计算密集型应用以Service Mesh为支点解决分布式问题的探索与实践

京东集团架构师 / 王志龙

个人简介

- 10年+互联网一线研发及架构经验，Kubernetes Contributor，Layotto Wasm Maintainer，专注云原生领域，擅长性能极限优化。
- 曾工作于腾讯、阿里，参与过微信 PaaS 云平台从0到1建设，阿里 Serverless C++ 和 Golang Runtime 研发及落地。
- 目前工作于京东集团搜索与推荐部，负责京东搜推微服务治理和新一代 Serverless 云化平台研发工作。

目录

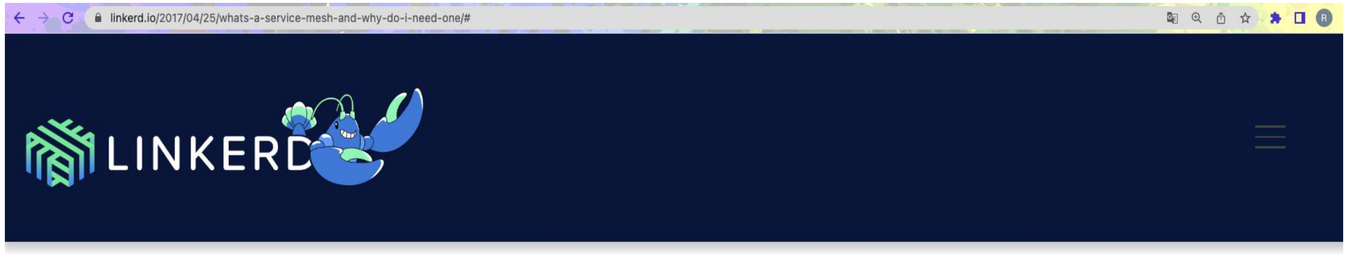
一、Mesh溯源及背景介绍

起源于 Buoyant 内部分享，从落地到概念

专门的一层基础设施；负责可靠传输；轻量的网络代理；对应用程序透明

2016.09.29 Buoyant

2016.01.15 初次发布
2016.09.29 概念诞生
Micro-Service =>
Service Mesh 一脉相承



WHAT IS A SERVICE MESH?

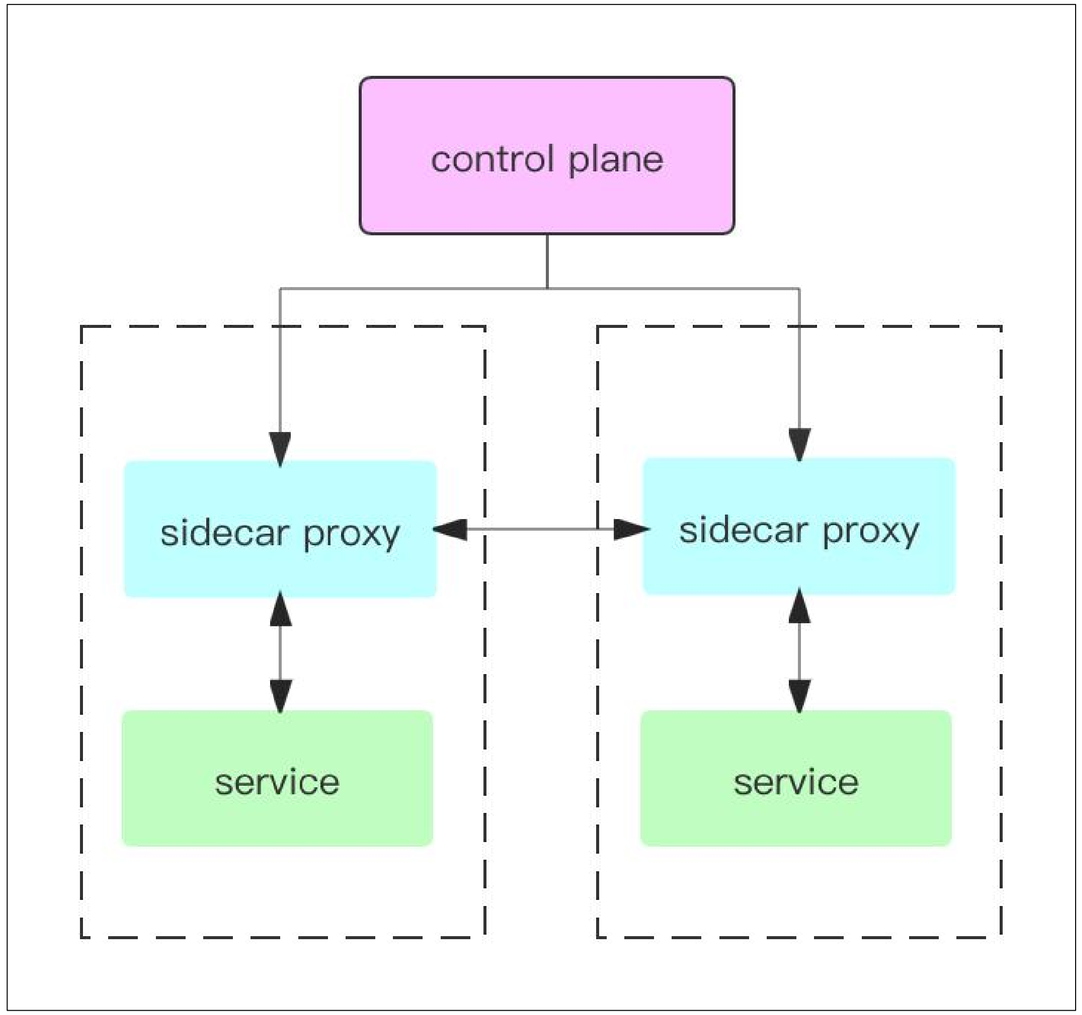
A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware. (But there are variations to this idea, as we'll see.)

William Morgan

Buoyant CEO
服务网格理念的提出者和先行者
以及最早的布道师

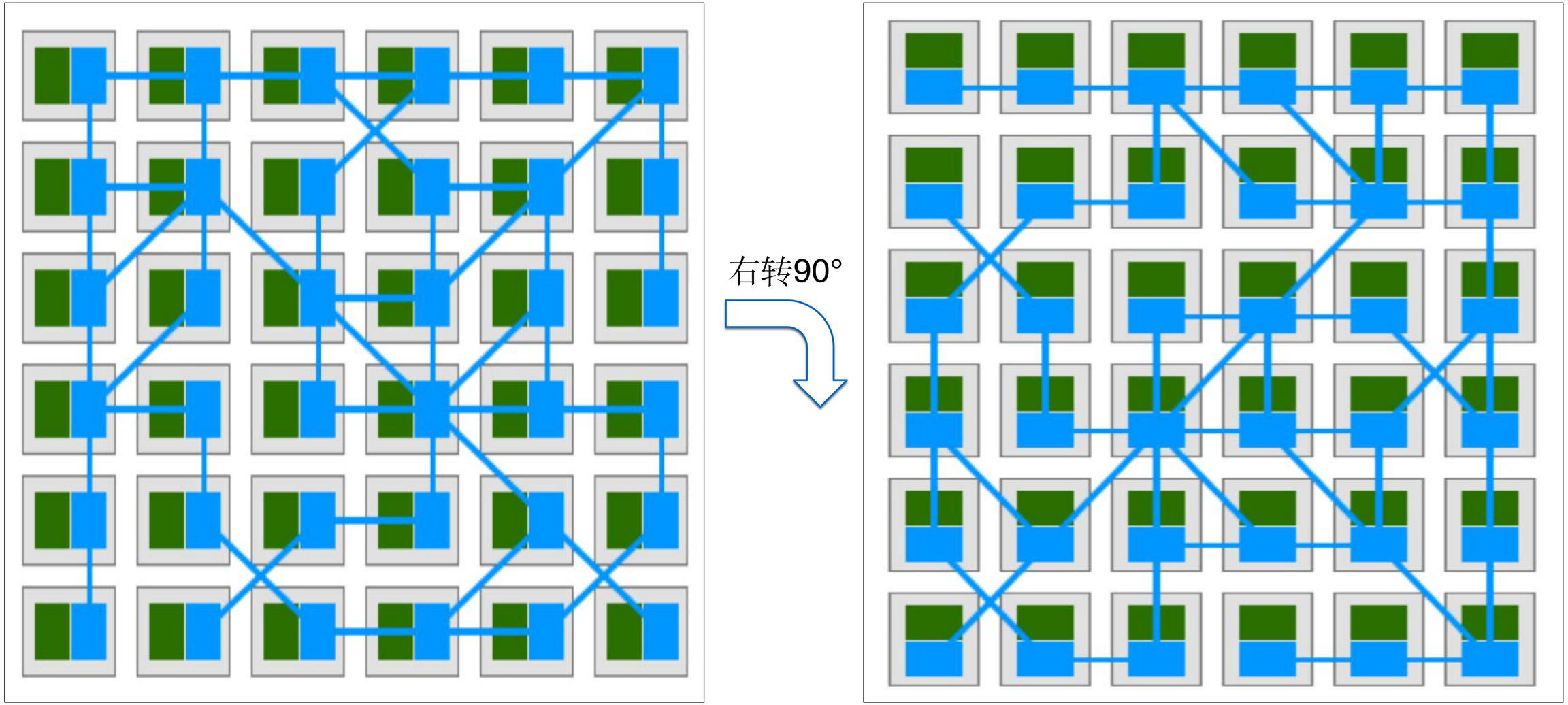
典型形式 —— Sidecar 部署

一般为 Pod 多容器，但是随着 Node 模式的演进，载体多样化起来，但整体形式一致



服务网格和 Sidecar 的关系

绿方块为服务，蓝方块为边车部署的代理，多个 Sidecar 之间的连接和交互组成了 Mesh

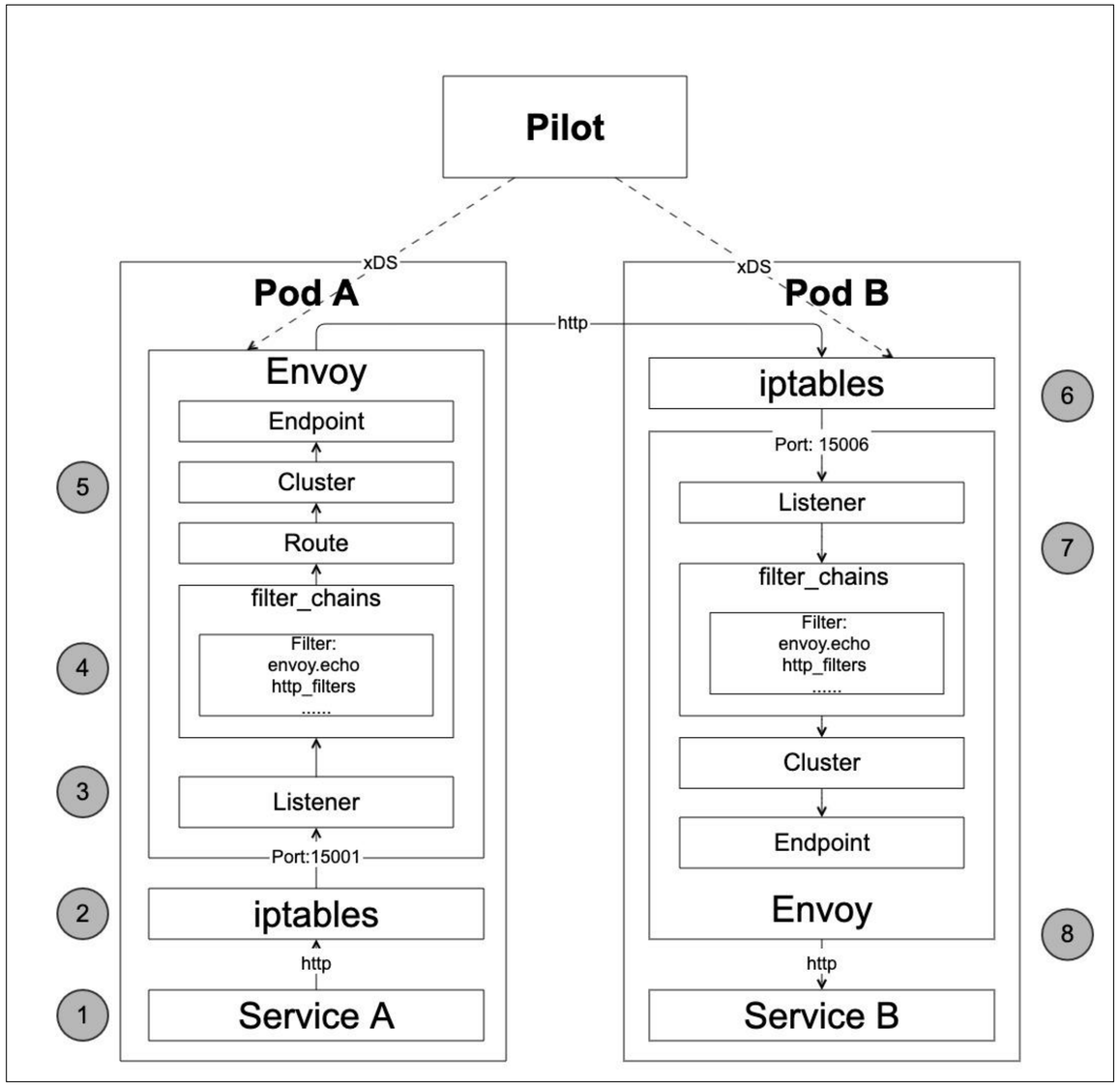


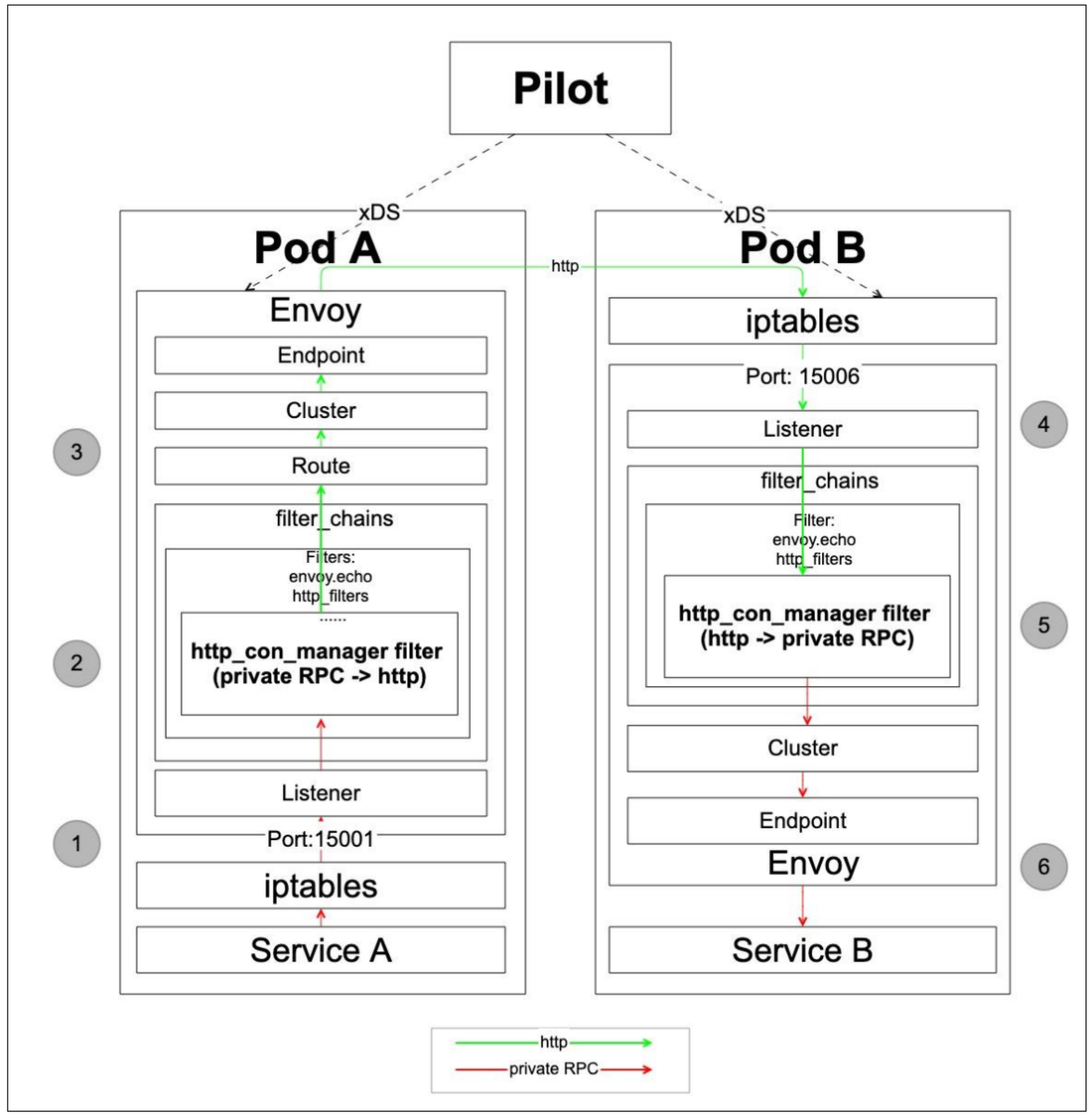
从微信 Svrkit 框架与业务分离方案，回看 Mesh 的意义

基础框架作为承上启下的重要一环：对下充分利用底层系统能力，对上提供灵活可靠的底座

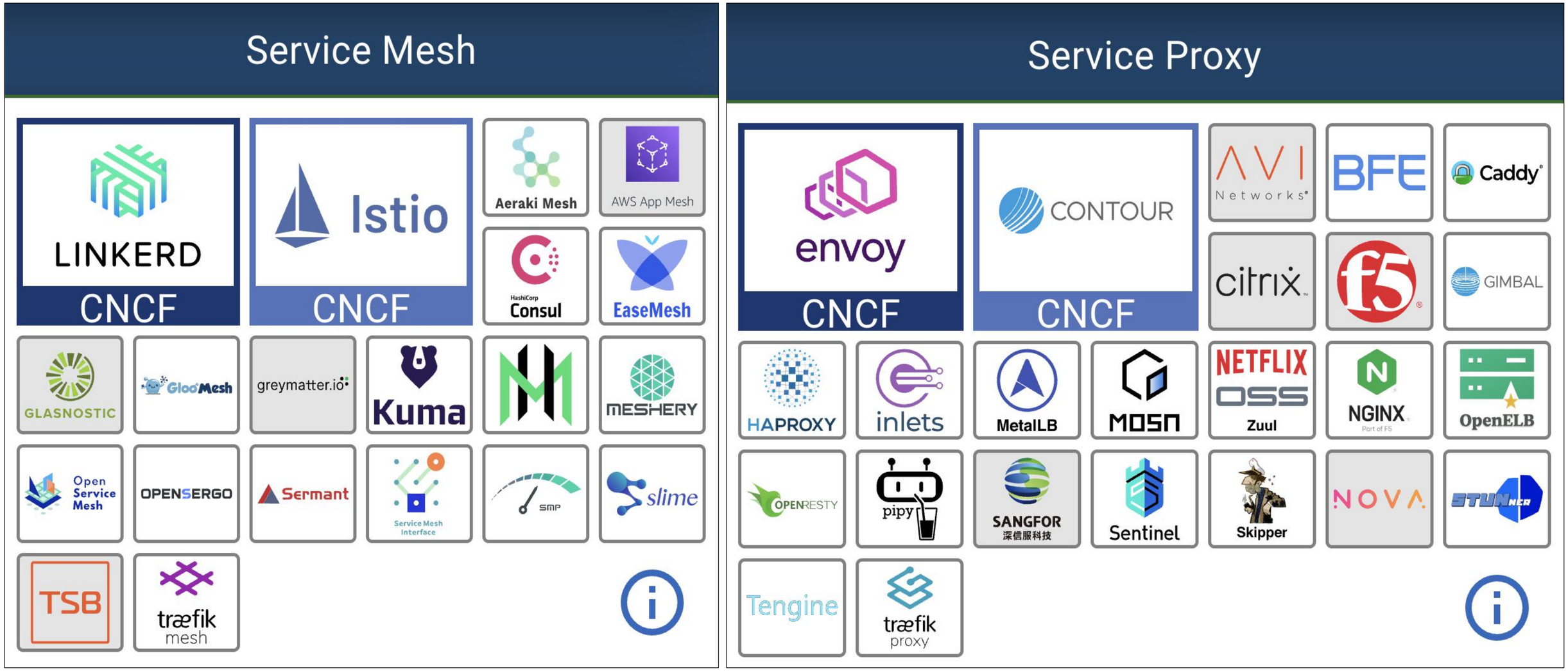
栏目	栏目细分	方案1 框架(bin) + 业务(so)	方案2 框架(so) + 业务(bin)	方案3 框架，业务一起编译	方案4 框架(bin) + 业务(bin)	方案5 框架(bin+支持插件) + 业务（bin)	方案6 框架(bin) + 业务（bin + 多so)	方案7 框架bin + filter bin + 业务bin	方案八 框架bin容器 + 多bin
目标	消除框架侵入	√	√	√	√	√	√	√	√
	消除代码浸入	√	√	√	√	√	√	√	√
	可观察性	中	中	中	高	高	高	高	高
	可测试性	中	中	中	高	高	高	高	高
	可扩展性	中	中	中	较高	高	很高	很高	很高
分离度		中	中	低	较高	高	很高	很高	很高
业务代码修改量		低	低	不需要	低	低	低	低	低
运维修改量		较高	较高	低	中	中	高	高	高
基础模块梳理量		高	高	中	高	高	高	高	高
框架开发量		中	中	低	较高	高	高	高	高
与框架发展契合度		低	低	低	高	高	高	高	高
潜在风险		So符号未定义和符号冲突	符号冲突	-	-	-	So符号未定义和符号冲突	-	-

当年的基于 Envoy HTTP 通道传输私有协议方案





如今的 Service Mesh 百家争鸣，百花齐放



Mesh——协调微服务能力和分布式压力的一个支点

微服务

分散能力

解决系统复杂度问题

逻辑垂直拆分

……

日益复杂多样的需求

高效迭代和极致性能

大促突发大流量挑战

跨部门跨语言联动

共性问题难聚焦复用

小语种服务治理弱

……

分布式

分散压力

解决系统性能问题

物理横向拆分

……

Mesh

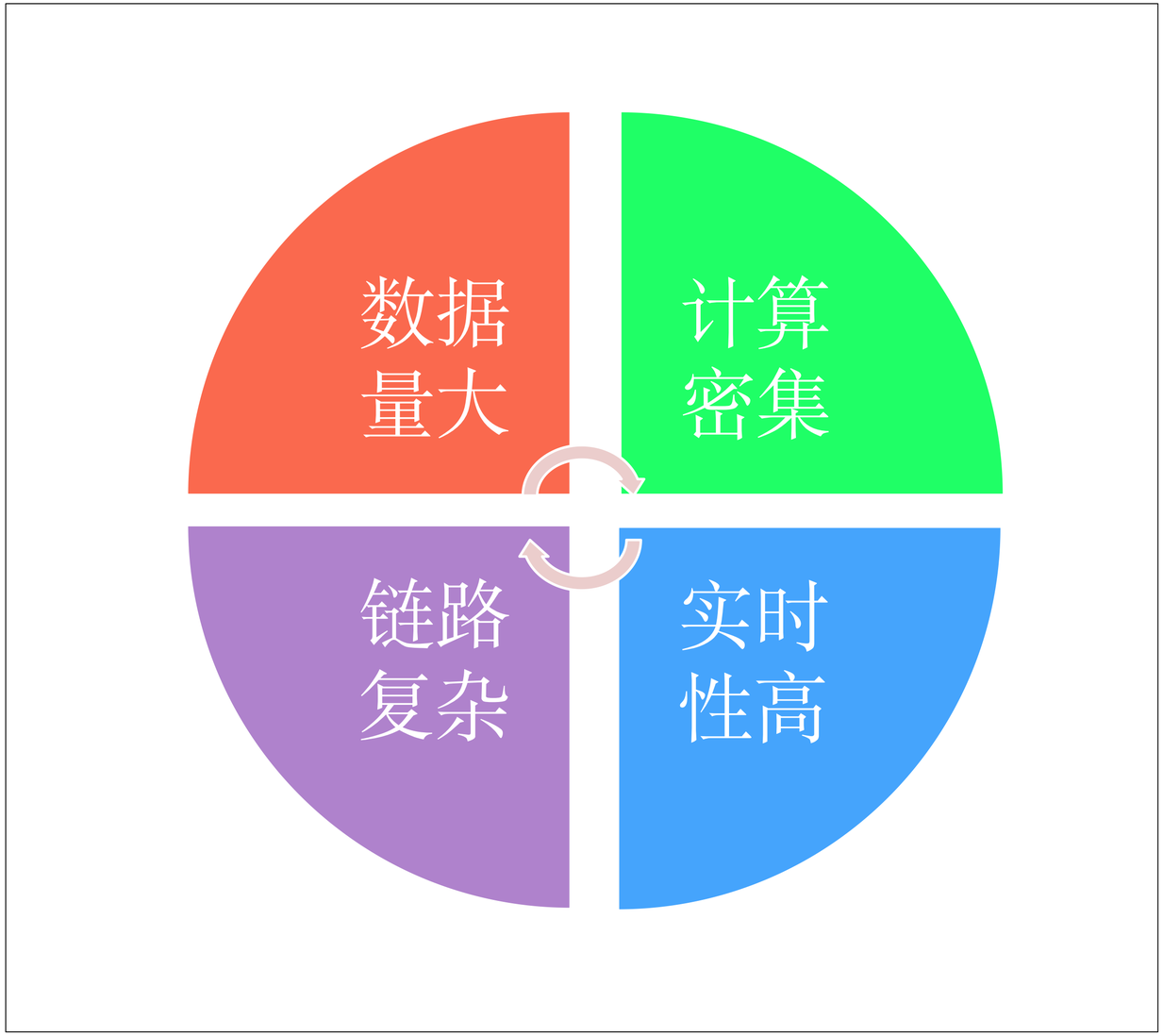
屏蔽分布式系统复杂性

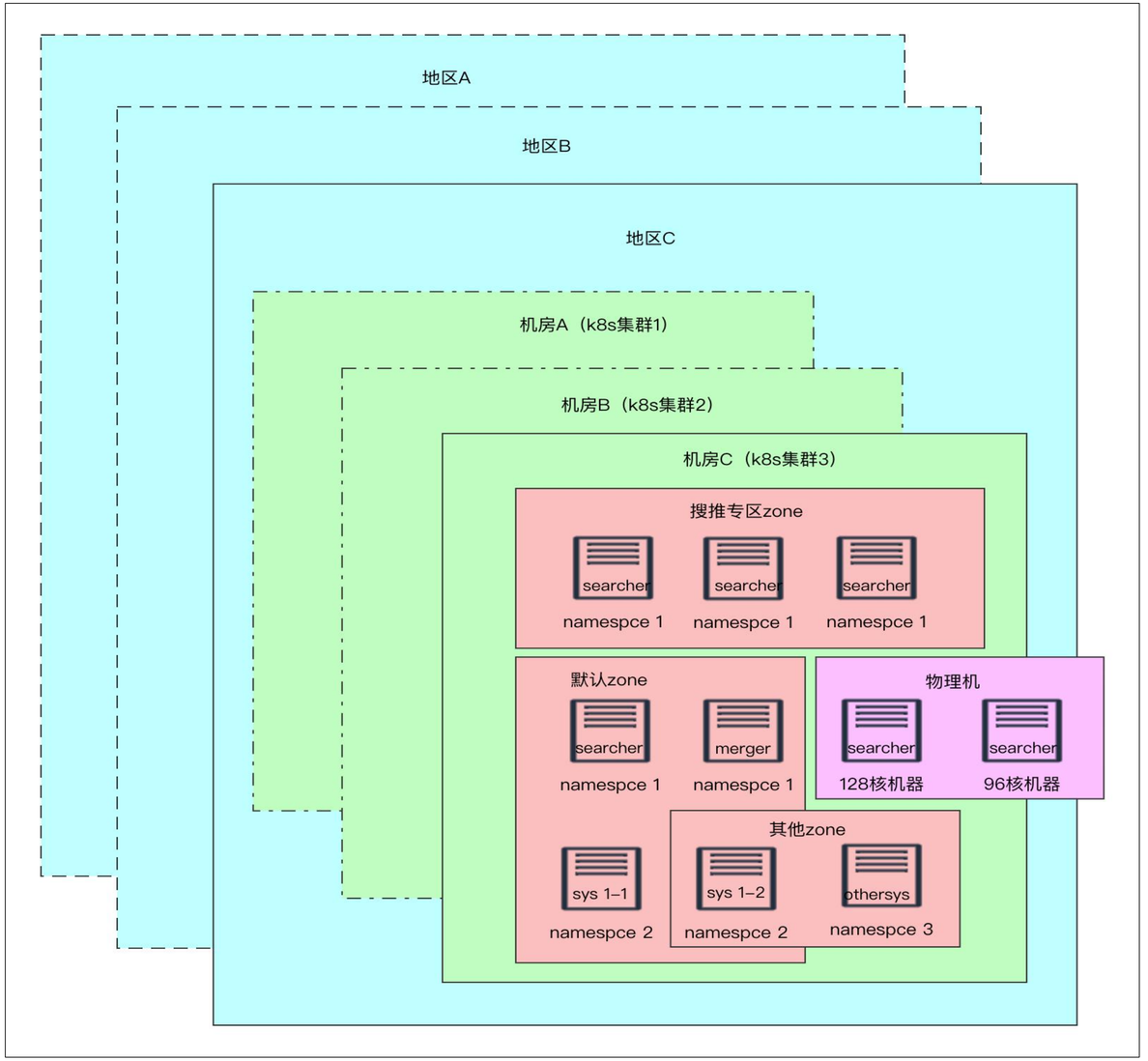
只关注业务逻辑、语言无关、高级流控

对应用透明可以单独高效迭代升级……

二、落地挑战和方案选型

搜推广等计算密集型应用特点及落地挑战





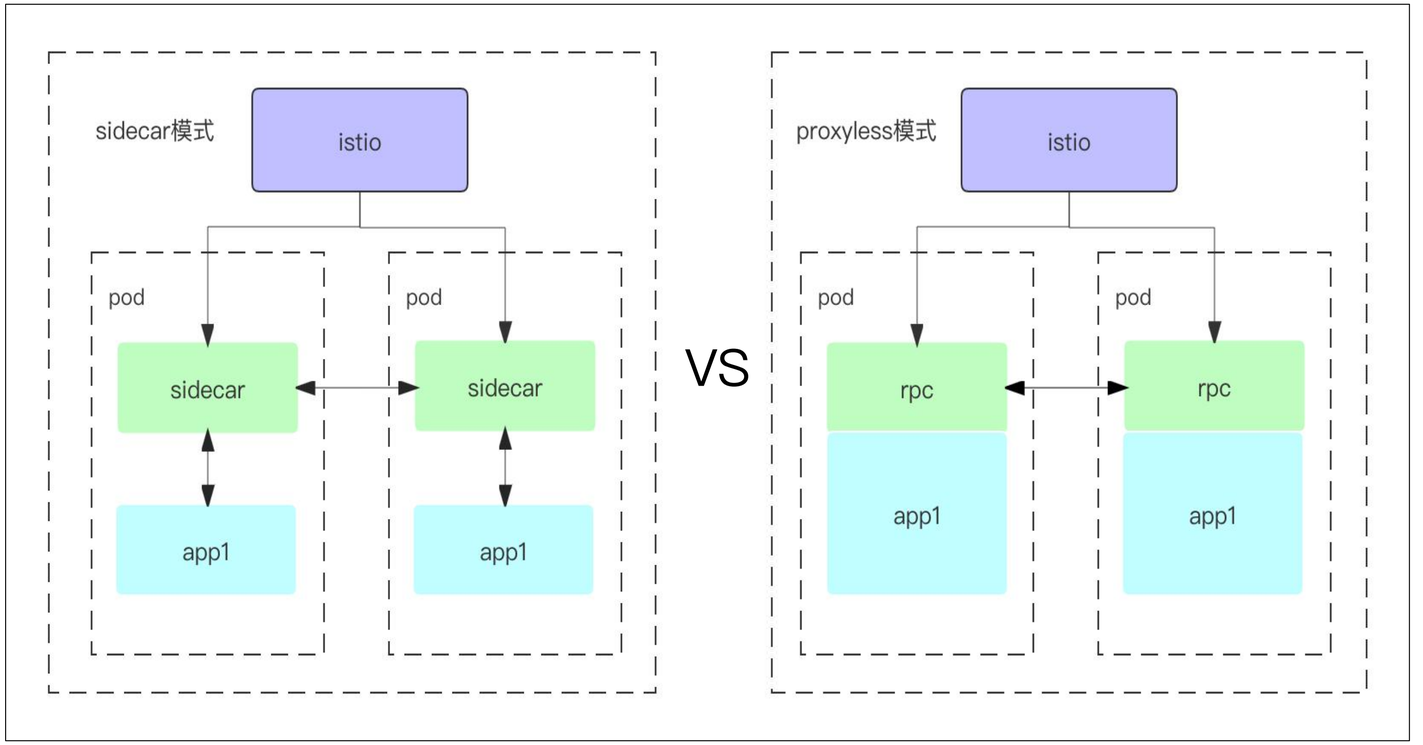
技术选型

Istio

MOSN

envoy

Proxy 性能损耗 vs Proxyless 业务耦合 —— Proxy无损耗？！



MOSN 多协议框架快速落地，中长期使用 MoE “双语”扩展

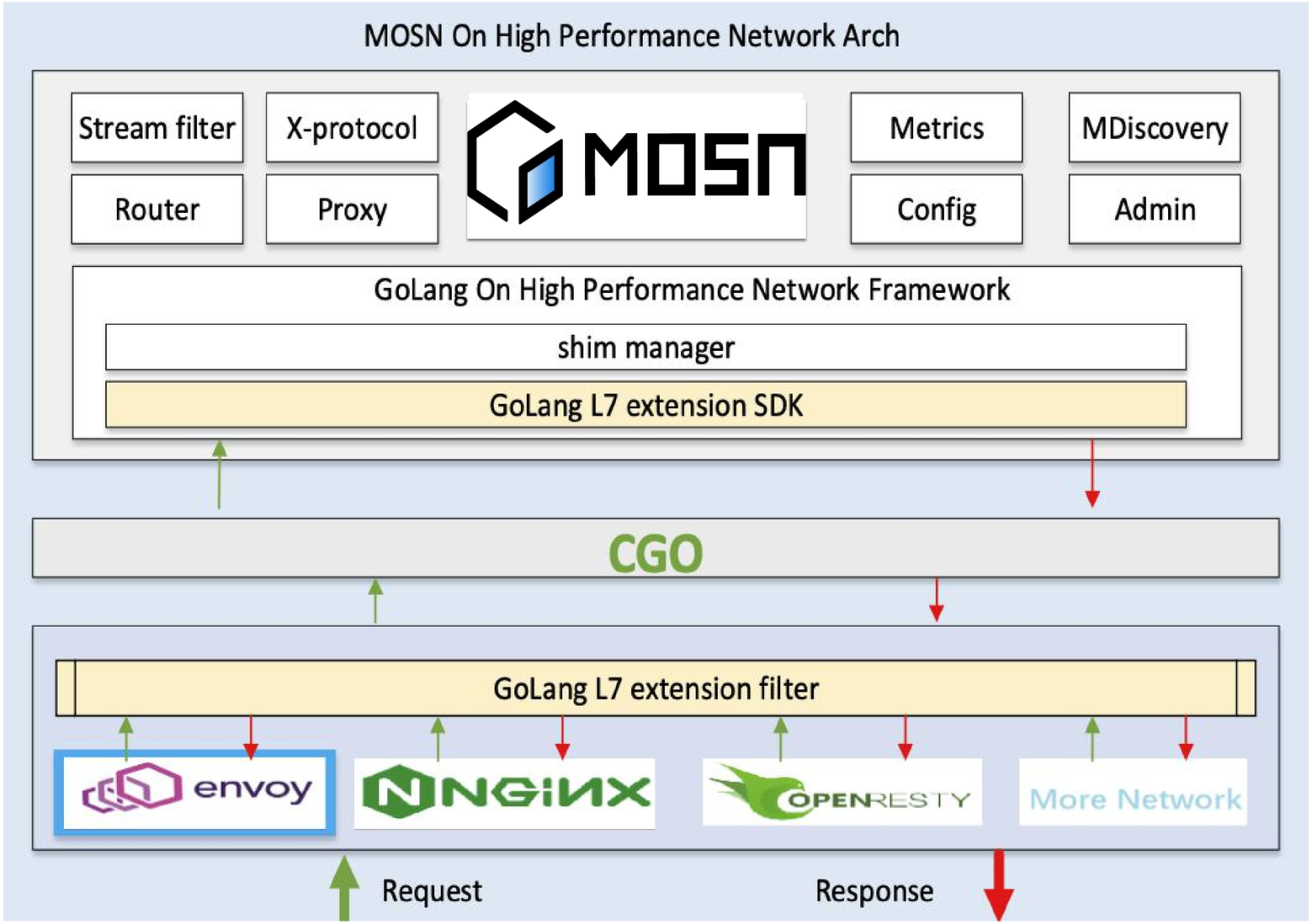
MoE —— Mosn On Envoy

研发效能高
（Golang）

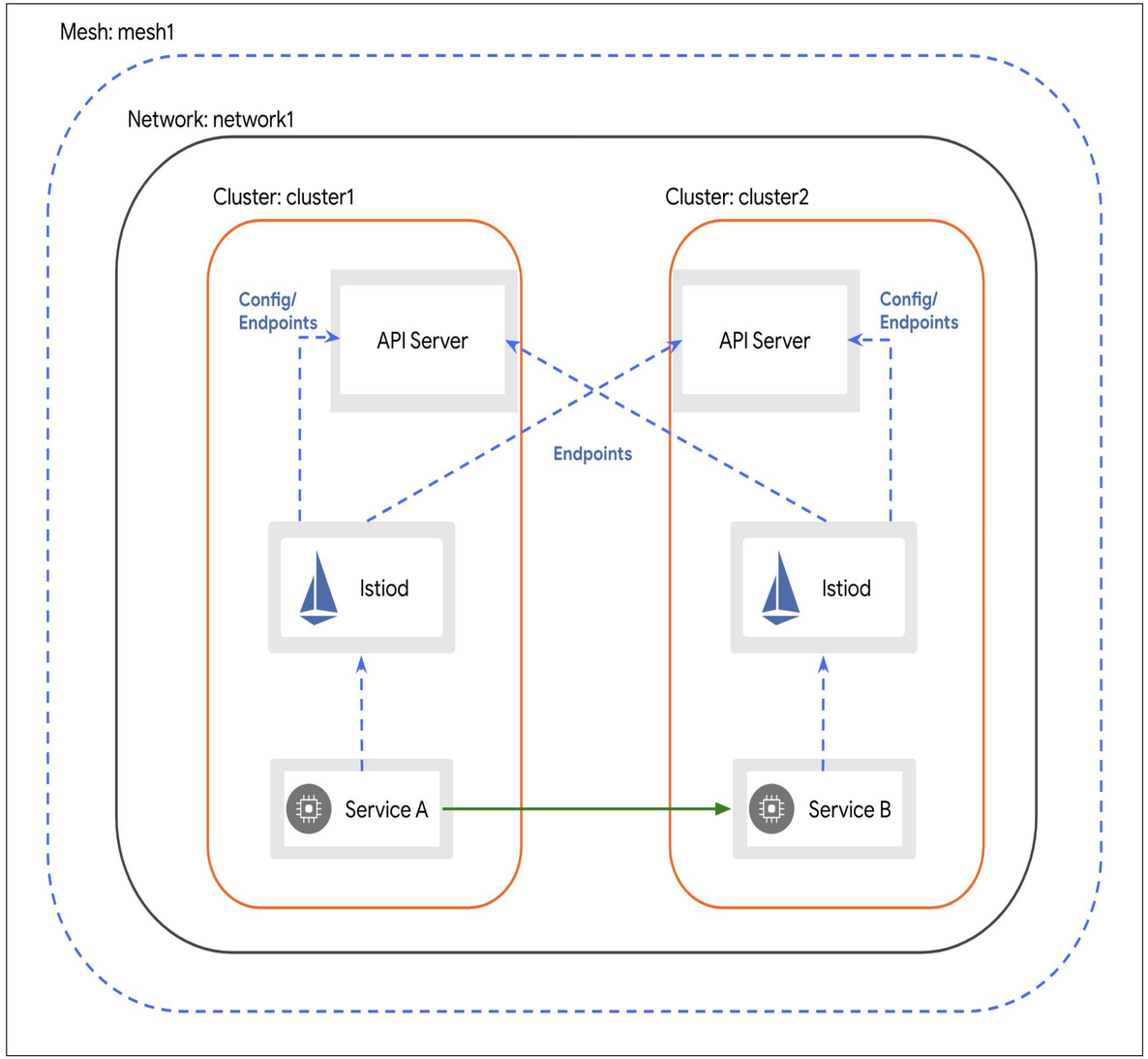
MOSN

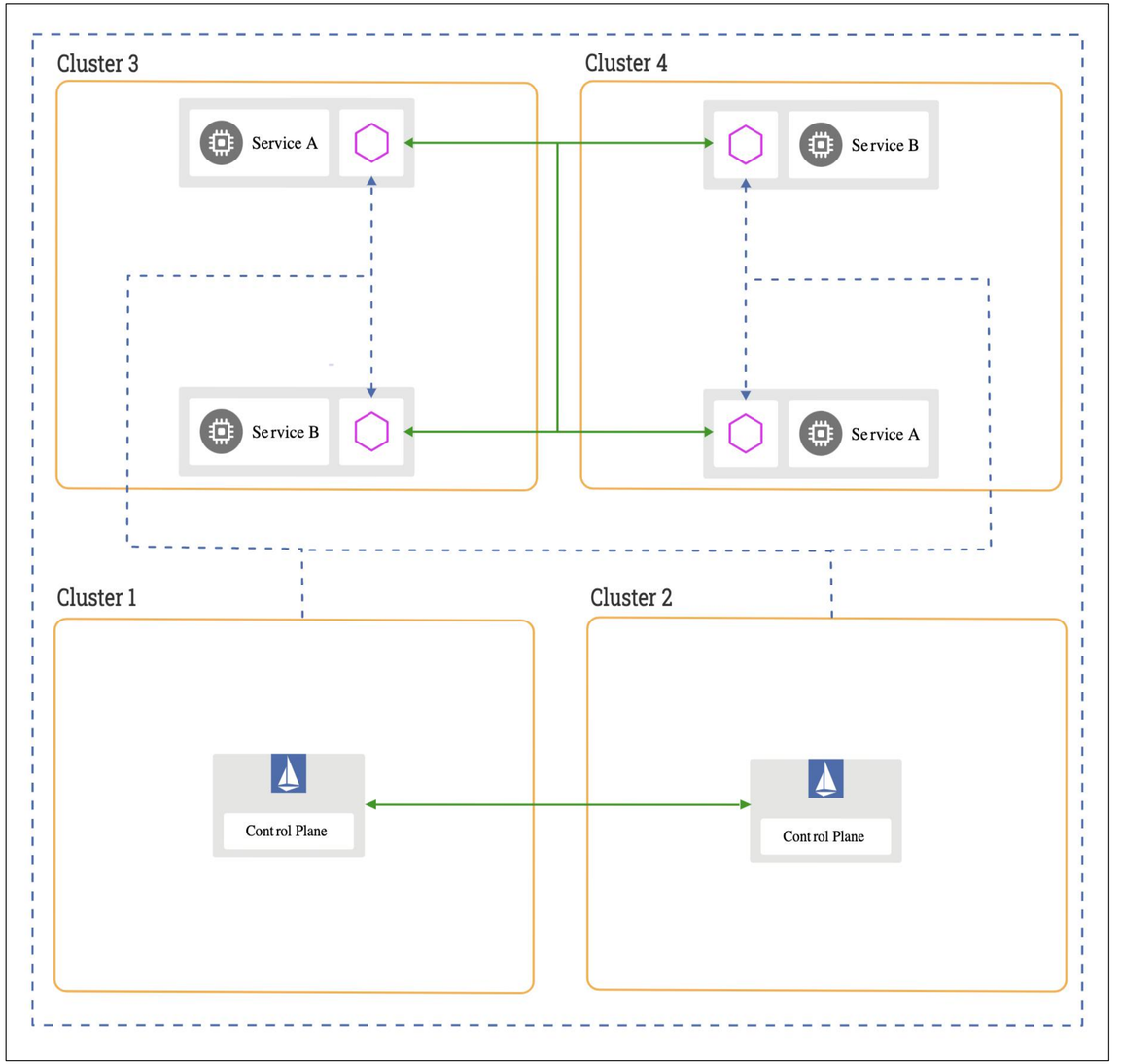
处理性能高
（C++）

envoy

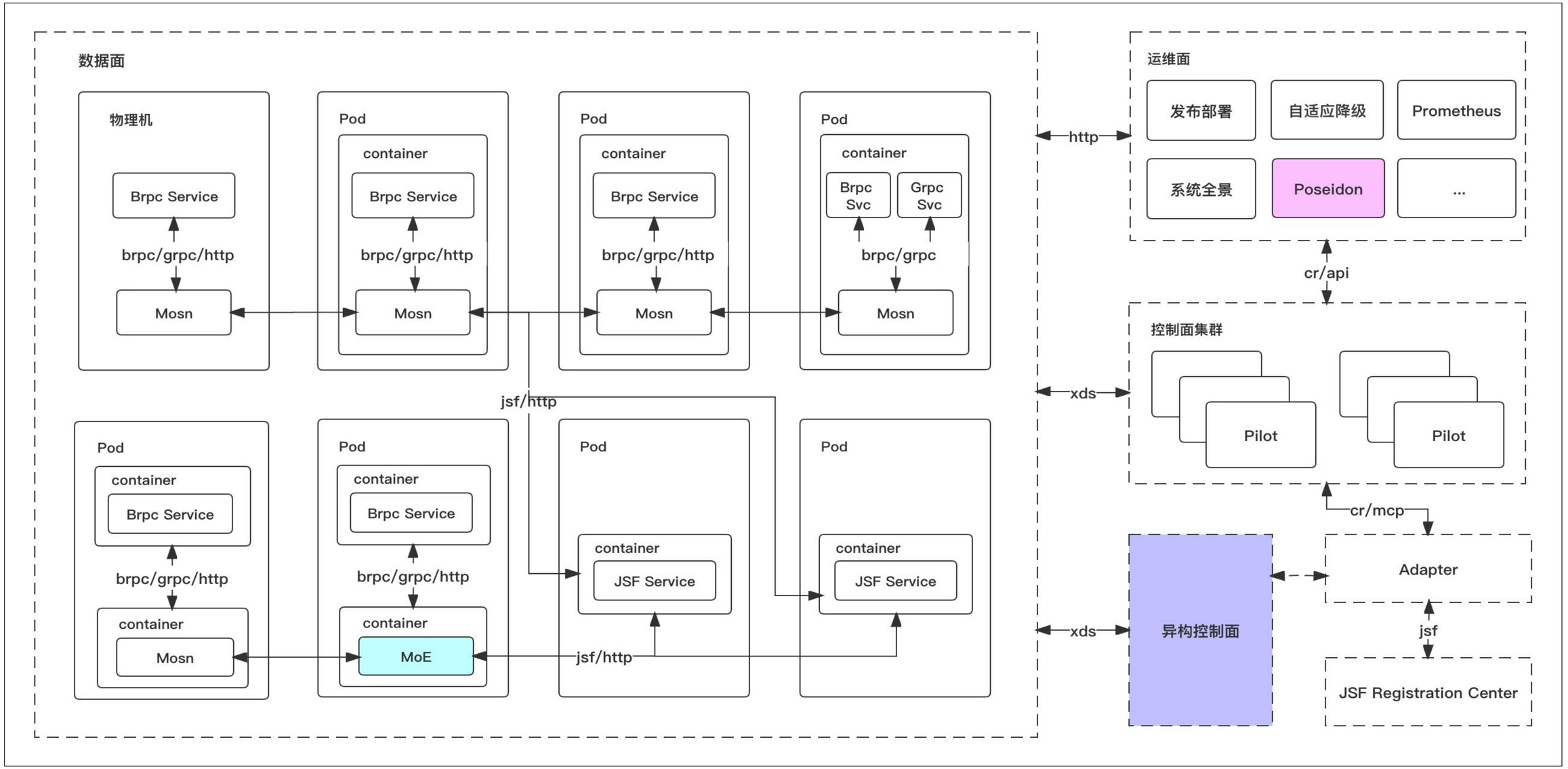


多集群多主控制面架构


Mesh: mesh1
Network: network1
Cluster: cluster1
Cluster: cluster2
Config/
Endpoints
API Server
API Server
Config/
Endpoints
Endpoints
Istiod
Istiod
Service A
Service B


Cluster 3
Cluster 4
Service A
Service B
Service B
Service A
Cluster 1
Cluster 2
Control Plane
Control Plane

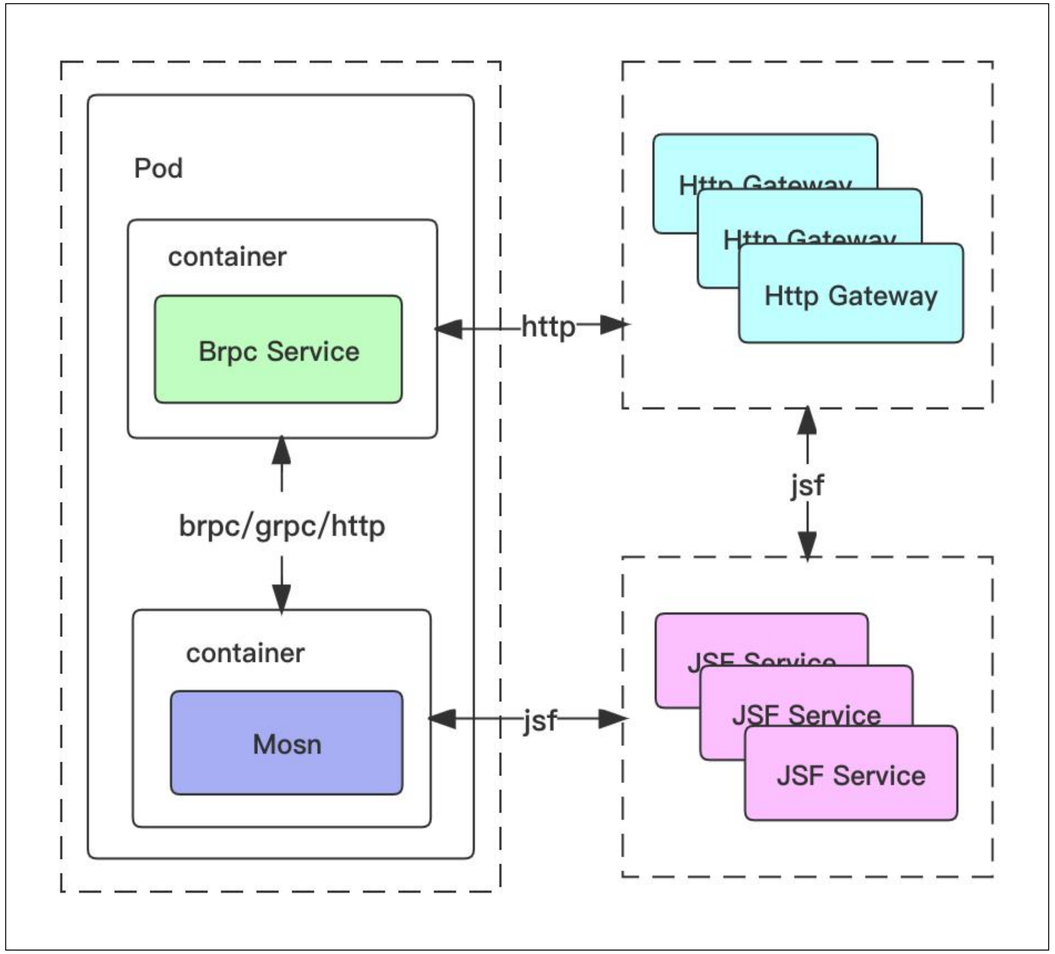
多形态数据面&多数据面 + 多控制面架构

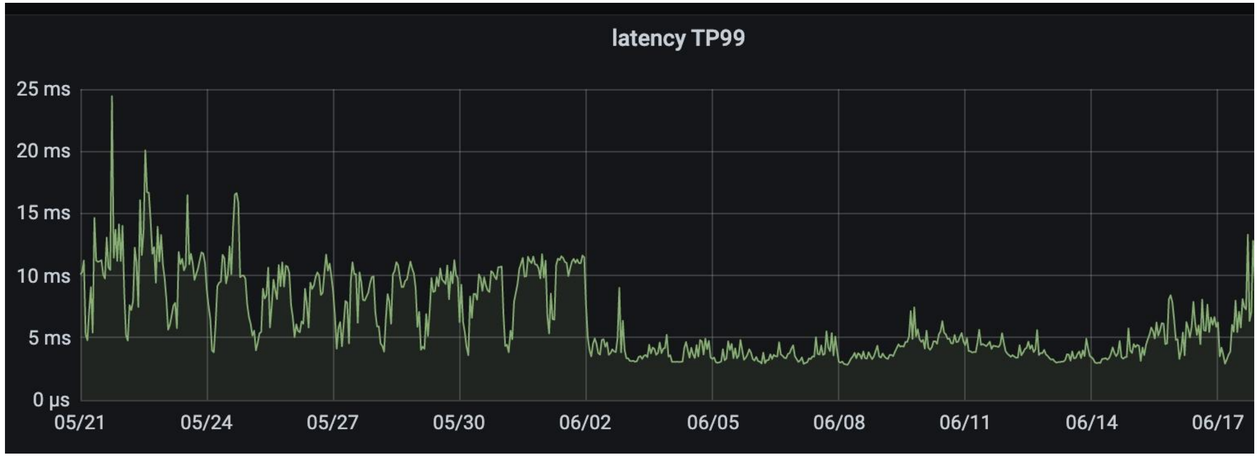


三、业务赋能探索&实践

跨语言、多协议去中心化网关

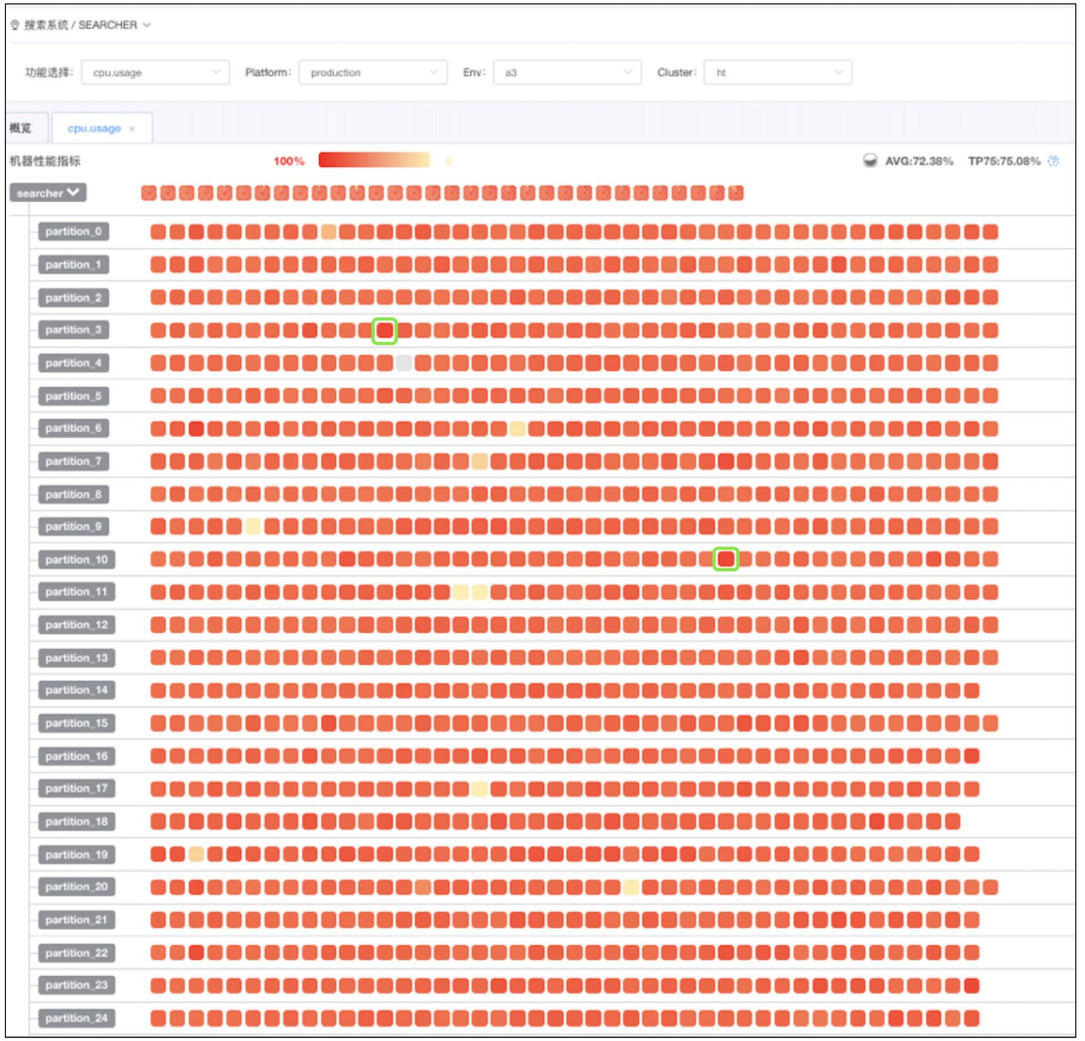
HTTP网关下沉到数据面=>私有协议RPC调用　TP99 降低 50%，抖动明显好转，可用率提高一个数量级





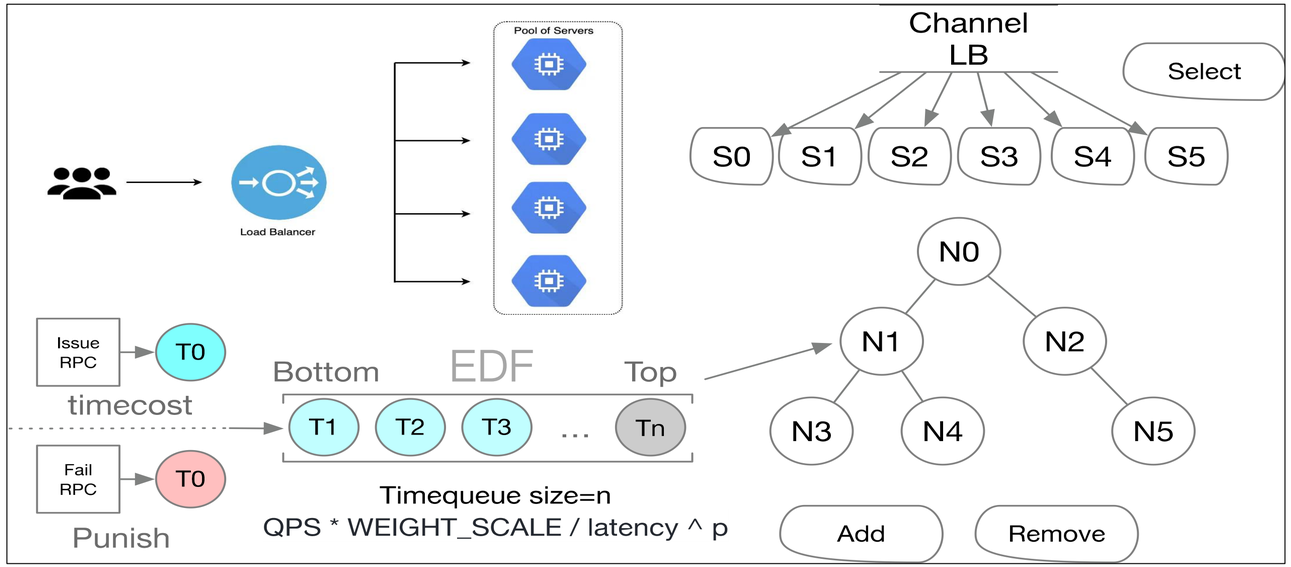
异构环境负载均衡——加权最小连接数

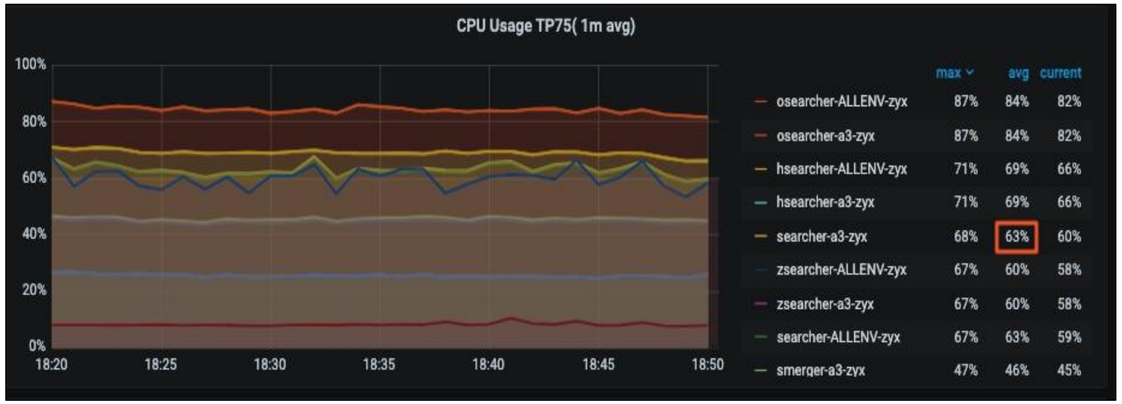
加权后不同规格机器可以相对均匀， TP99 降 5ms，但是个别算力或容器跟物理机差别大的，依然会不均匀

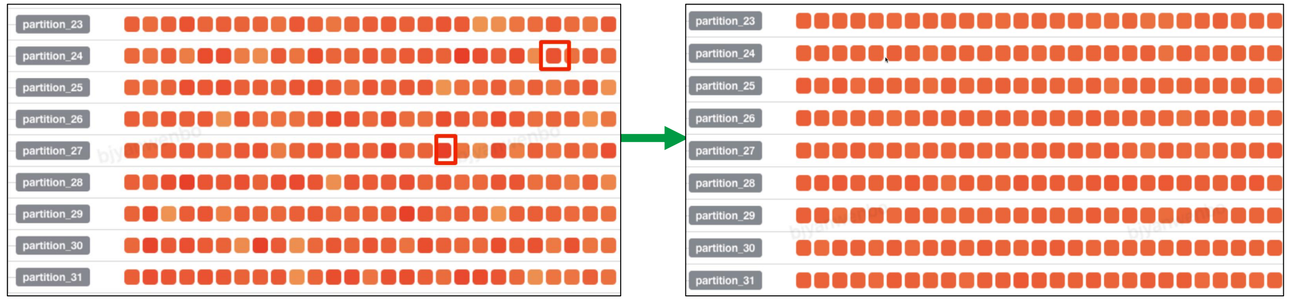


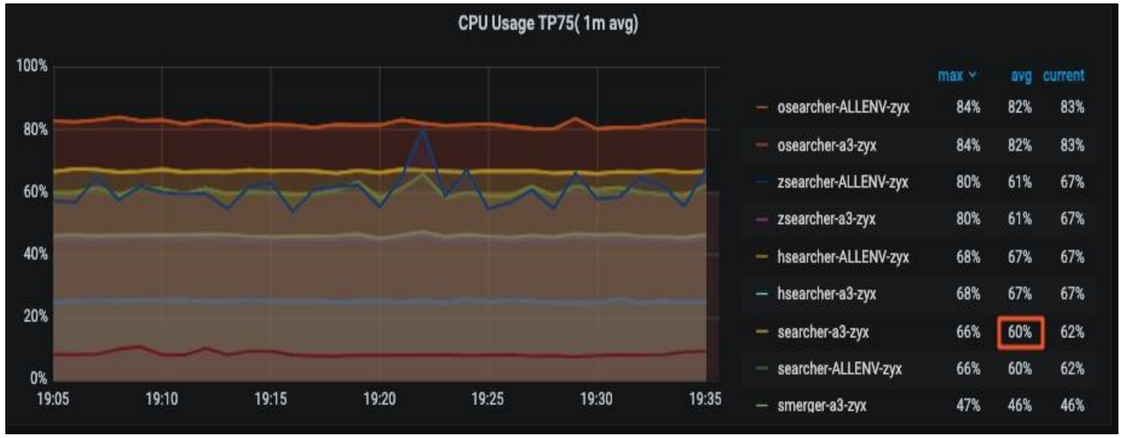
复合多策略负载均衡——加权&本地耗时感知&远端负载感知

可根据业务需要设置 CPU 保护水位，打开远端负载感知　常规流量 CPU TP75 63%=>60%，TP99 降 8 ms









基于Envoyfilter下发的混合跳步CPU/QPS自适应限流

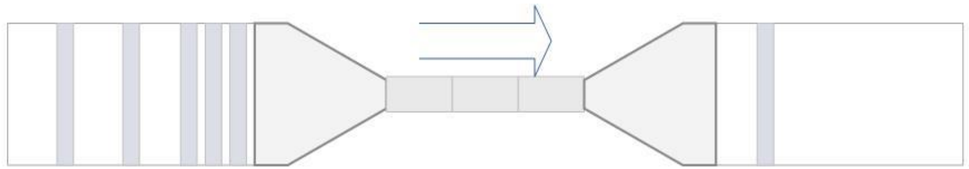
应对突发大流量与业务内嵌限流的关键指标对比

对比项	业务内嵌	MOSN	差值
生效速度	19s	12s	-36%
限流CPU	80%	78%	-2.5%
限流可用率	83%	88%	+6%

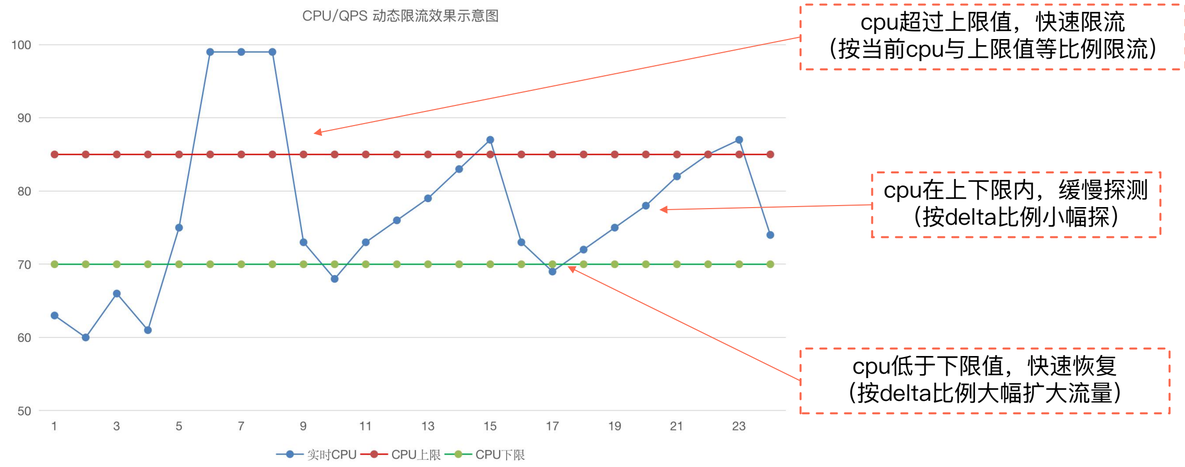
Little’s law : L = λ W

传输 BDP = BW * RTT 应用 TW = TPS * LATENCY

T ≈ QPS * Avg(RT)

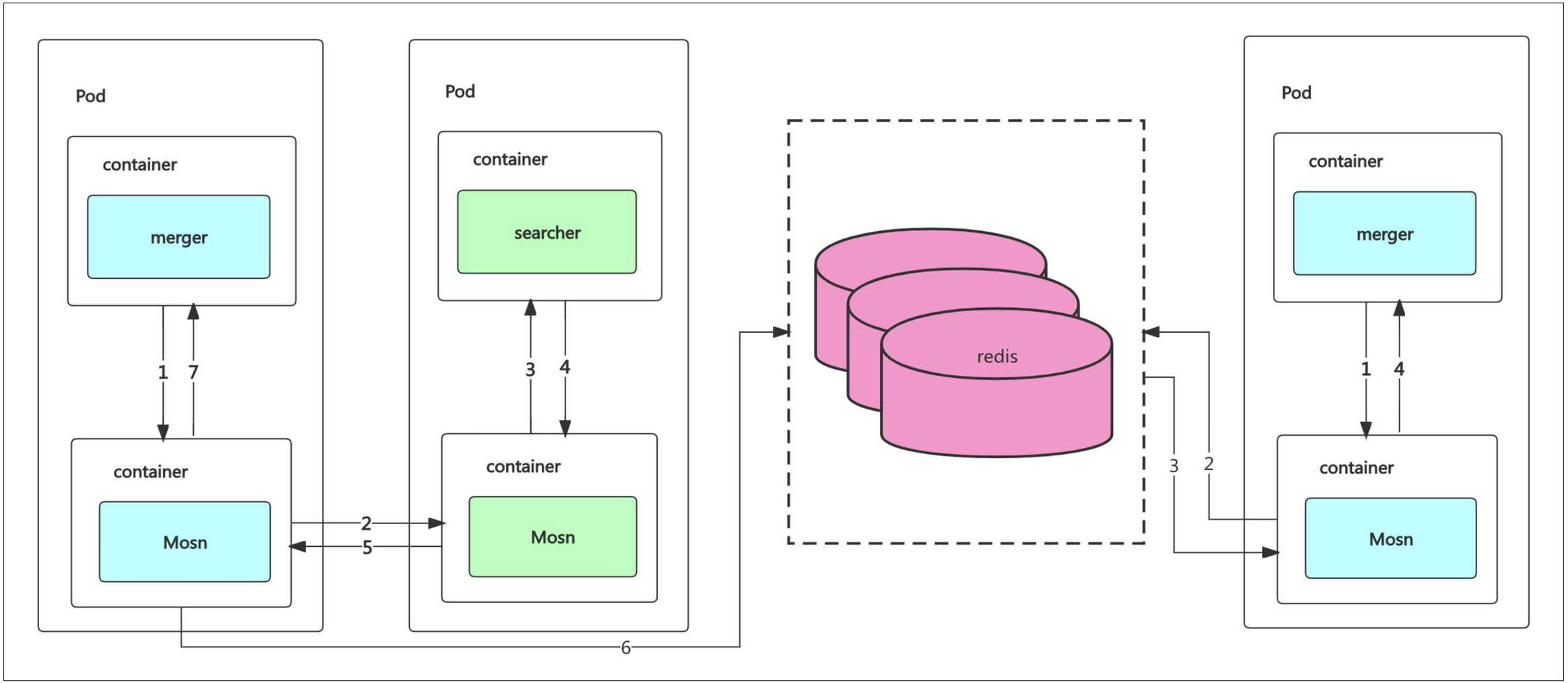


CPU/QPS 动态限流应对常规流量，可用率更高，TP99更低



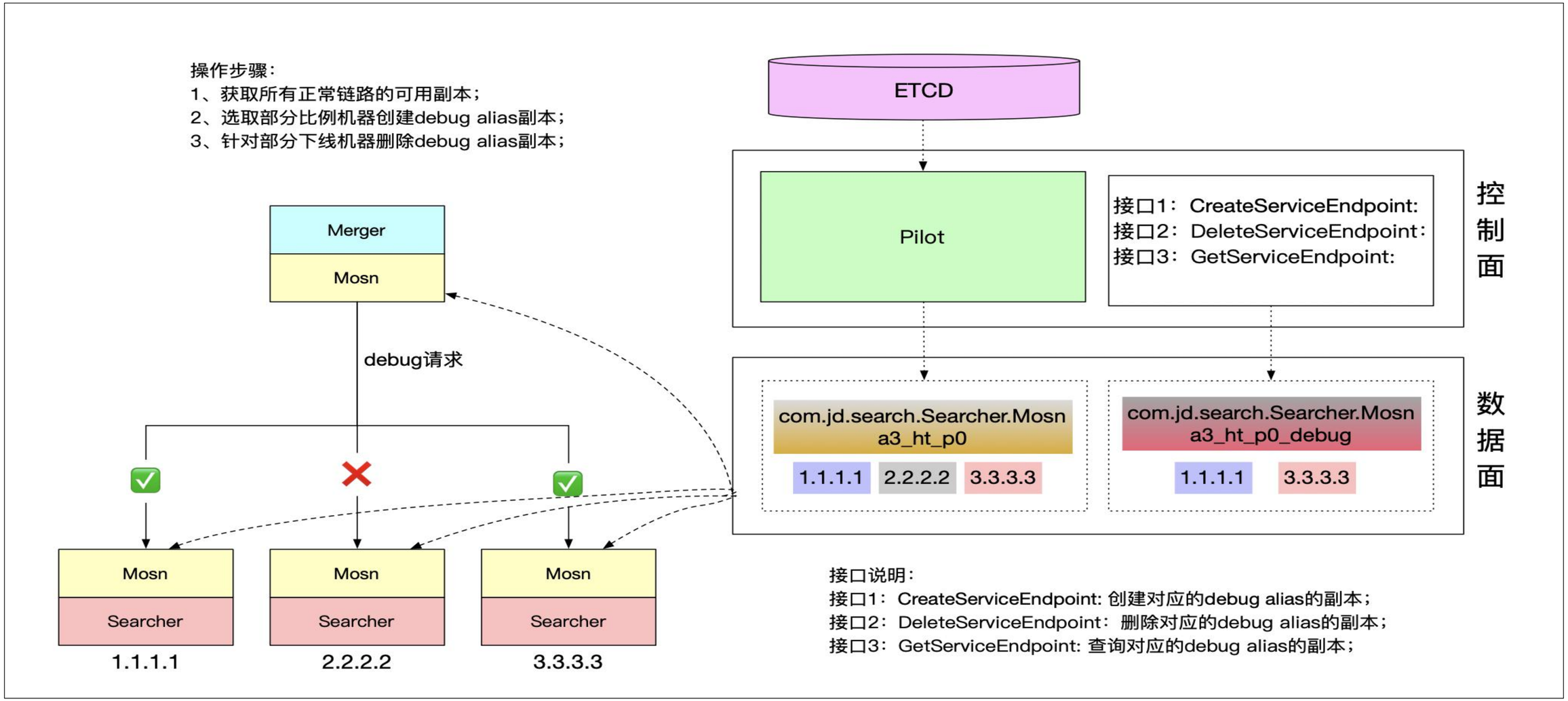
测试环境治理——单模块 Mock 测试

屏蔽个性化影响，提高压测效率；数据面一次修改，所有模块透明复用，一劳永逸；目前测试提效20%+



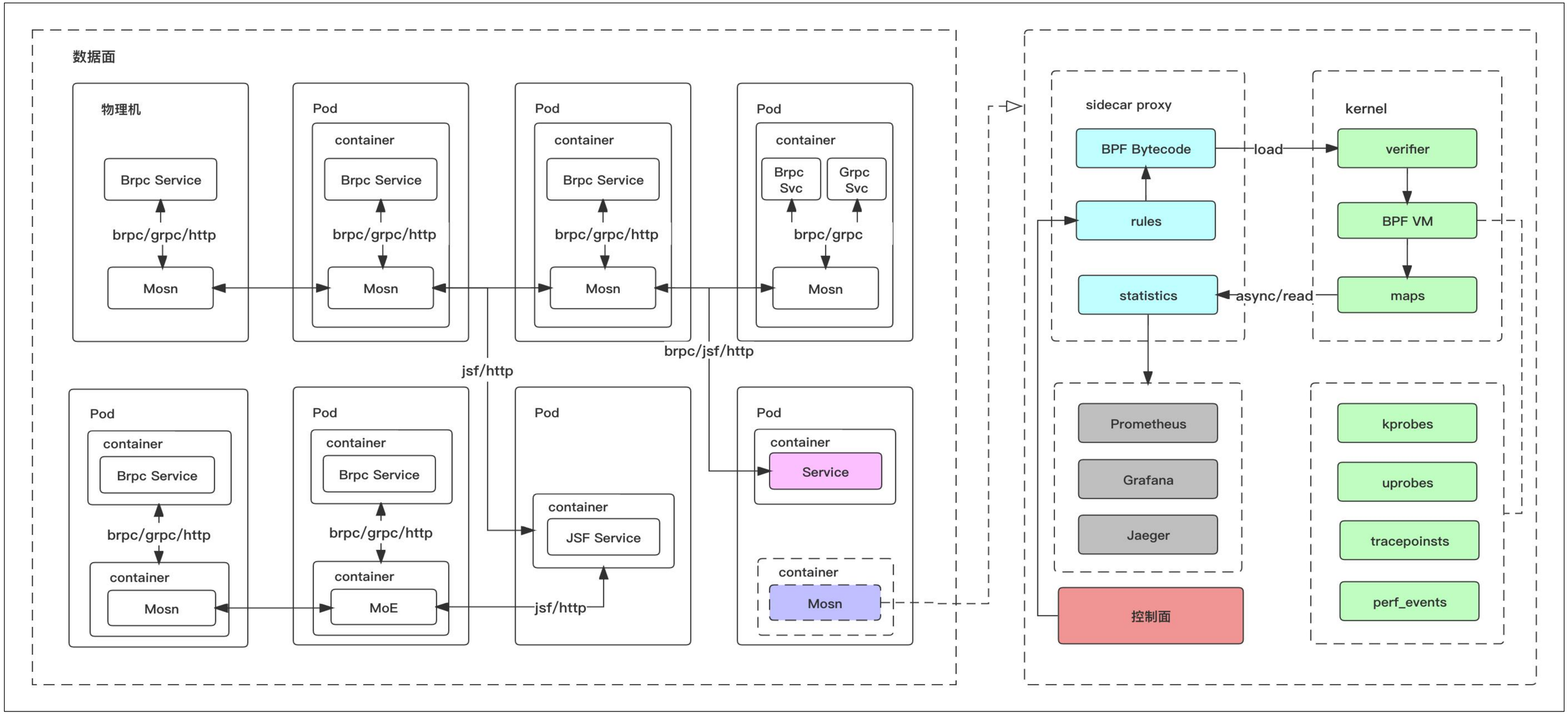
流量分组——以 Debug 流量为例

路由动态别名，实例按需分组，赋能异常流量测试，跨集群流量调度，动态扩分片，全流量实验



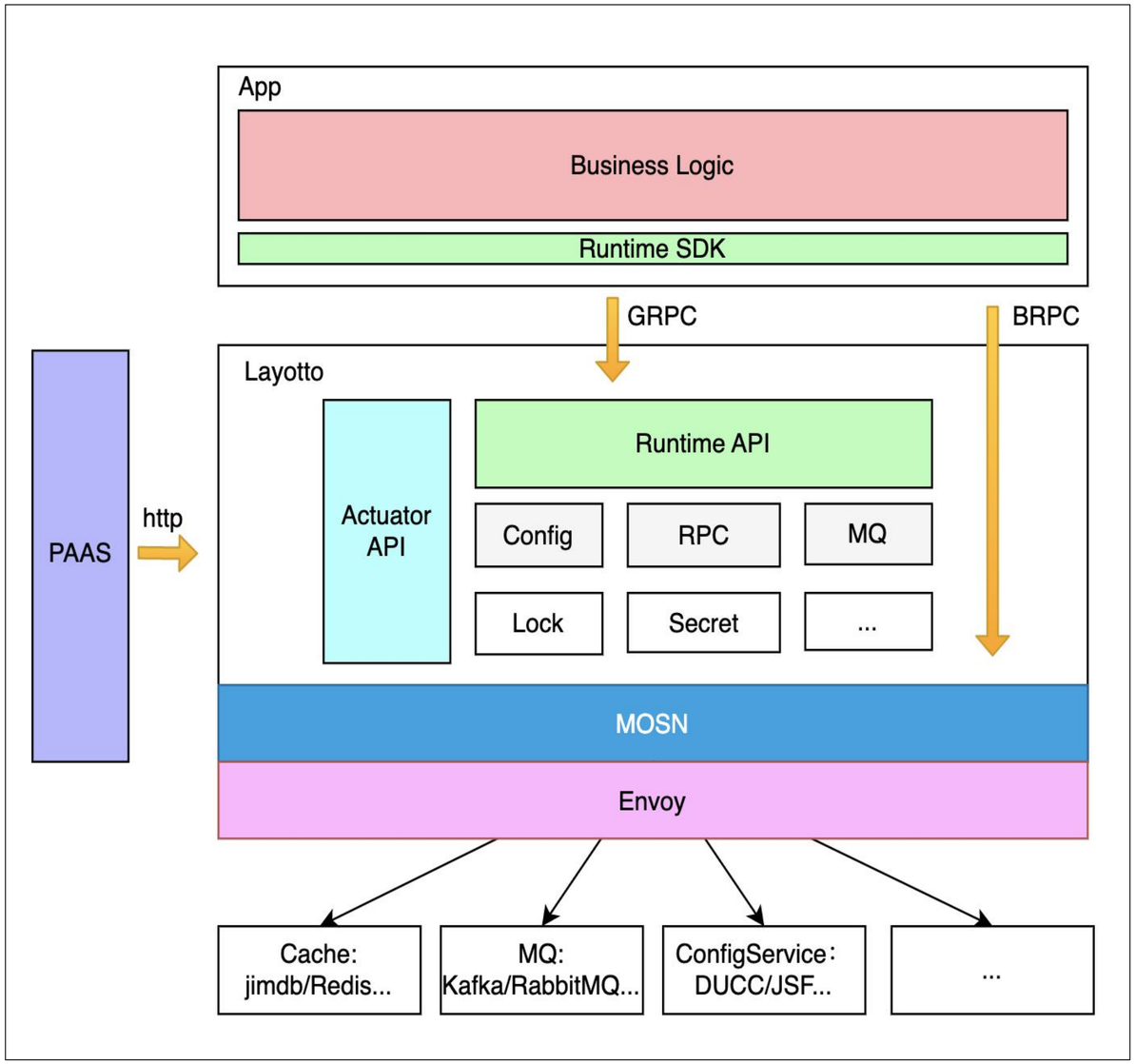
基于 eBPF 的旁路无侵入观测

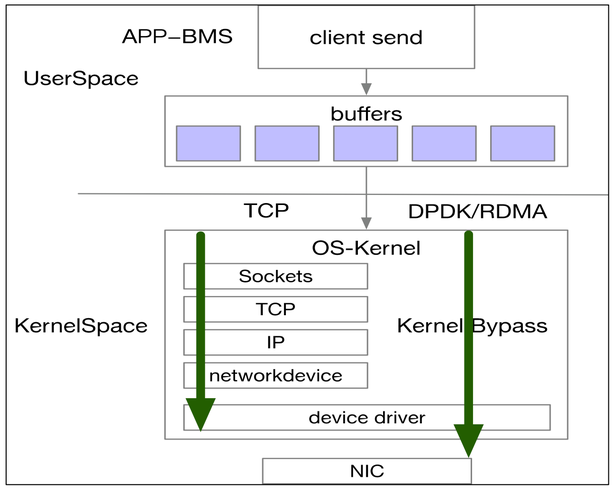
零侵入，跨语言，高扩展，低损耗 —— 有效快速解决跨语言异构系统、多模块的问题紧急排查和定位



四、技术规划与未来展望

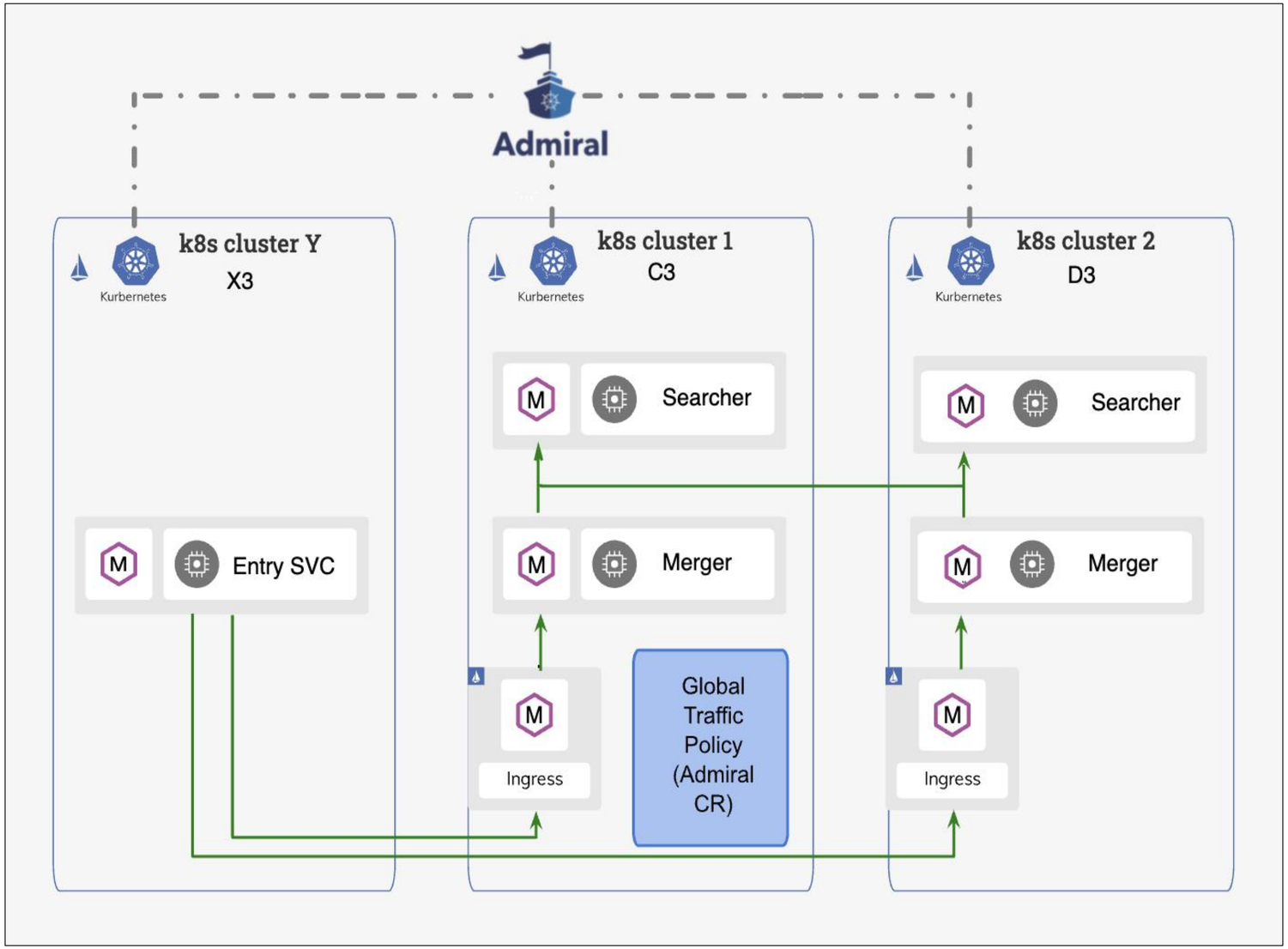
LiMoE = Layotto in MOSN on Envoy “能力 X 性能”



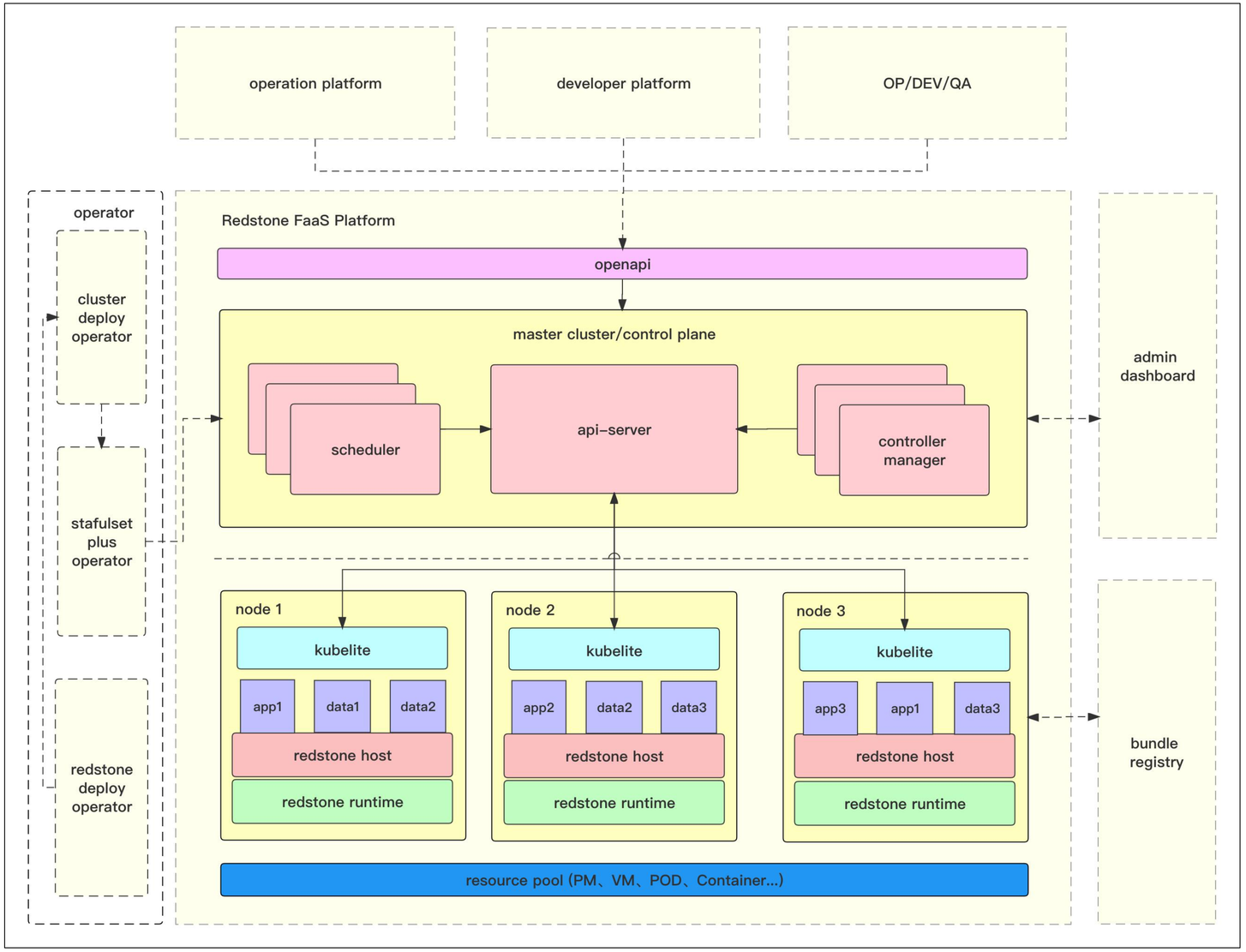


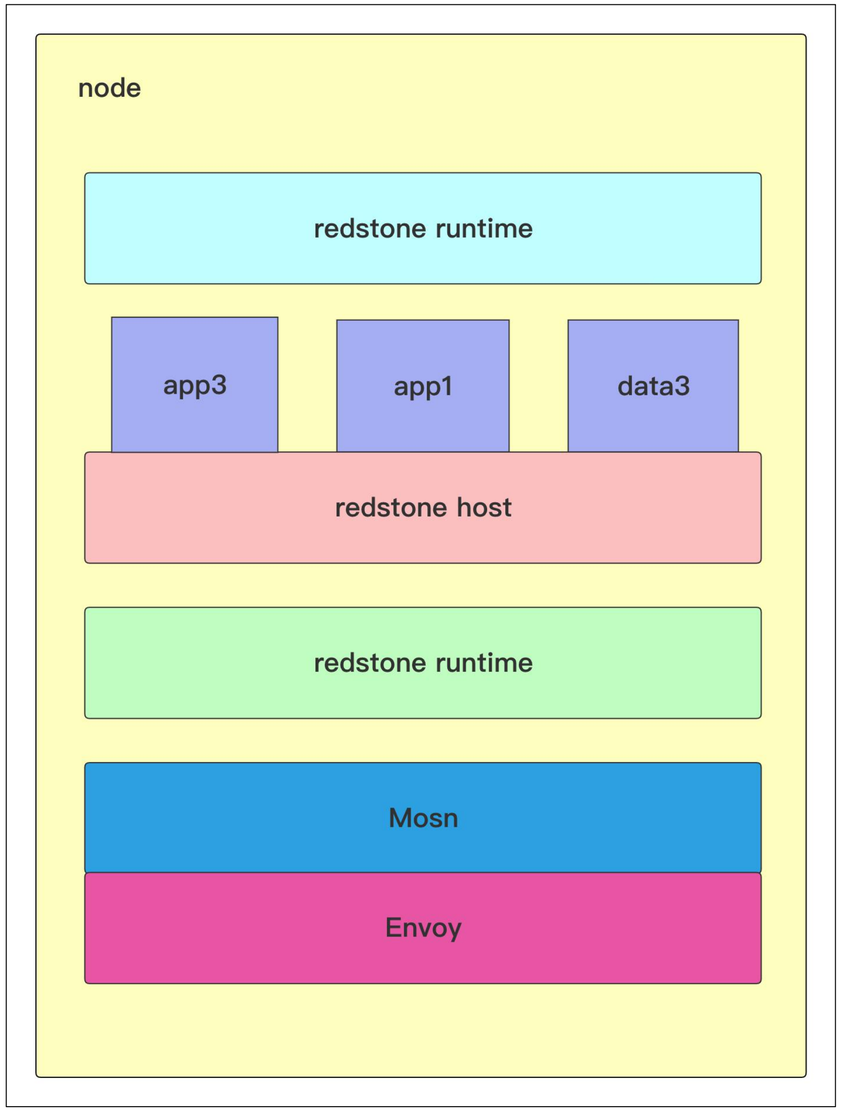
Attachment	RDMA	TCP/IP
1MB	Avg–Latency: 431, 90th-Latency: 437, 99th-Latency: 443, 99.9th-Latency: 446, Throughput: 1942.76MB/s, QPS: 1.98938k, Server CPU-utilization: 105%, Client CPU-utilization: 33% 2000qps	Avg–Latency: 632, 90th-Latency: 781, 99th-Latency: 857, 99.9th-Latency: 982, Throughput: 1459.37MB/s, QPS: 1.4944k, Server CPU-utilization: 83%, Client CPU-utilization: 31% 1500qps
3MB	Avg–Latency: 1180, 90th–Latency: 1188, 99th–Latency: 1203, 99.9th–Latency: 1208, Throughput: 2040.34MB/s, QPS: 0.696435k, Server CPU–utilization: 108%, Client CPU–utilization: 34% 700qps	Avg–Latency: 1898, 90th-Latency: 2131, 99th-Latency: 2357, 99.9th-Latency: 2484, Throughput: 1495.25MB/s, QPS: 0.510379k, Server CPU-utilization: 86%, Client CPU-utilization: 26% 510qps
5MB	Avg–Latency: 1918, 90th–Latency: 1930, 99th–Latency: 1945, 99.9th–Latency: 1952, Throughput: 2188.17MB/s, QPS: 0.448137k, Server CPU–utilization: 129%, Client CPU–utilization: 36% 450qps	Avg–Latency: 2569, 90th–Latency: 2656, 99th–Latency: 3939, 99.9th–Latency: 4227, Throughput: 1830.62MB/s, QPS: 0.37491k, Server CPU–utilization: 99%, Client CPU–utilization: 33% 375qps
10MB	Avg–Latency: 3774, 90th–Latency: 3781, 99th–Latency: 3793, 99.9th–Latency: 3808, Throughput: 2491.11MB/s, QPS: 0.25509k, Server CPU–utilization: 130%, Client CPU–utilization: 37% 250qps	Avg–Latency: 6127, 90th-Latency: 7398, 99th-Latency: 7662, 99.9th-Latency: 8391, Throughput: 1477.67MB/s, QPS: 0.151314k, Server CPU-utilization: 86%, Client CPU-utilization: 25% 150qps

Istio Ecosystem——基于 Admiral 智能自动化流量调控



Mesh Node 化架构赋能新一代 Serverless 平台





想一想，我该如何把这些技术应用在工作实践中？

THANKS

欢迎技术交流

InfoQ 极客传媒

ArchSummit

全 球 架 构 师 峰 会

数字化转型下的架构升级