

KubeCon + CloudNativeCon

North America 2026

OpenAI

KubeCon CloudNativeCon North America 2026

Inside the Inference Load Balancer: Proxy/Controller Patterns for LLM Serving on Kubernetes

Salt Lake City, Utah, United States





Qianru Lao Lu Zhang

Agenda

01 **Inference Load Balancer**

Introduction and its evolution

02 **Architecture**

Control plane and data plane

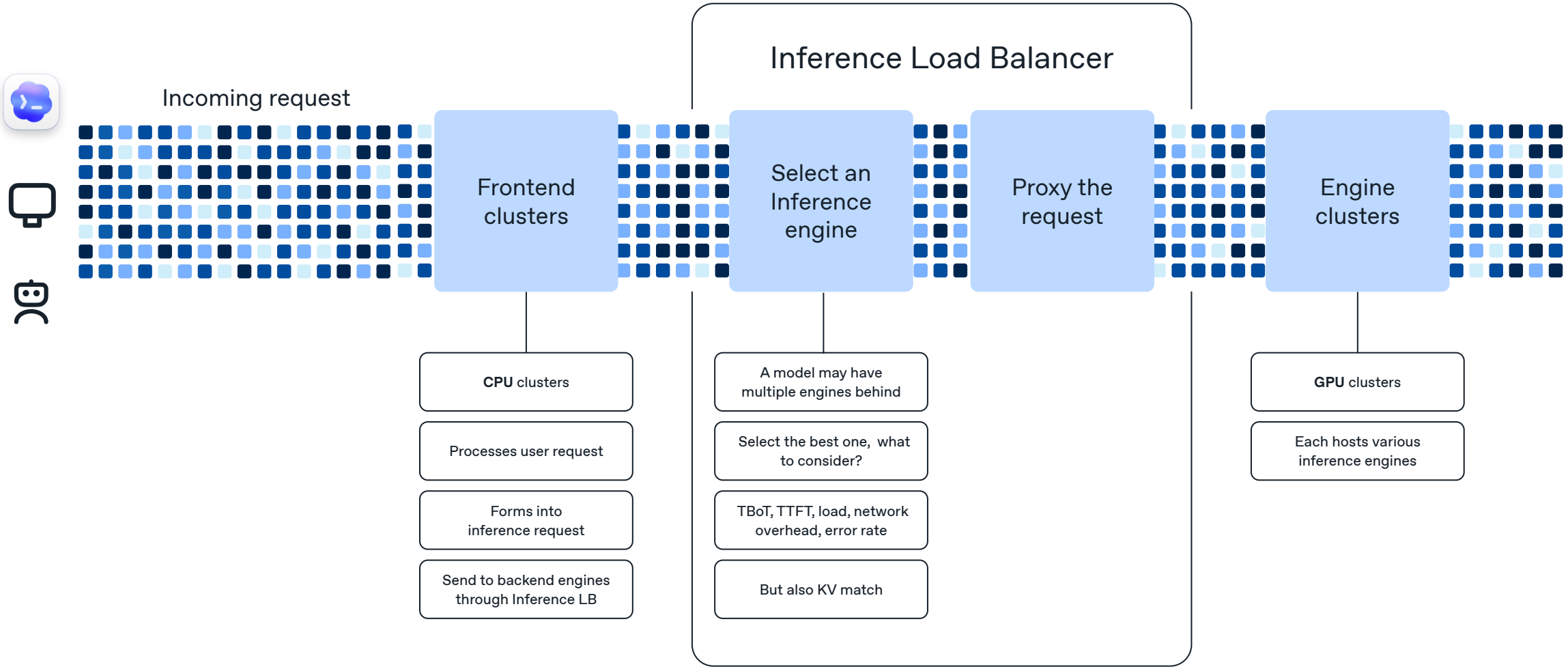
03 **Case study**

Reducing global networking overhead

04 **Protection mechanisms**

Error penalty, retry with cap and load shedding

What is Inference Load Balancer



Running the inference load balancer on Kubernetes

ILB runs in Kubernetes pods managed by Deployments.

Deployment

Rolling updates

When we release a new ILB version, Kubernetes gradually replaces the old pods.

HPA + readiness probes

Pod autoscaling

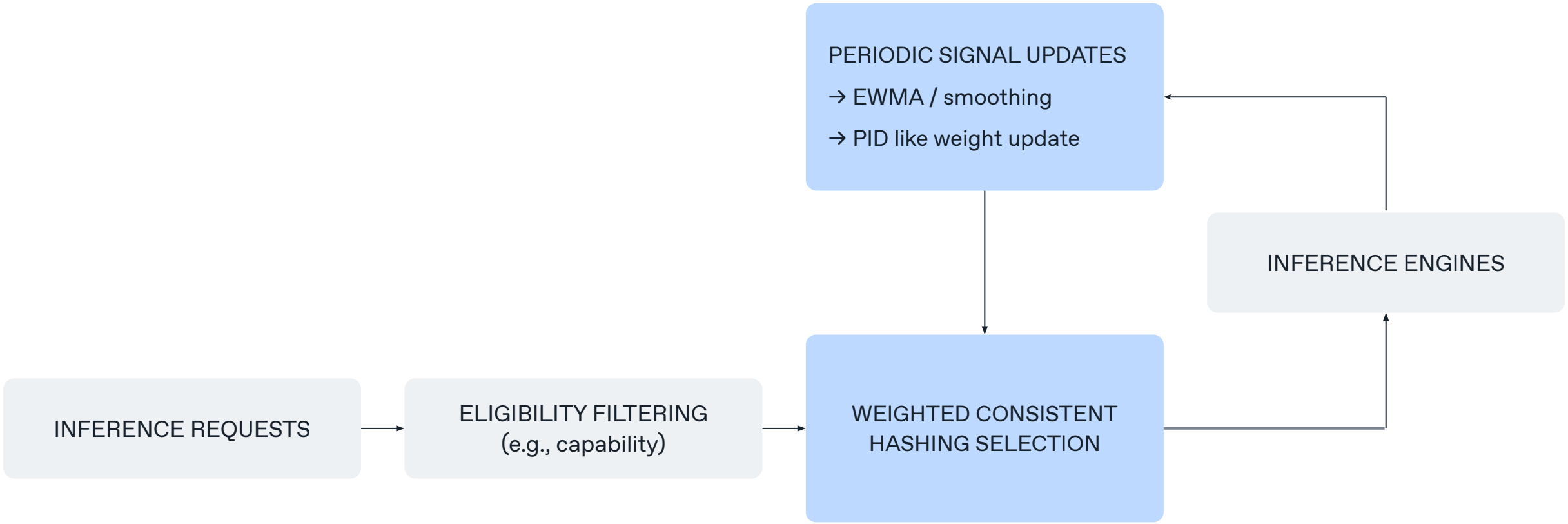
ILB pods serving requests scale with CPU load. New pods receive traffic once their warm-up check passes.

ConfigMap

Cluster locations

ILB pods read GPU engine cluster locations from a mounted configuration file.

Load Balancing in the “early days”: A periodic feedback loop



Load Balancing in the “early days”: A periodic feedback loop

Many signals considered

Routing restrictions naturally balanced out

Self-adaptive, like a “PID controller”

It “just works”

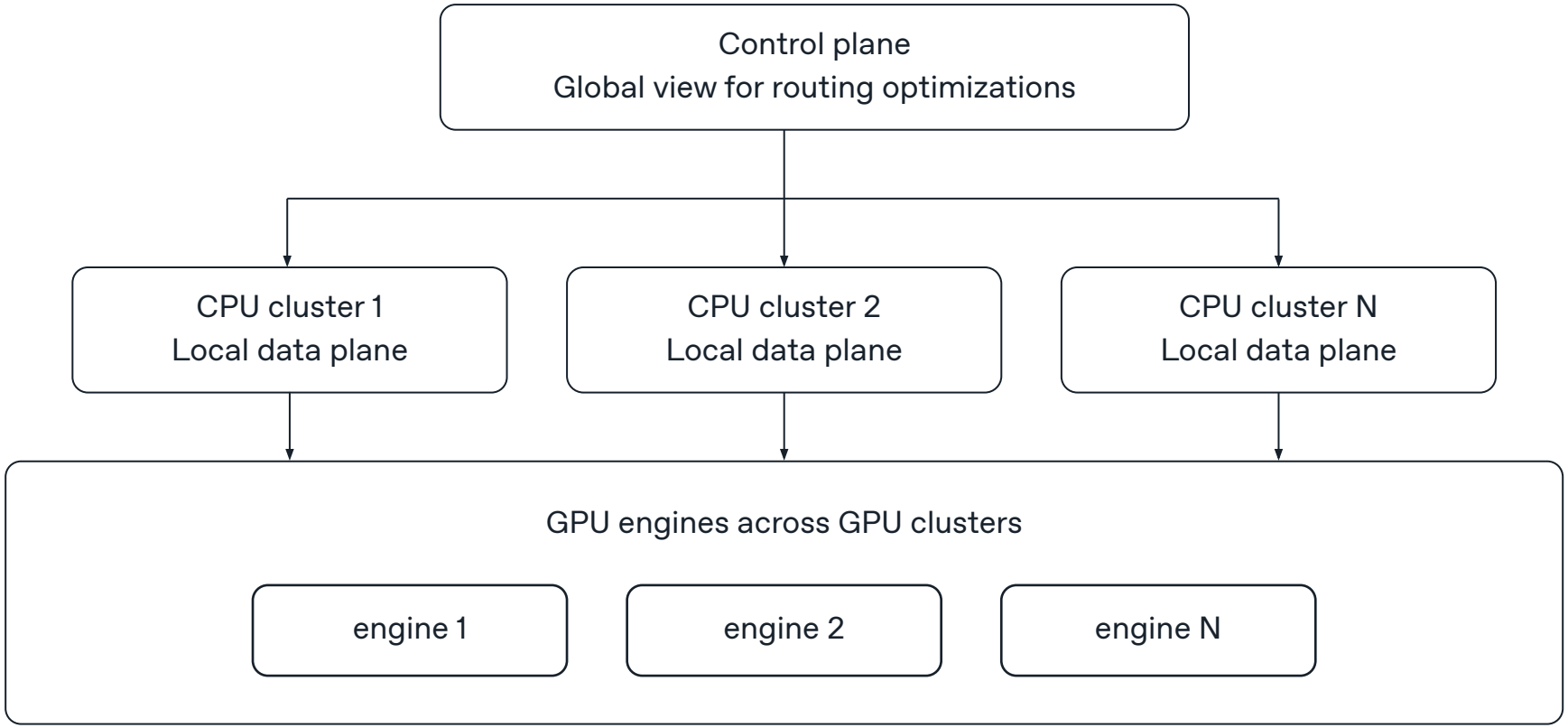
Hard to reason about routing decisions or control

Load not always evenly distributed, especially for a mixed fleet

Feedback loop creates slow oscillations, impacting KV cache hit rate

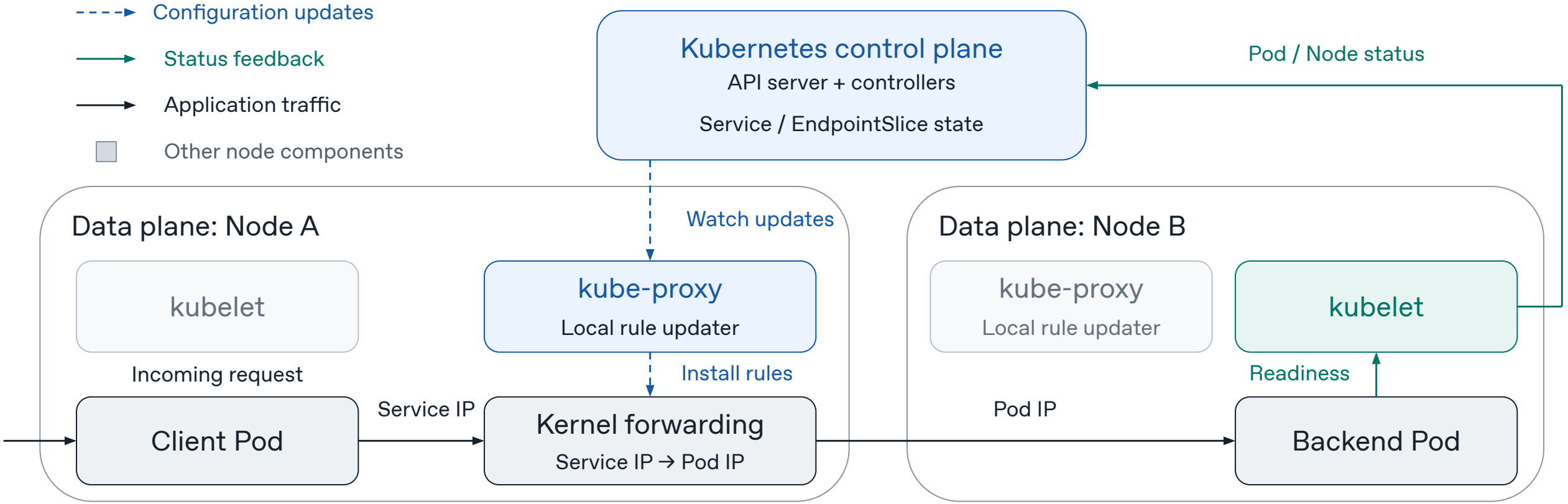
Which GPU engine should serve each request from a CPU cluster?

Input: CPU cluster + model. Output: one eligible engine in a GPU cluster



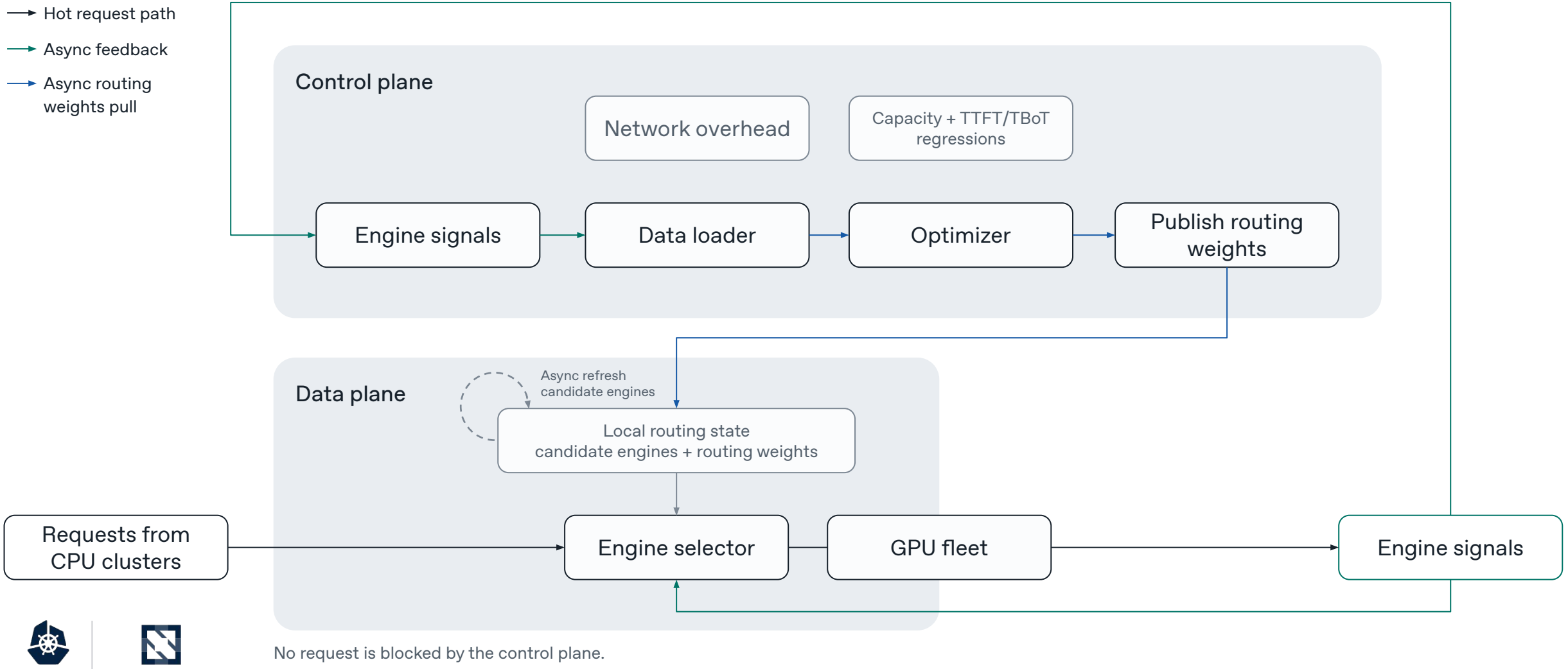
K8s: Control plane and data plane separation

Kubernetes Service networking with kube-proxy



ILB: Control plane and data plane separation

A familiar Kubernetes pattern: control decisions outside the request path



No request is blocked by the control plane.

Three paths through the system

One synchronous request path; two asynchronous control paths

01.
Inference request path

synchronous, per request

Request from CPU cluster → Engine selection + route → GPU engine serve request

02.
Engine signal path

async feedback

GPU engine signals → Aggregate signals → Control/data plane ingest

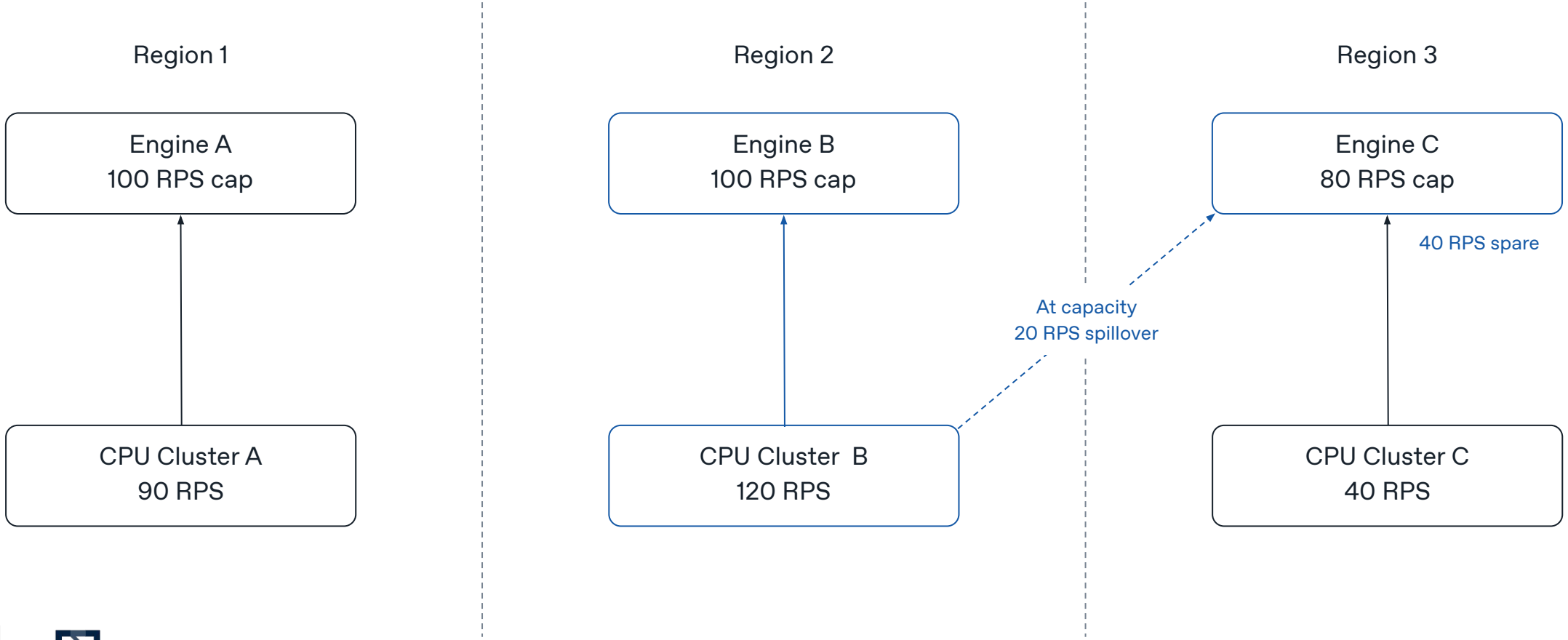
03.
Routing weight path

async pull

Control plane computed routing weight → Publish routing weights → Data plane weights updated

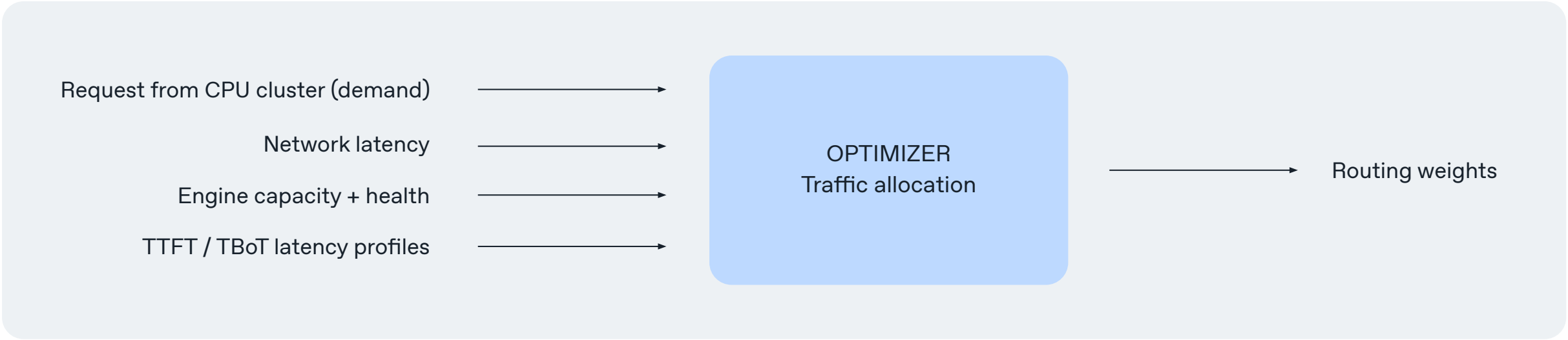
Why nearest-only fails

Demand and GPU capacity are not geographically balanced



Optimizer: From signals to weights

Compute how each CPU cluster should route traffic across engines



Optimization goal

Minimize expected end-to-end latency across all routed traffic.

Balance network distance with engine-side latency.

Hard constraints

Route all demand

Stay within effective engine capacity

Keep routing weights non-negative

Protection mechanisms



Penalties



Retry with cap



Load shedding



KubeCon + CloudNativeCon

North America 2026

OpenAI