA
endpoint IP / any => KUBE-MARK-MASQ
any / any => DNAT endpoint IP:Port

Endpoint Chain
Mark hairpin traffic (client = target) for SNAT
DNAT to the endpoint

iptables challenges

- iptables kernel interface requires sync of all rules in one shot
  - Longer sync time (~seconds)
  - Significant memory usage (>100 mb)
- Use of conntrack leads to "right-sizing" issues
  - How much memory to reserve?
  - DNS + UDP leads to entry exhaustion (even worse adding in v6)
- This is everything you can possibly do with iptables
  - Very hard to conceive of adding additional functionality...

IPVS overview



Service

IPVS virtual server

Pods

IPVS real servers

Updates

Atomic (pod level)

IPVS implementation


```
$ ipvsadm --list --numeric --tcp-service 10.200.200.68:80
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn

TCP  10.200.200.68:http rr

  -> 10.1.242.2:5000              Masq    1      0          0
  -> 10.1.243.2:5000              Masq    1      0          0
```

Virtual Server: Service (ClusterIP:port)

Real Services: Pods (PodIP:port)

IPVS challenges

- **Graceful termination**
  - Introduced in 1.12.2 (and cherry-picked in 1.11.5)
  - First implementation had a race-condition on the netlink socket
  - A few bugs (service deletion, proxy restarts, UDP services), now fixed
- **Double connection tracking**
  - IPVS + iptables (SNAT)
  - Default timeouts (900 / 60 / 300)
- **CNI interaction**
  - Slightly different routing path in the Kernel
  - Lyft plugin: uses IIF which is no longer the pod veth

kube-proxy alternatives

Kube-router

- Pod Networking with BGP + Network Policies
- **IPVS based service-proxy**

Cilium

- **Relies on eBPF to implement service proxying**
- Implement security policies with eBPF

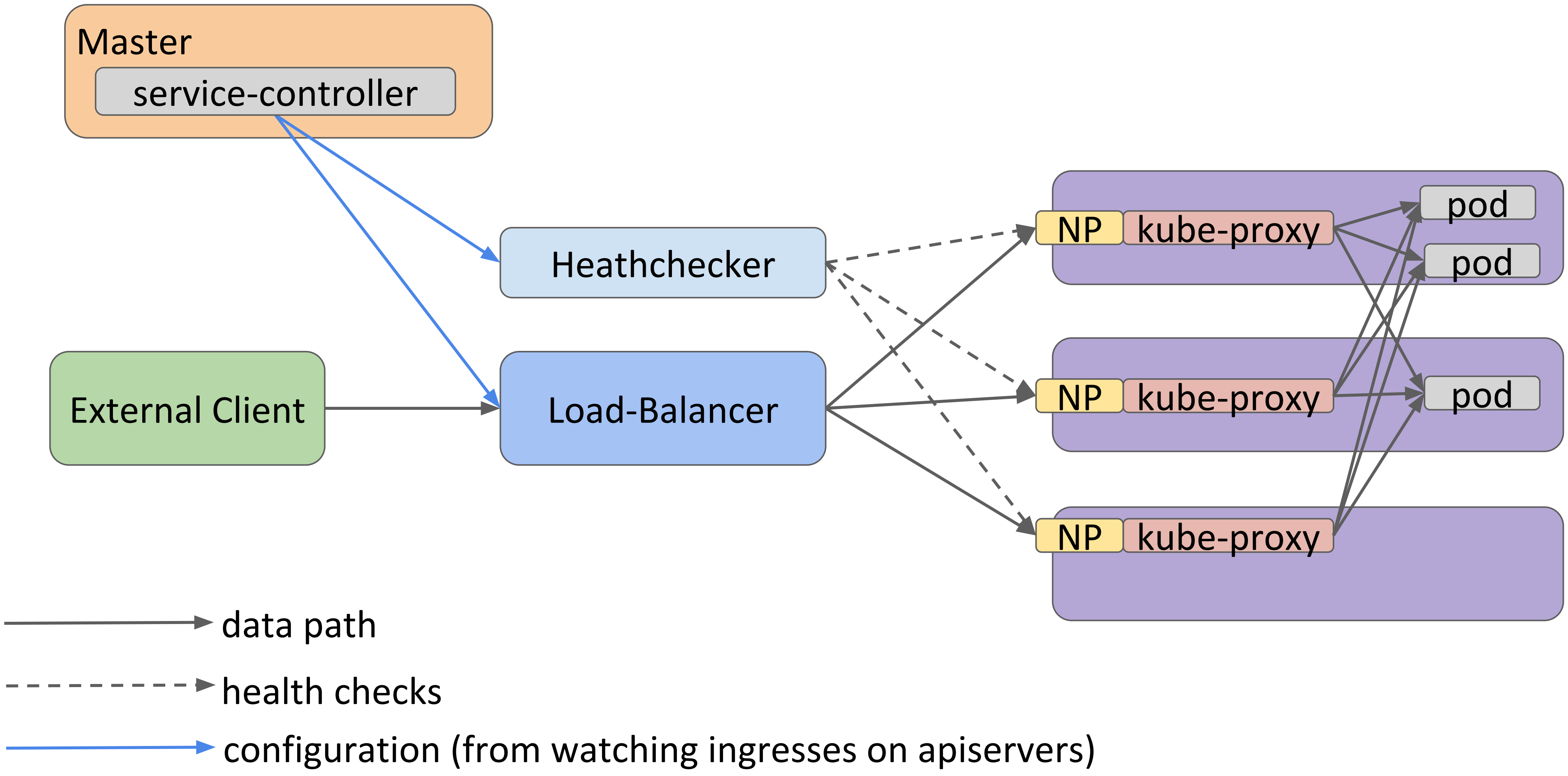
Features in preparation

- **Topology aware service routing**
  - Route to “local” domain
  - Connect to node local agent (monitoring, logs)
  - Connect to backends in the same zone (efficiency / pricing)
  - KEP: 0033-service-topology

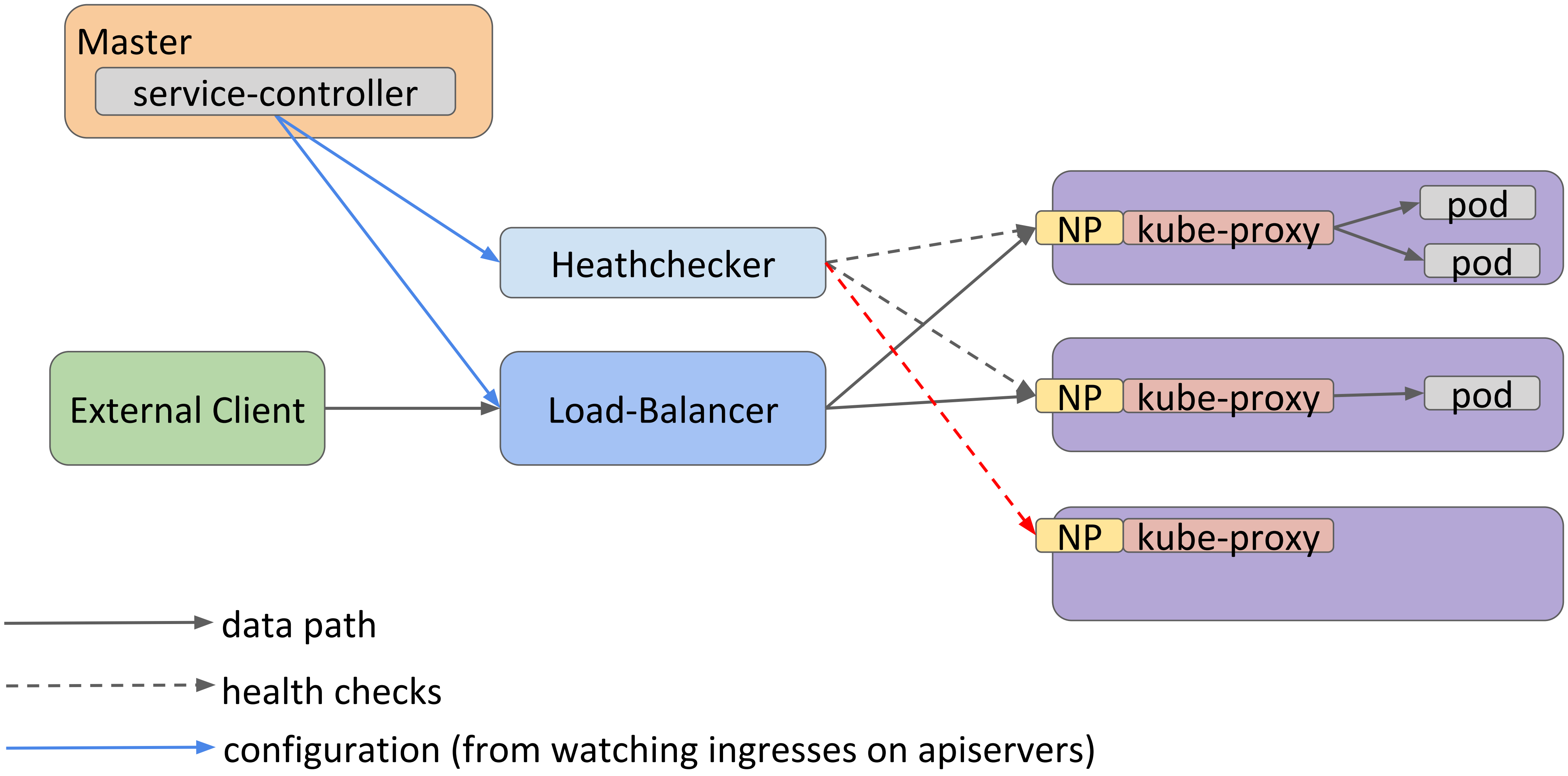
- **Scalability: EndpointSlice API**
  - Support services with 1000s of endpoints
  - More conditions than Ready / NotReady
    - Readiness is used both for services and rollout control
    - *Example for datastores:*
    - “Ready to serve traffic” vs “data is synced, rollout can continue”

Ingress traffic

Load-balancer services

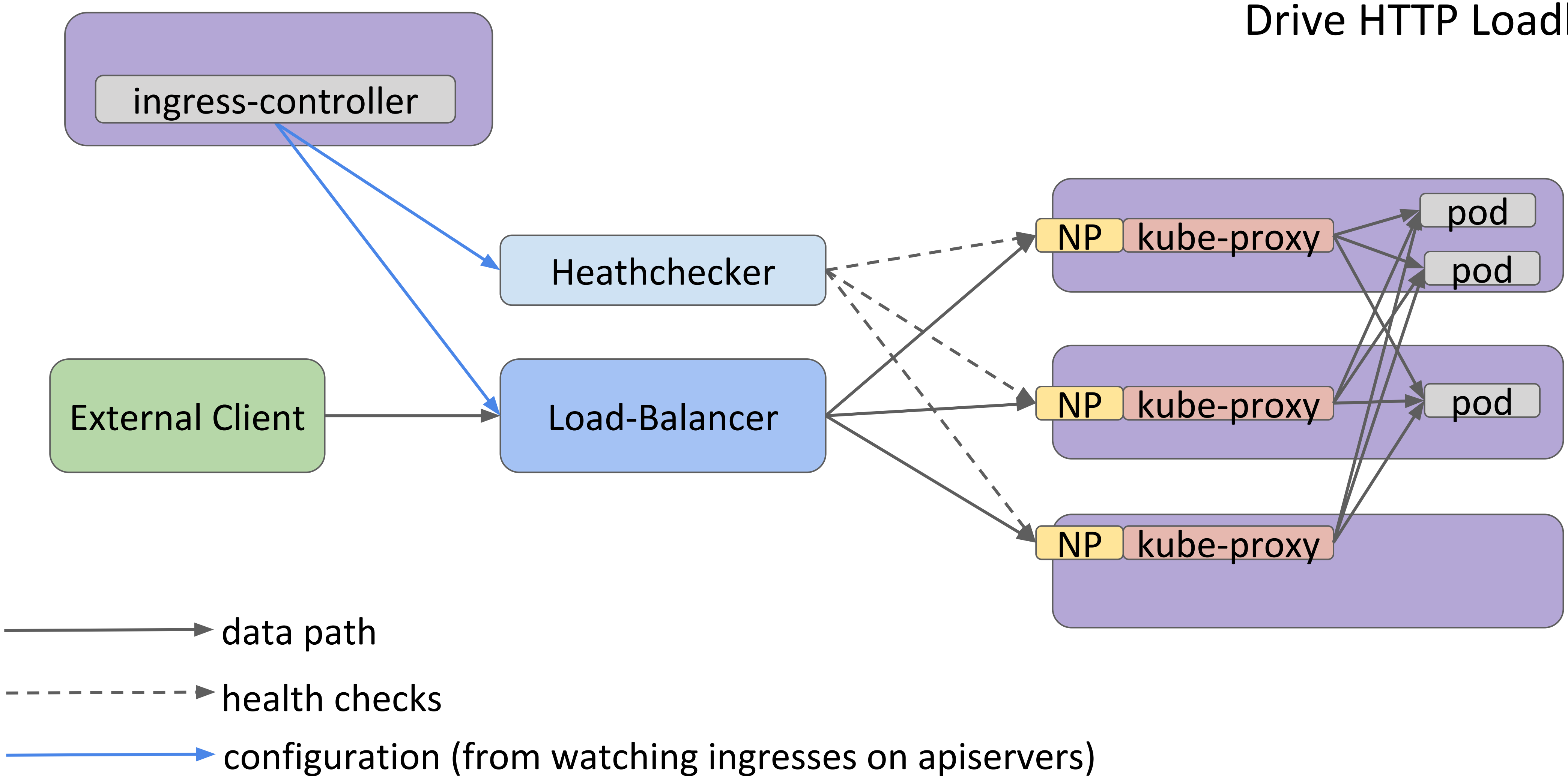


ExternalTrafficPolicy: Local

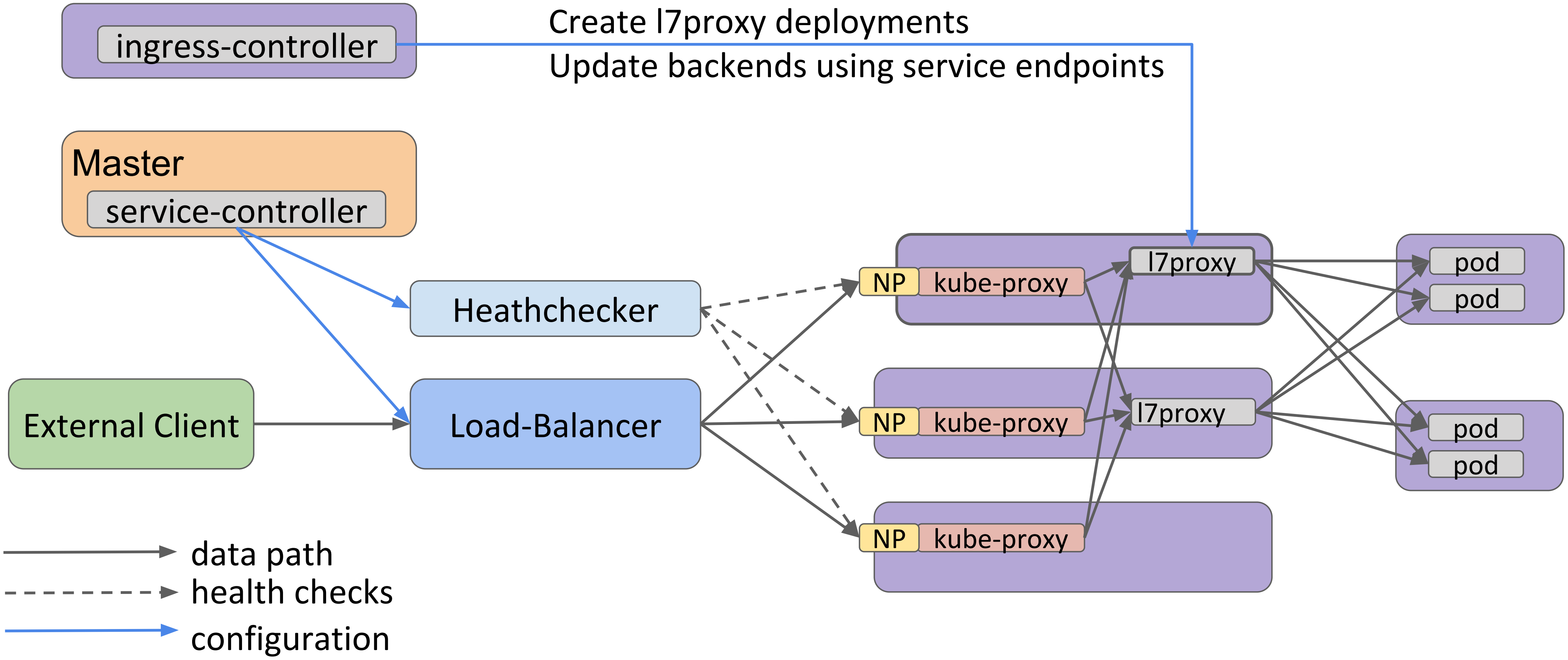


K8S HTTP Ingresses

Drive HTTP Loadbalancers



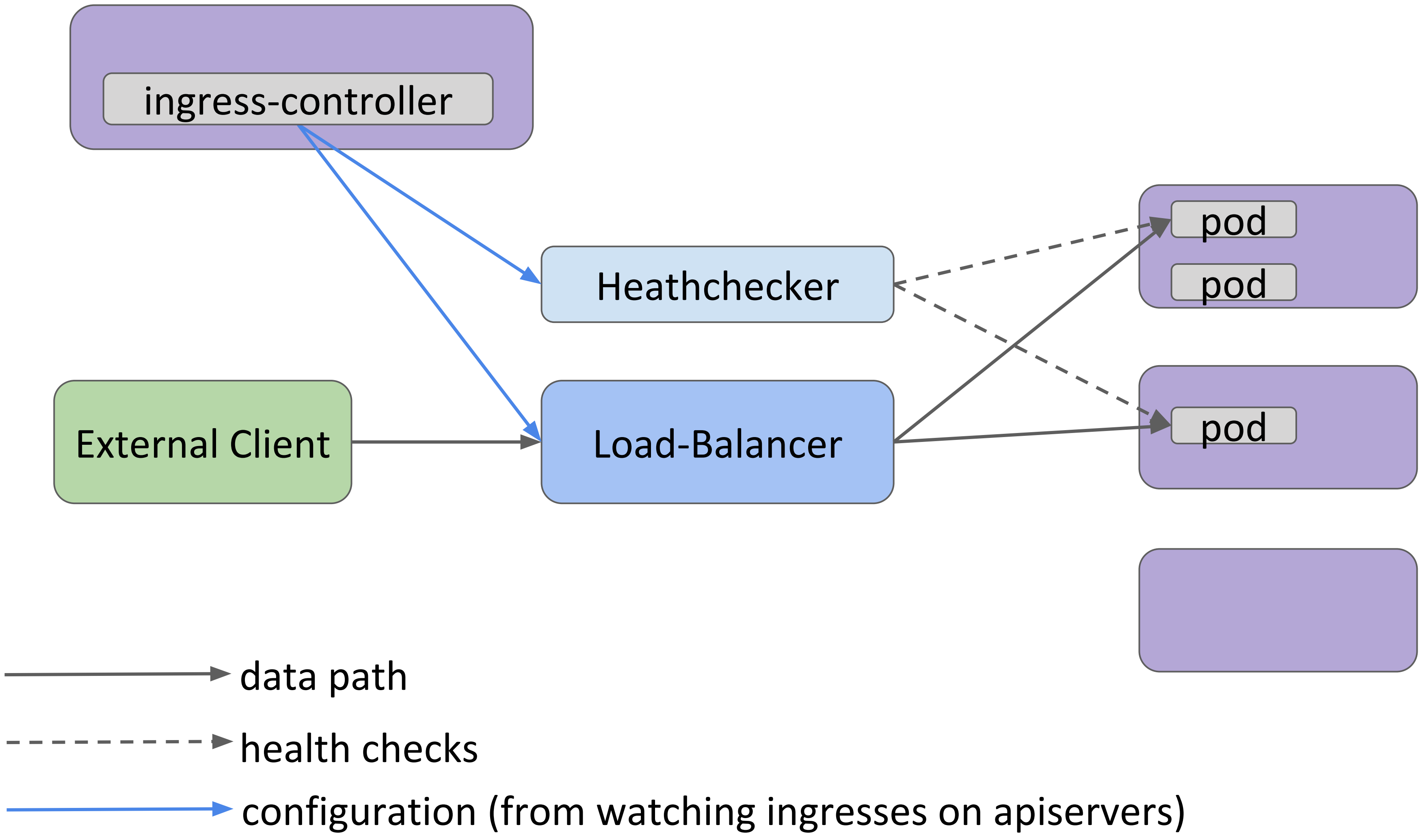
L7-proxy ingress controller



from watching ingresses/endpoints on apiservers (ingress-controller)

from watching LoadBalancer services (service-controller)

With native pod routing



Implementations

- AWS
  - alb-ingress-controller with “target-type: ip”
  - Creates an NLB
  - Attaches target-groups routing to pod IPs
- GCP
  - Network Endpoint Groups
- Proxy
  - ingress-nginx

Remaining challenges

Limited Loadbalancer support

No AWS ELB

No GCP ILB

Limited to HTTP ingresses

No TCP/UDP traffic

Need to change LB controllers

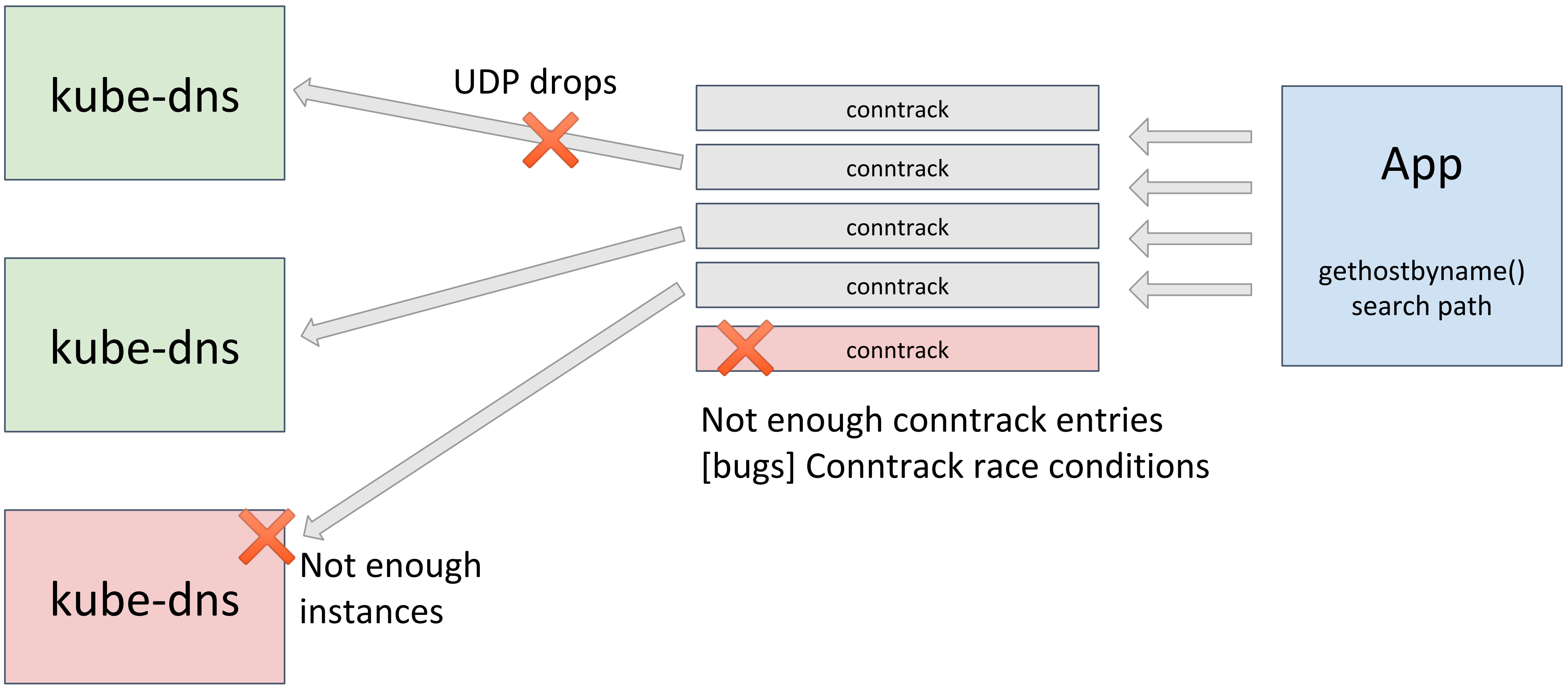
(NLB / NEG support TCP)

DNS challenges

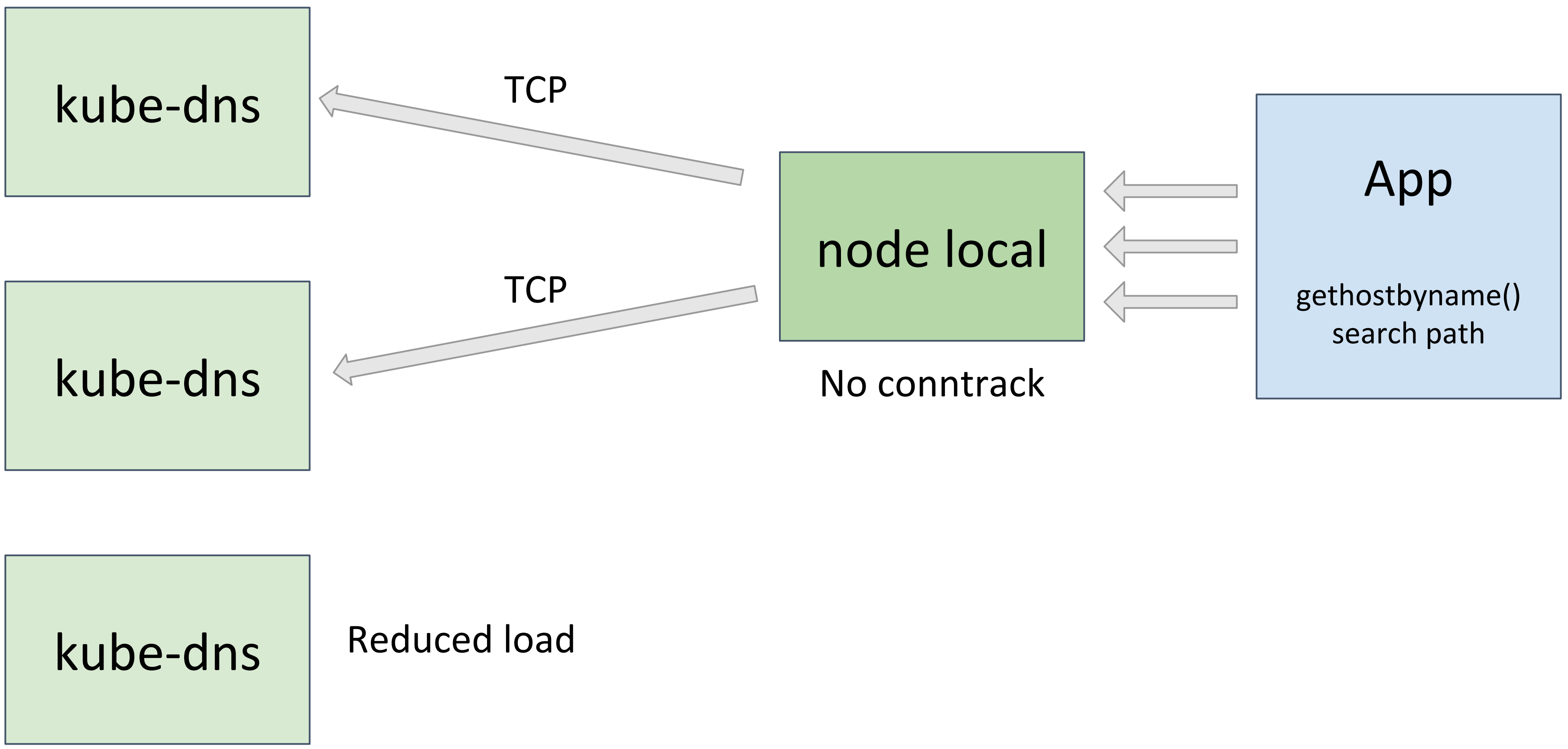
DNS issues at Datadog

- Conntrack race condition
  - Disable IPv6, use IPVS
  - *Much better*
- Issues with UDP and IPVS
  - Some traffic blackholed during rolling updates
  - *Should be fixed now*
- Volume of traffic: 1000s of queries per second
  - Partially due to search path
  - Rate-limiting on AWS VPC resolver

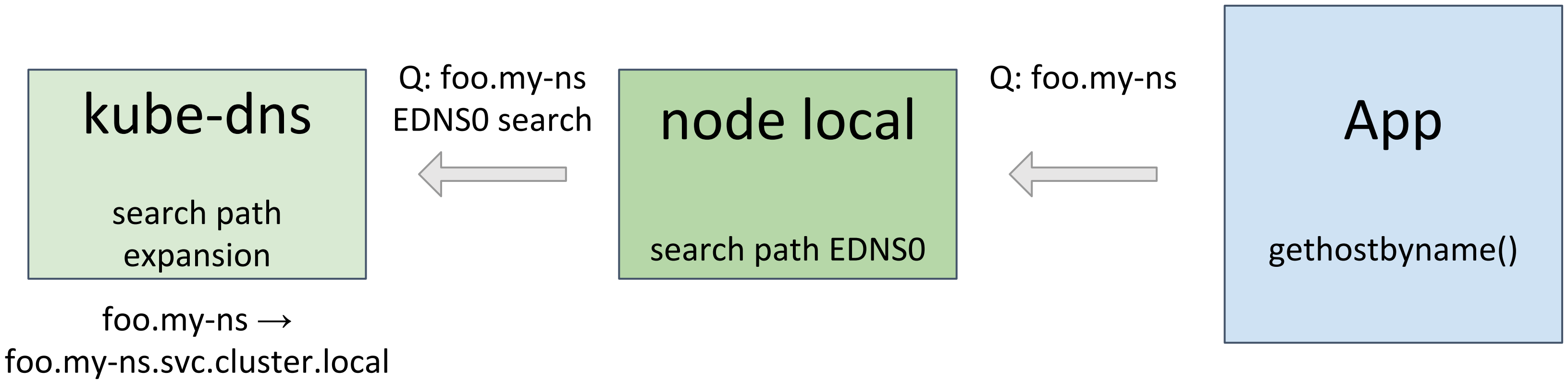
DNS challenges



DNS node local cache



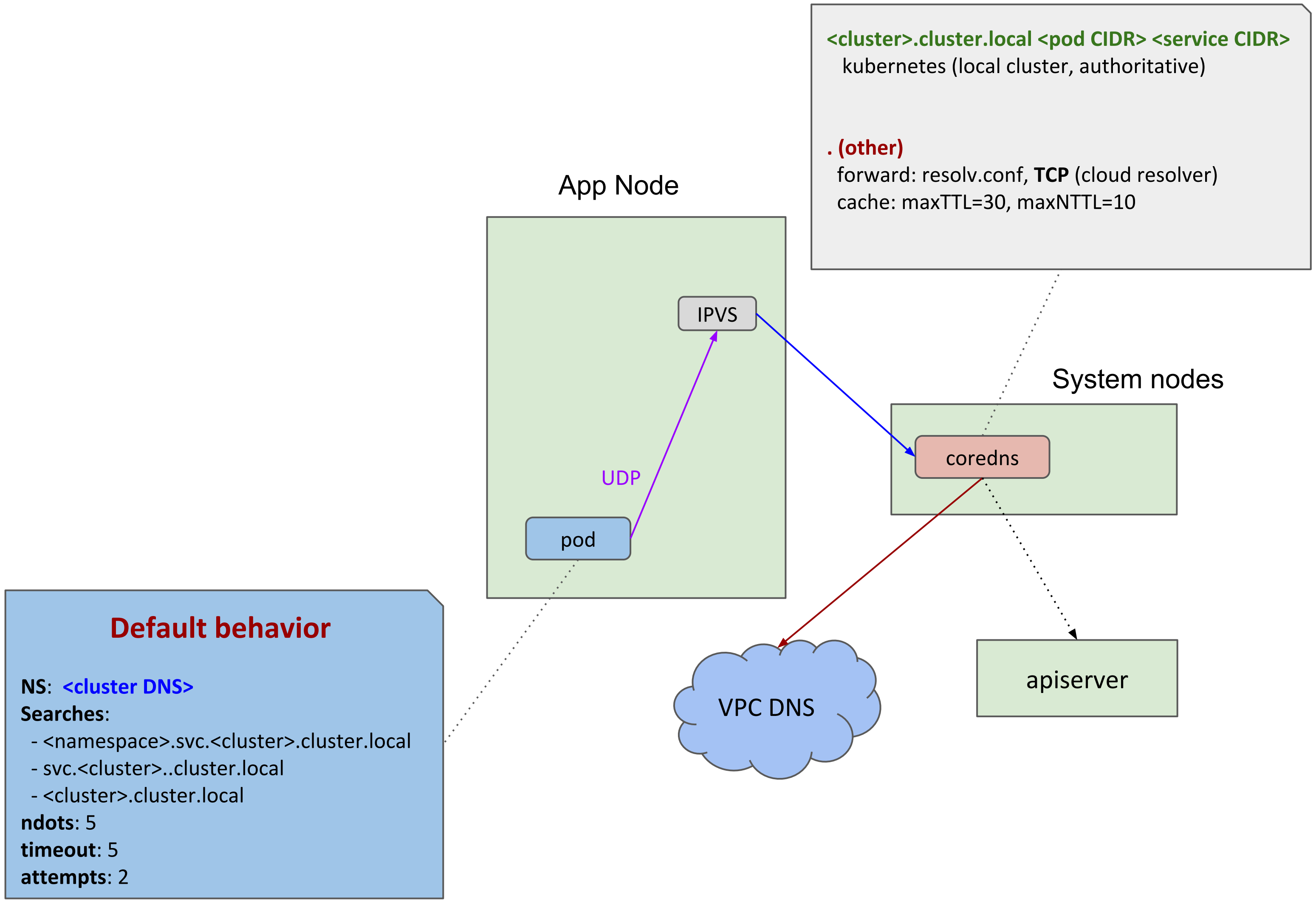
DNS future directions



KEP: DNS autopath in pod API

Current DNS setup at Datadog

Default behavior



Current DNS setup at Datadog

Opt-in behavior

NS: -169.254.20.10
- <cluster DNS> (fallback)

searches: svc.<cluster>.cluster.local
ndots: 2
timeout: 1
attempts: 1

cluster.local <CIDR>
forward: <cluster DNS>, **TCP**
cache: maxTTL=30, maxNTTL=10

. (other)
forward: resolv.conf, **TCP**
cache: maxTTL=30, maxNTTL=10

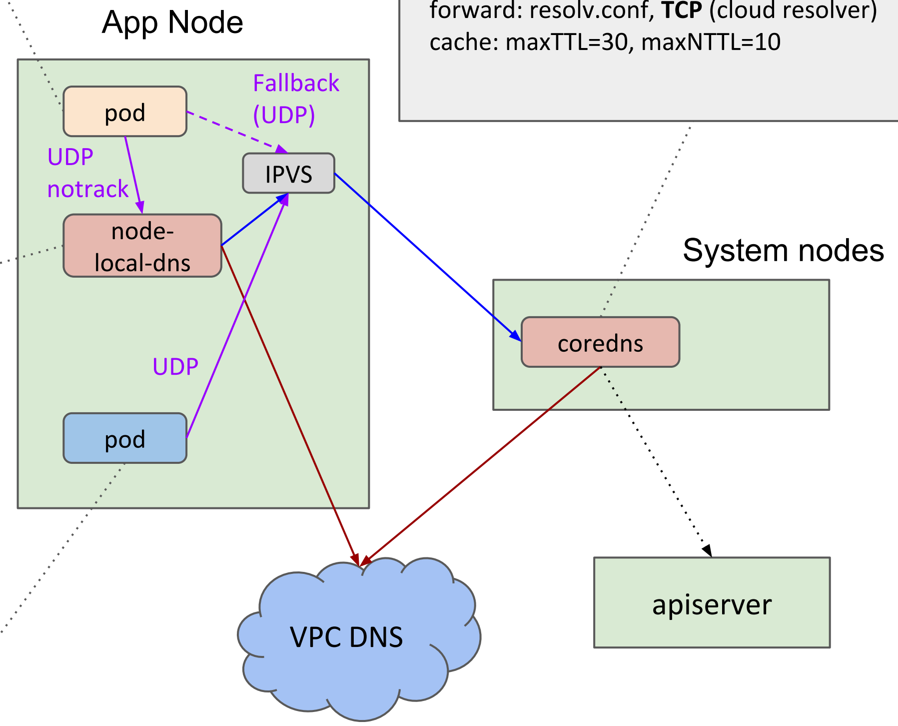
Default behavior

NS: <cluster DNS>
Searches:
- <namespace>.svc.<cluster>.cluster.local
- svc.<cluster>..cluster.local
- <cluster>.cluster.local

ndots: 5
timeout: 5
attempts: 2

<cluster>.cluster.local <pod CIDR> <service CIDR>
kubernetes (local cluster, authoritative)

. (other)
forward: resolv.conf, **TCP** (cloud resolver)
cache: maxTTL=30, maxNTTL=10



Default behavior

Opt-in with annotation + mutating webhook
- Single search domain, ndots=2
- Use local resolver (daemonset)
- Local resolver routes based on query
- Local resolver caches

➔ Queries are a lot faster
➔ Load on coredns is much lower
➔ A few slow queries on local pod update

Conclusion

Where should community go?

- Native integrations with the infrastructure is key
- Many KEPs in flight to improve scalability (IPVS, DNS, ...)
- Interesting future technologies not yet explored (eBPF, service mesh?)
- Make sure everything scales **by default** out of the box.