



Smart Routing at Scale

How Spotify's xDS Control Plane Cut 75% of Cross-Zone Traffic

Contents

- Introduction
- The Heuristic
- Implementation
- Stories from Production

Introduction



Audio Streaming · Music · Podcasts · Audiobooks · 180 Markets

2,900+

deployments/day

1.8T+

events/day

2,900+

microservices

3.5 TB/s

bandwidth

11M

req/s peak

1.8M+

K8s pods

What are zones?

One Zone



Multiple Pods

One Region



Multiple zones

Spotify's fleet



2M+ nodes mesh

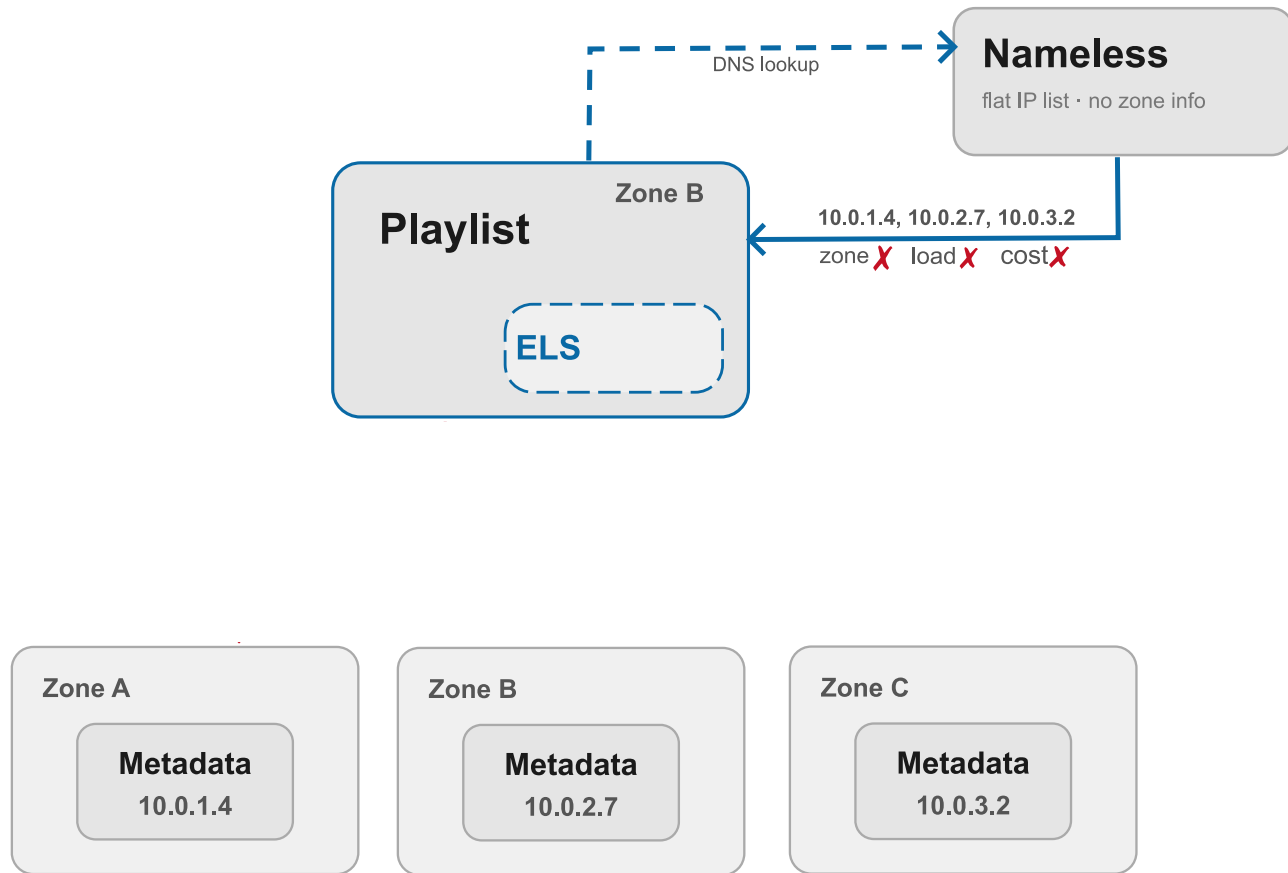
An availability zone is an isolated infrastructure location inside a region.

Cross-zone traffic = \$\$\$

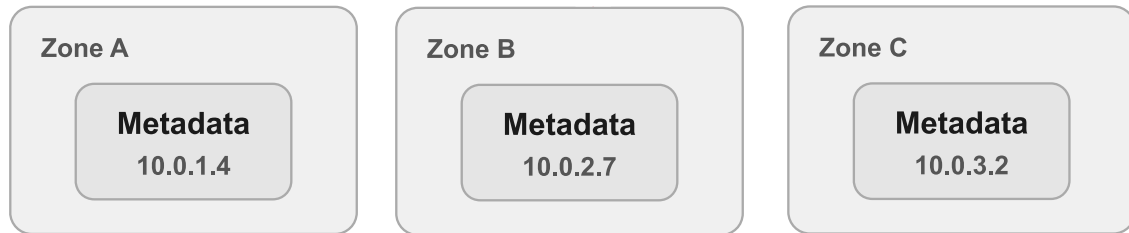
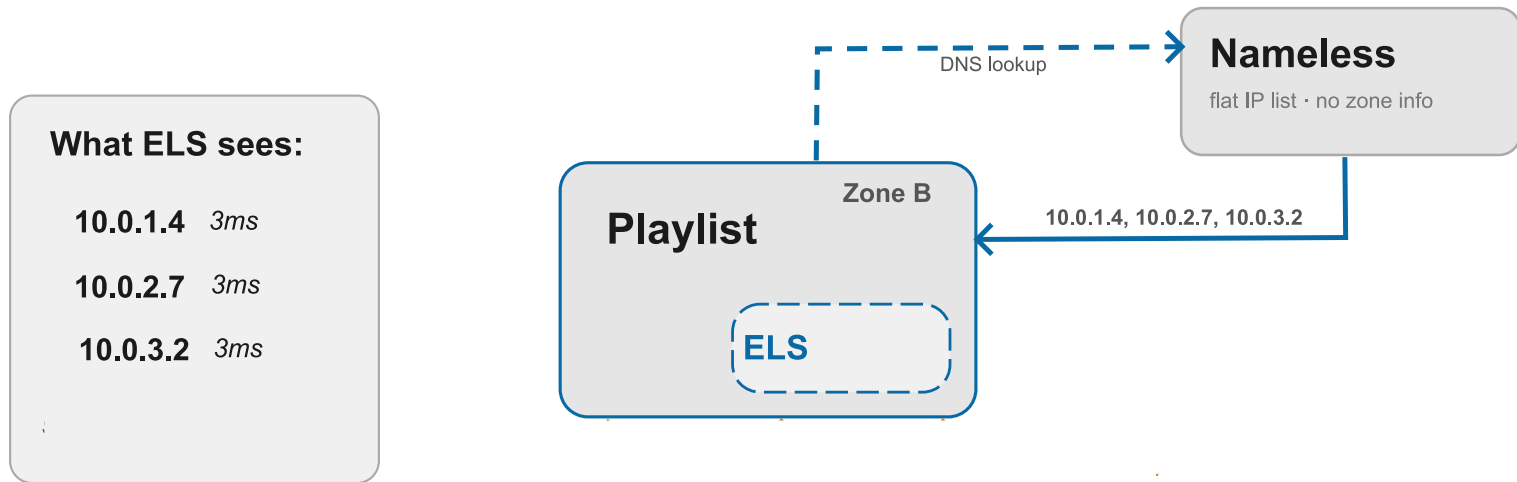


The optimization challenge — keep as much traffic as possible within the same zone whilst maintaining high availability.

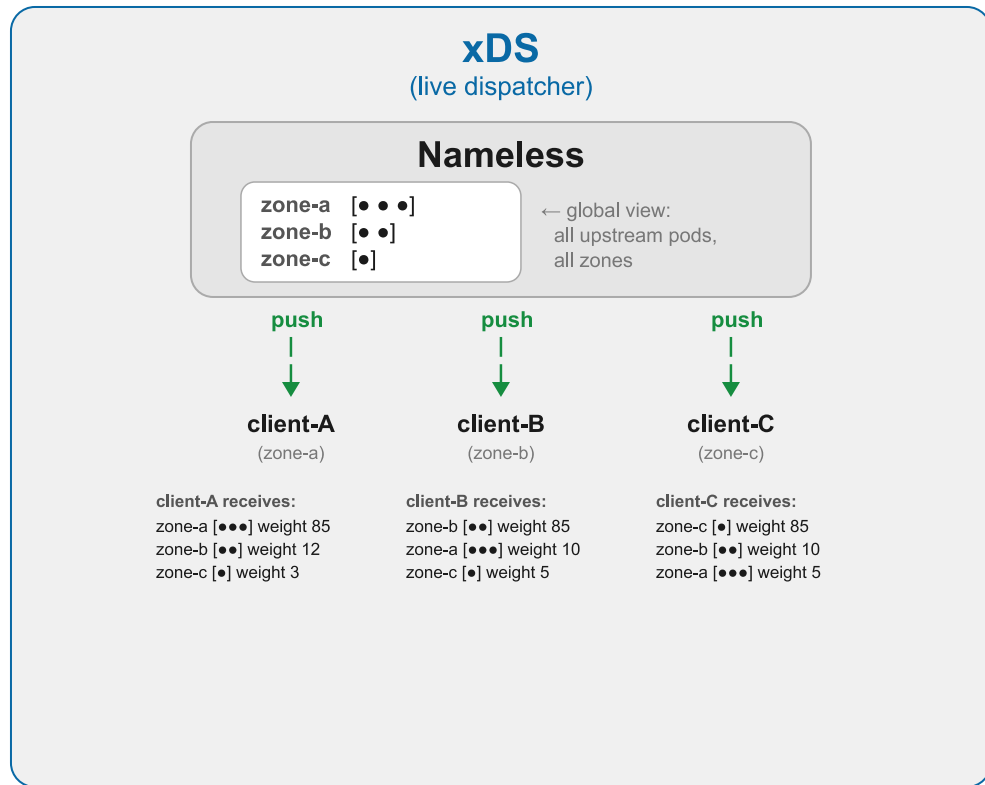
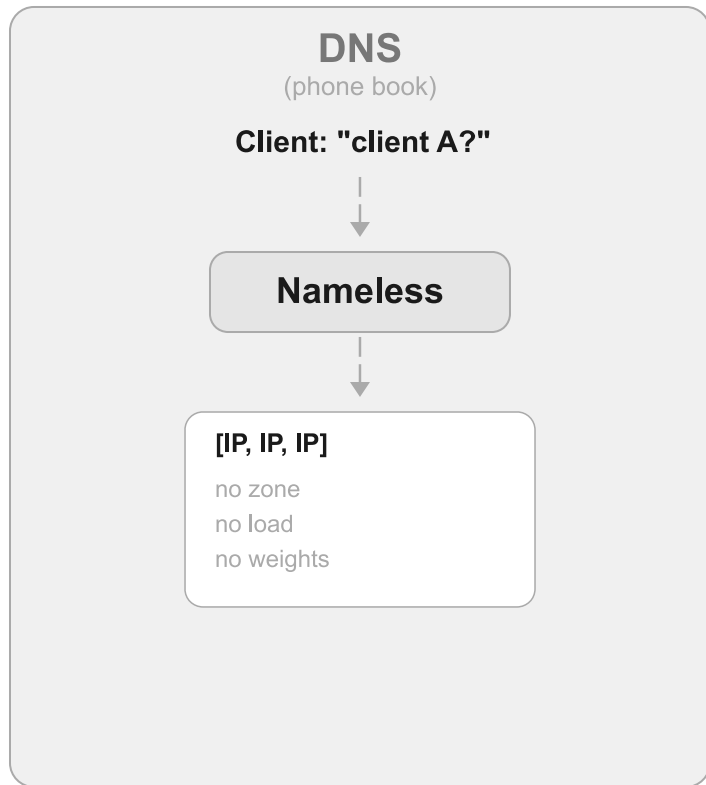
Service Discovery



Service Discovery

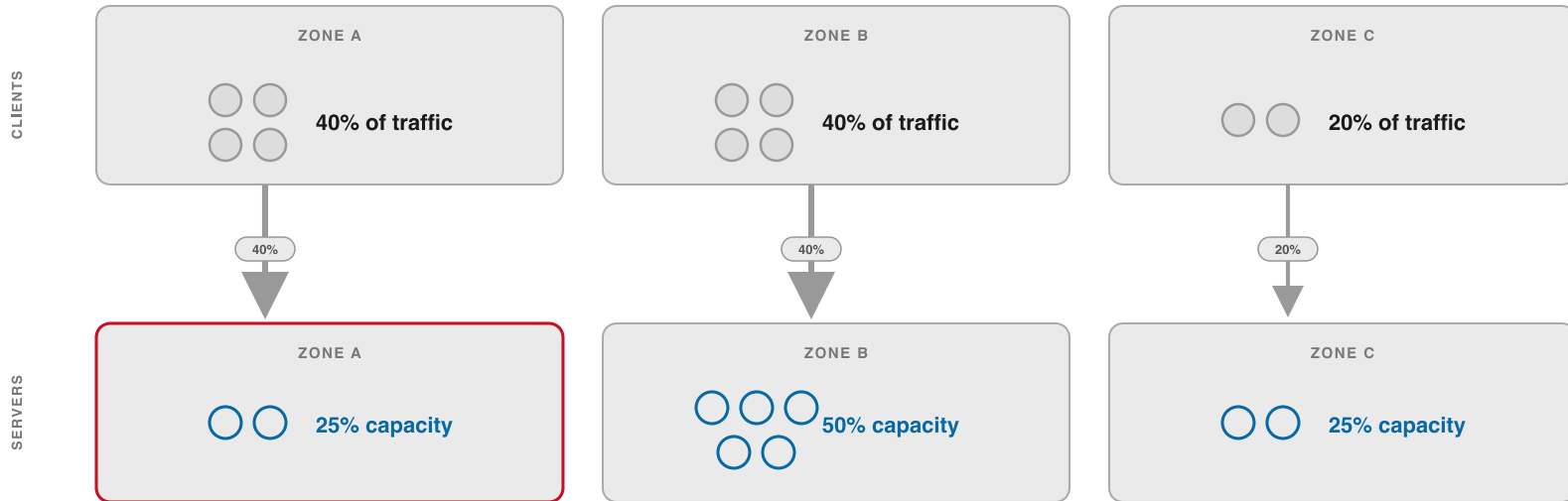


Nameless as an xDS Control Plane

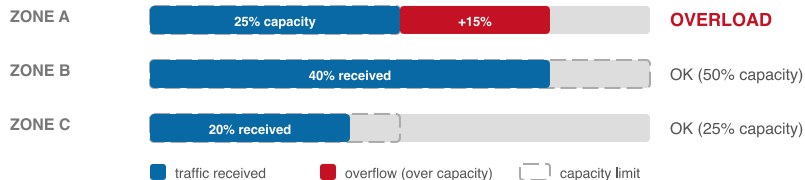


The Heuristic

Naive: route everything locally

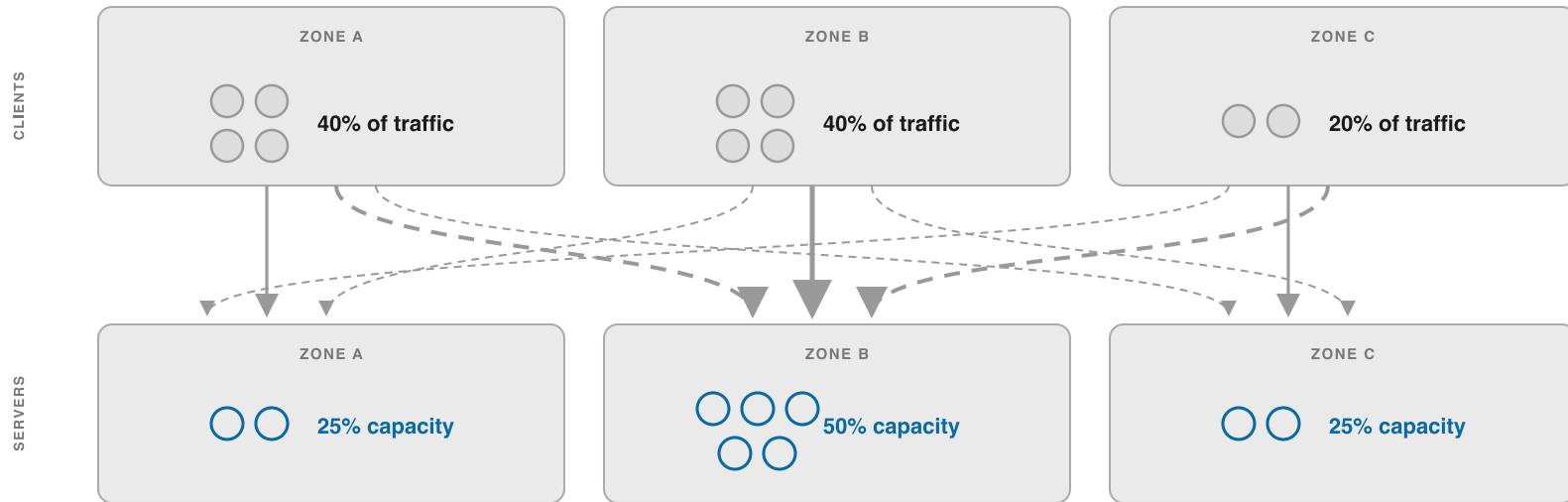


TRAFFIC RECEIVED vs CAPACITY

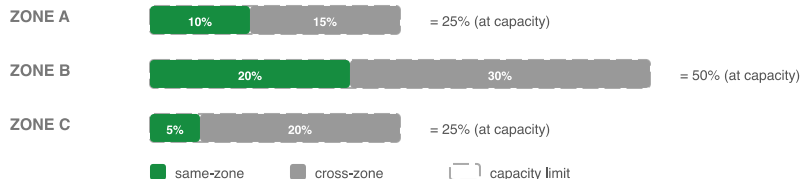


Zone A receives 40% but can only serve 25%. Constraint violated.

Safe but suboptimal: route by capacity

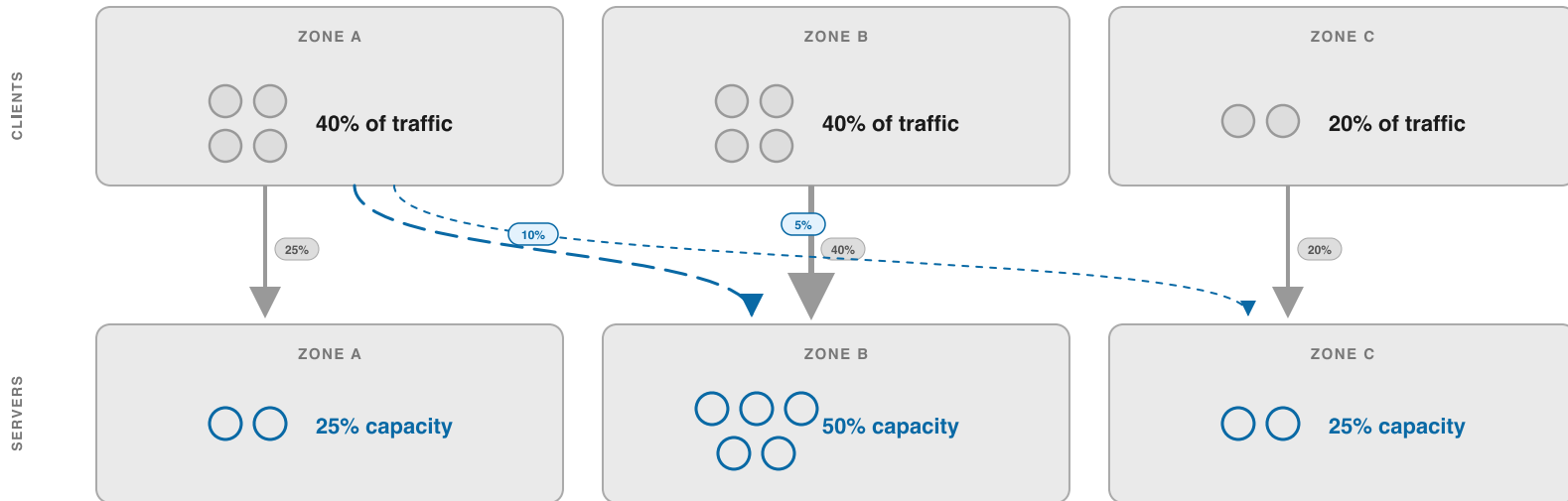


TRAFFIC RECEIVED (same-zone vs cross-zone)

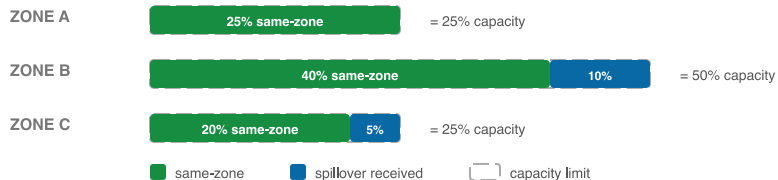


No overload — but only 35% same-zone. 65% of traffic crosses zones unnecessarily.

Optimal: fill local, spill the rest



TRAFFIC RECEIVED (same-zone vs spillover)



No overload AND 85% same-zone traffic. Every bar fills exactly to capacity.

How does Envoy calculate the weights?

STEP 1 — SAME-ZONE WEIGHT

For each zone, send as much locally as capacity allows:

$$w_{\text{local}} = \min\left(1, \frac{\text{capacity}}{\text{demand}}\right)$$

STEP 2 — DISTRIBUTE SPILLOVER

The remaining traffic goes to other zones, proportionally to their residual capacity:

$$\text{spill}_{x \rightarrow y} = (1 - w_{\text{local}}) \times \frac{\text{residual}_y}{\sum \text{residual}}$$

Total spillover = total residual capacity. Self-balancing.

EXAMPLE: ZONE A (40% DEMAND, 25% CAPACITY)

STEP 1 Same-zone weight:

$$\min\left(1, \frac{25}{40}\right) = 62.5\%$$

$$62.5\% \times 40\% = 25\% \text{ locally}$$

STEP 2

$$37.5\% \times 40\% = 15\% \text{ to spill}$$

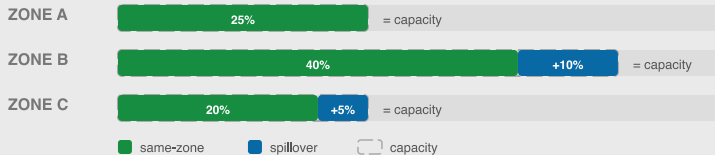
Zone B residual: 50% - 40% = 10%

Zone C residual: 25% - 20% = 5%

A → B: 10%

A → C: 5%

RESULT: TRAFFIC RECEIVED PER ZONE



Every bar fills exactly to capacity. 85% same-zone.

Mathematical foundations

OPTIMISATION PROBLEM

Minimize the number of requests sent cross-zone

subject to: no zone receives more requests than it can serve (capacity constraint)

THEOREM 1 — NO OVERLOAD

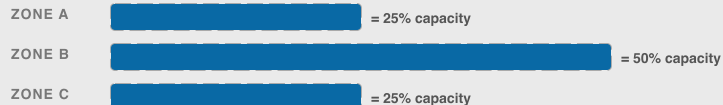
CONDITIONS

Downstream host dist. = traffic dist.

Upstream host dist. = capacity dist.

GUARANTEE

Every zone receives exactly its capacity share of total traffic.



INTUITION

Spillover is distributed proportionally to residual capacity. Total spillover = total residual, so every zone fills to exactly its capacity.

THEOREM 2 — OPTIMALITY

CONDITIONS

Downstream host dist. = traffic dist.

Upstream host dist. = capacity dist.

GUARANTEE

The greedy heuristic achieves a global optimum.

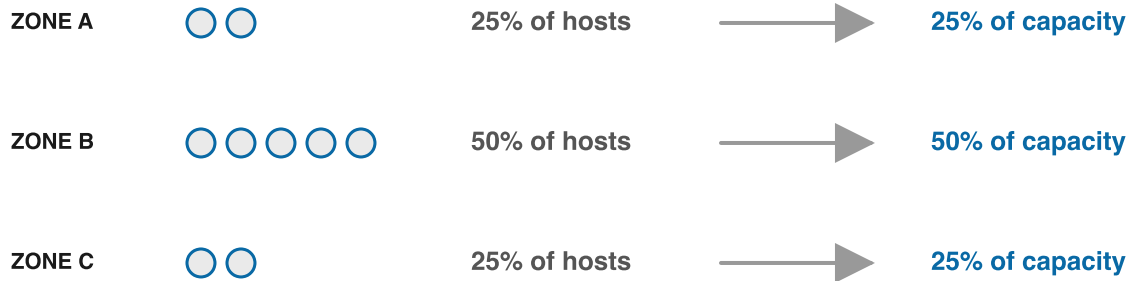


INTUITION

Each zone already sends the maximum it can locally. Any alternative optimum has the same same-zone traffic — it can only differ in how spillover is distributed.

Every host in a zone has equal capacity.

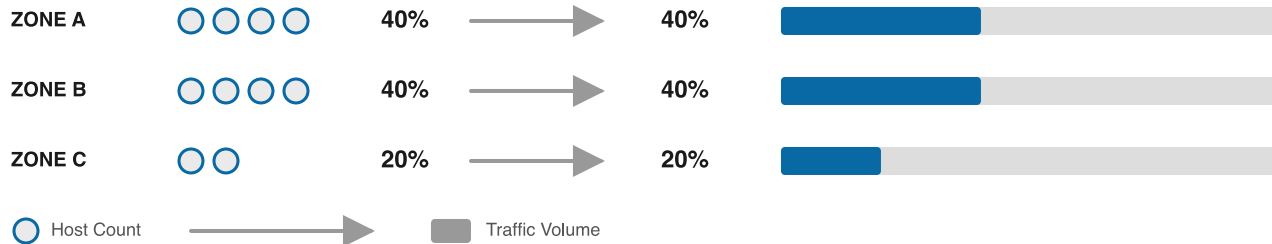
The capacity of a zone is proportional to its host count.



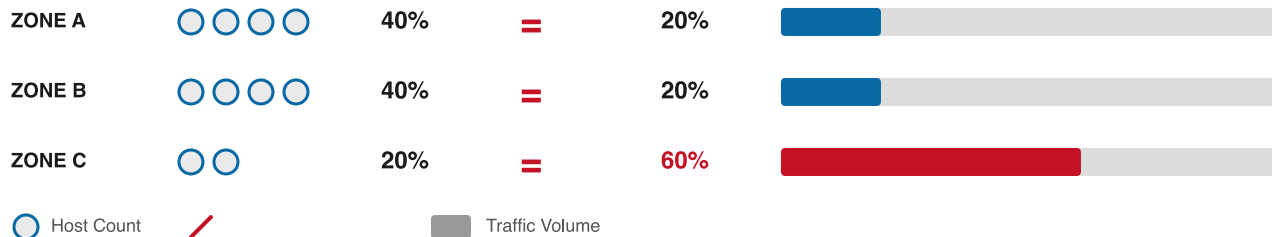
Reasonable: same code, same hardware, same config.

Does host distribution = traffic distribution?

ASSUMPTION



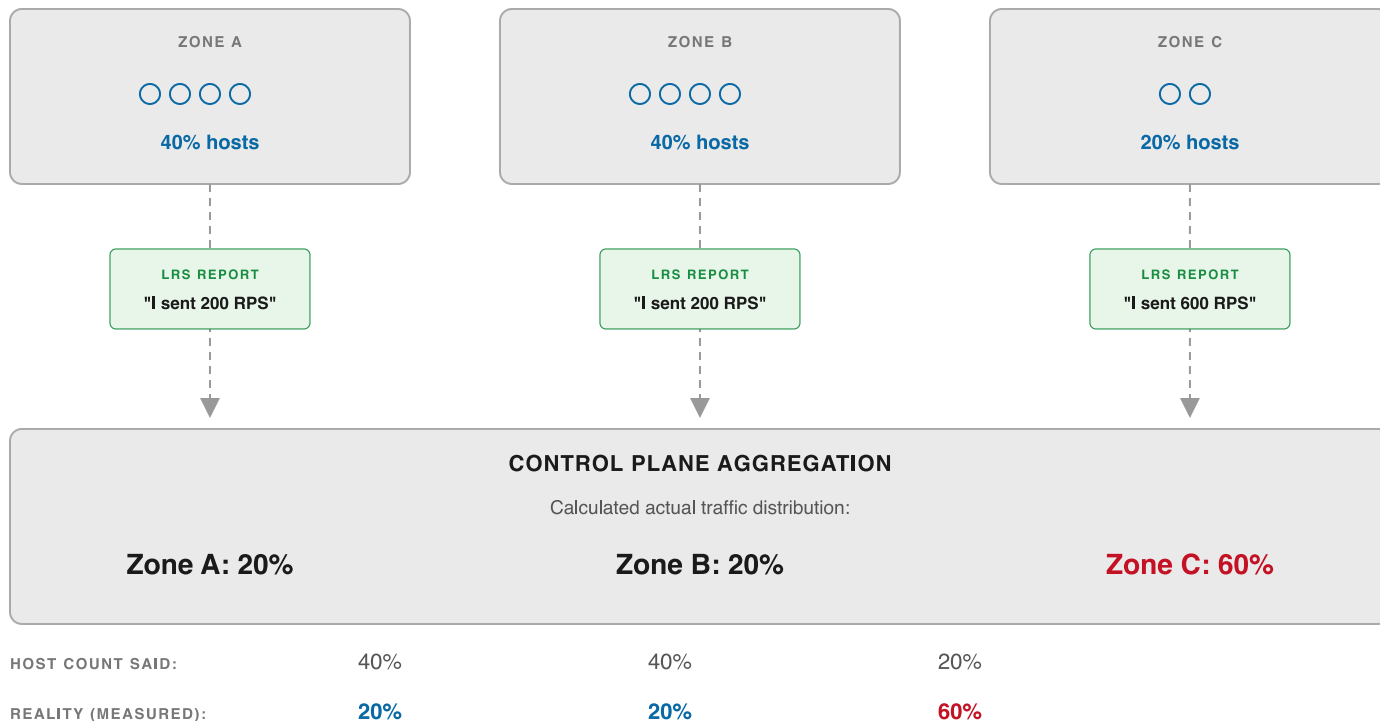
REALITY AT SCALE



Zone C: 60% of traffic hitting 25% of upstream capacity

This is not suboptimal. This is an overload.

Instead of assuming hosts = traffic, measure it.



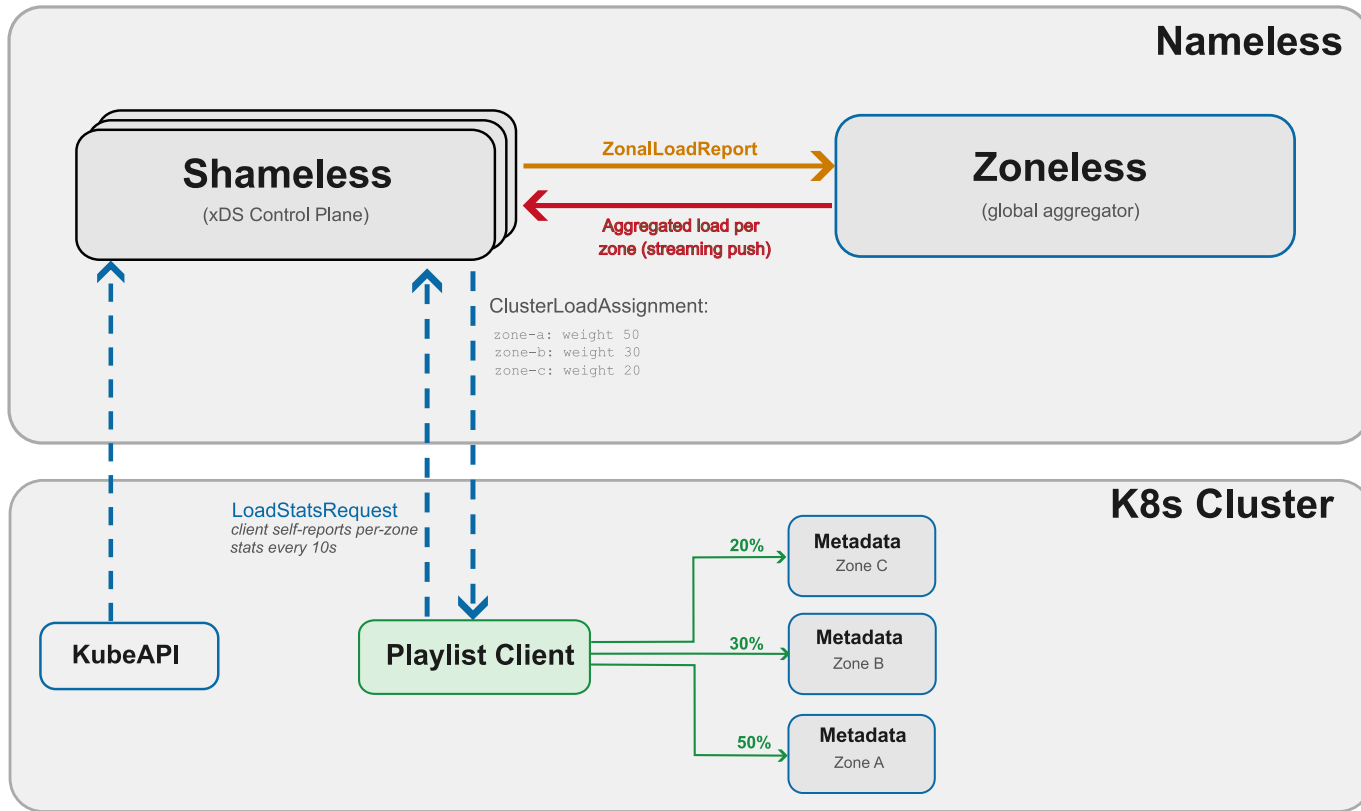
One substitution in the formula:
Replace host fraction with measured traffic fraction.

Where we are now

	Reliability (no overload)	Optimality (max same-zone)
Envoy's heuristic	REQUIRES - downstream host dist. = downstream traffic dist. - upstream host dist. = capacity dist.	REQUIRES - downstream host dist. = traffic dist. - upstream host dist. = capacity dist.
Our heuristic (+ LRS measurement)	REQUIRES upstream host dist. = capacity dist.	REQUIRES upstream host dist. = capacity dist.

Implementation

System Architecture



Architectural Trade-offs

✗ PubSub PoC

✓ gRPC bidirectional streaming
(Zoneless)

The math:

100k clients × every 20s = **430M** messages/day
× 10 Shameless subscribers = **4B+** deliveries/day
× GCP 1KB minimum billing unit = 

✗ Embedded in Shameless

✓ **Separate service**

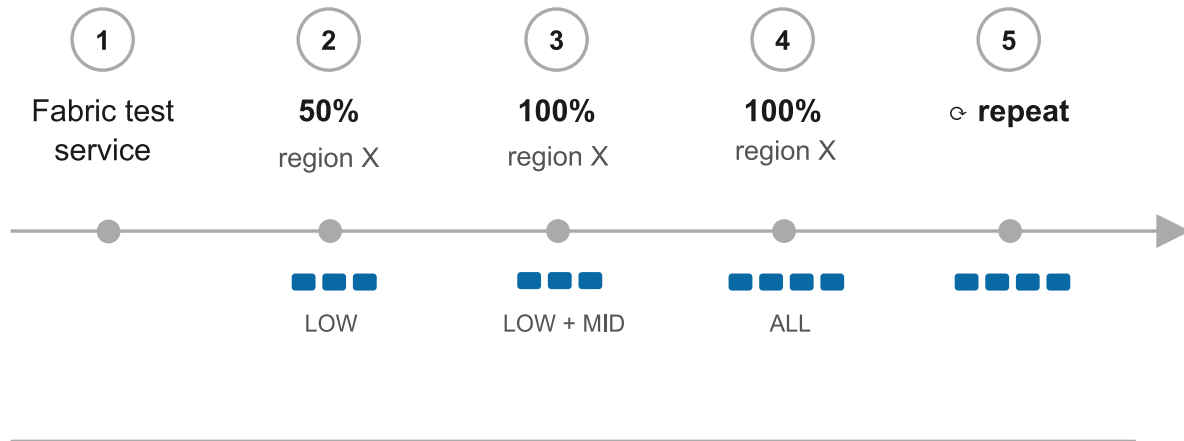
operational isolation · focused responsibility



Zoneless unavailable → falls back to host-count distribution automatically

Shameless is a high availability, zero downtime service.

Rollout Strategy



Single config change → revert any service or full fleet instantly

“Never needed”

Cost impact on individual services

Cross-zone networking cost for one high-volume component

82%

reduction in cross-zone networking cost

BEFORE



AFTER



Component owners have one less cost to worry about.

Stories from Production

Remember this?

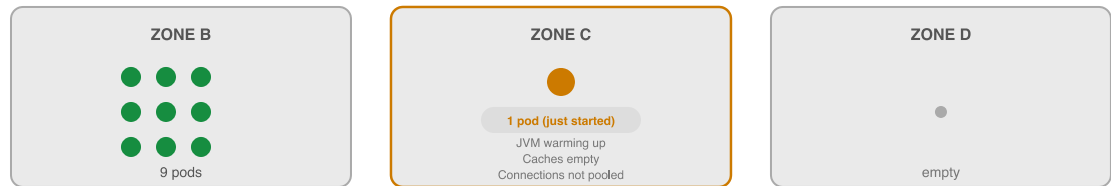
Heuristic	Reliability (no overload)
Our heuristic	Requires: ● upstream host dist. = capacity dist. <i>"Reasonable: same code, same hardware, same config."</i>

Service X · us-east1 · ~300k req/s

ZONE DISTRIBUTION (BEFORE)



SCHEDULER PLACES A POD IN ZONE C



WHAT THE MODEL SEES

Zone C: $1/10 = 10\%$

"Send 10% of traffic here"

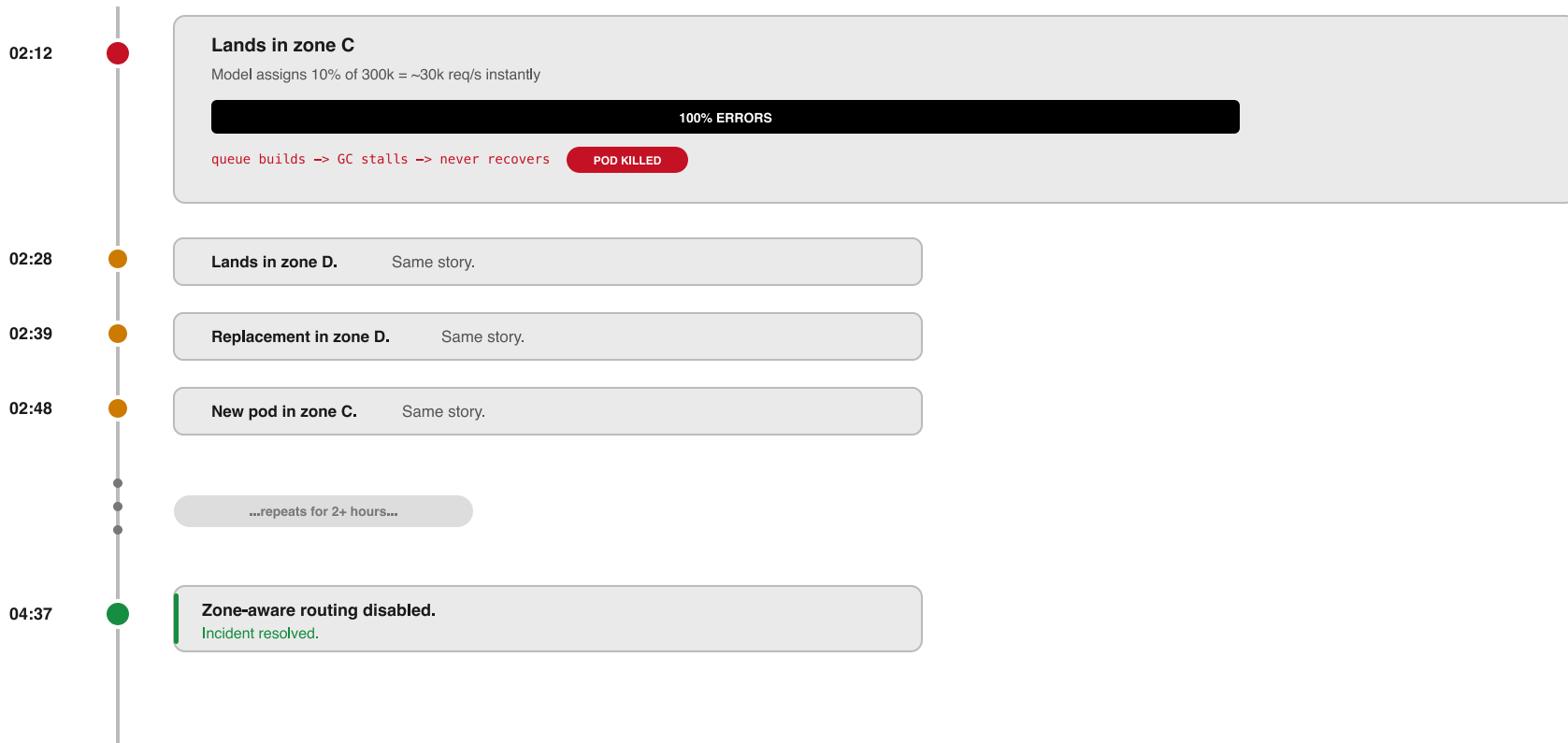
WHAT'S ACTUALLY HAPPENING

Real capacity ~ 0%

~30k req/s to a cold pod

The Death Spiral

Service X · zone-aware routing incident



The assumption breaks

upstream host dist. $\neq$ capacity dist.

CAUSES

Cold starts

Empty caches, JIT not optimized, connections not pooled

MINUTES

Noisy neighbors

Shared resources contended

HOURS

Hardware differences

Different GPU types for same workload

PERMANENT

IMPACT

In large zones:

Leaf LB routes around it.

No problem.

In small zones:

Leaf LB has no alternative.

Overload.

The zone-aware routing layer must detect and compensate,

because the leaf load balancer cannot.

capacity(X): detecting degradation

Problem: A zone can degrade due to noisy neighbors, GC pressure, or resource contention.

Solution: Use queue depth to scale down effective capacity. Hosts that can't keep up count for less.

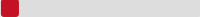
$$\text{capacity}(X) = H_X \times \frac{I_X}{I_X + Q_X}$$

VARIABLES (from load reports, every 10 seconds)

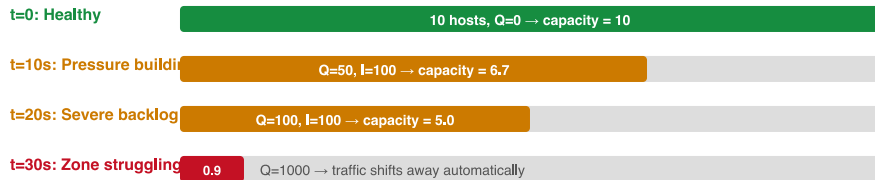
H_X = healthy hosts in zone X I_X = issued requests

Q_X = queued (outstanding) requests

EXAMPLE: 100 ISSUED REQUESTS

Queued	$I/(I+Q)$	Effect	Visual
0	1.00	full capacity	
10	0.91	-9%	
50	0.67	-33%	
100	0.50	-50%	
1000	0.09	-91%	
0 ($I=0$)	H_X	fallback to static	

HOW IT WORKS: ZONE DEGRADES OVER TIME



Queue depth is a leading indicator — it detects stress before errors occur.

When Q=0 and I=0 (cold start / zoneless restart): capacity defaults to host count (static fallback).

ramp(X): slow-start protection

Problem: New capacity is not the same as existing capacity. Cold hosts need time to warm up.

Solution: A zone must earn its capacity by handling traffic successfully. Ramp compares success share to host share.

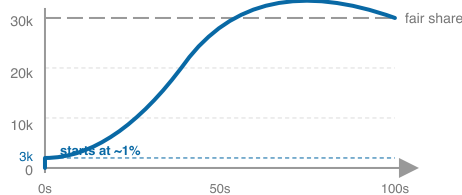
$$\text{ramp}(X) = \max\left(0.05, \frac{S_X \cdot \sum H_Z}{S_X \cdot \sum H_Z + H_X \cdot \sum S_Z}\right)$$

S_X = successful requests H_X = healthy hosts

STATIC MODEL: 0 → 30k req/s instantly



DYNAMIC MODEL: gradual ramp over ~100 seconds



CYCLE BY CYCLE: ZONE C ADDS A HOST (300K REQ/S SERVICE)

TIME	WEIGHT	REQ/S	TRAFFIC TO ZONE
0s	1%	3k	
10s	2%	6k	
20s	3%	10k	
40s	7%	21k	
60s	9%	27k	
80s	10%	29k	
100s	10%	30k	

= fair share

3k req/s on a cold pod vs 30k req/s on a cold pod.

How capacity and ramp interact

Effective capacity of zone $X = \text{ramp}(X) \times \text{capacity}(X)$. The two components are a damper, not a brake.

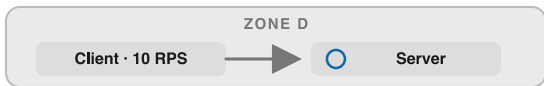
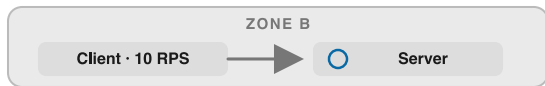
Scenario	capacity(X)	ramp(X)	ramp × capacity	Effect on zone traffic
Healthy, stable No changes, normal traffic	high $Q=0$, full hosts	high successes = fair share	high	Zone gets its fair share
Zone degrades Noisy neighbor, GC storm	↓ drops Q increases	unchanged (reacts next cycle)	↓ drops	Traffic shifts away
Host added Scale-up, deployment	↑ increases $H_X + 1$	↓ drops host share > success share	↑ slightly	Zone gets a bit more traffic Gradually earns the rest
Zone recovers After degradation resolves	↑ recovers Q drains	low initially must rebuild successes	↑ gradual	Traffic returns gradually No thundering herd
Cold start Zoneless restart, no data	H_X (static) $I=0$, $Q=0$	5% floor no successes yet	probe	Small probe traffic, then ramps up from data

Adding a host never decreases a zone's traffic. Ramp acts as a damper, not a brake.

capacity detects problems fast (leading indicator). ramp prevents overreaction (trailing confirmation).

Experimental Results

3 zones · 1 client + 1 server per zone · 30 RPS total · 5s deadline · LRS every 10s



Full zone degradation

Zone B server throttled to 0.1 RPS for entire test

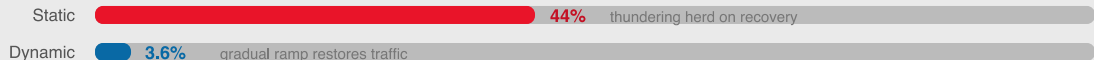
-91% ERRORS



Recovery after degradation

Zone B degrades then recovers mid-test

-92% ERRORS



Slow-starting instance

New pod joins zone B with 2 RPS limit for 60 seconds

-100% ERRORS



TIE Partial degradation (1/3 of instances) — leaf LB handles it

TIE Healthy baseline — 0% errors, no penalty for running dynamic model

The full picture

	Reliability (no overload)	Optimality (max same-zone)
Envoy's heuristic	Requires: - downstream host dist. = traffic dist. - upstream host dist. = capacity dist.	Requires: - downstream host dist. = traffic dist. - upstream host dist. = capacity dist.
+ LRS measurement	Requires: upstream host dist. = capacity dist.	Requires: upstream host dist. = capacity dist.
+ dynamic capacity model	Adapts to real capacity No static assumptions on upstream capacity	Trades some same-zone % for resilience

ASSUMPTION STATUS

downstream host dist. = traffic dist.

removed (measured via LRS)

upstream host dist. = capacity dist.

removed (dynamic capacity model)



Thank you