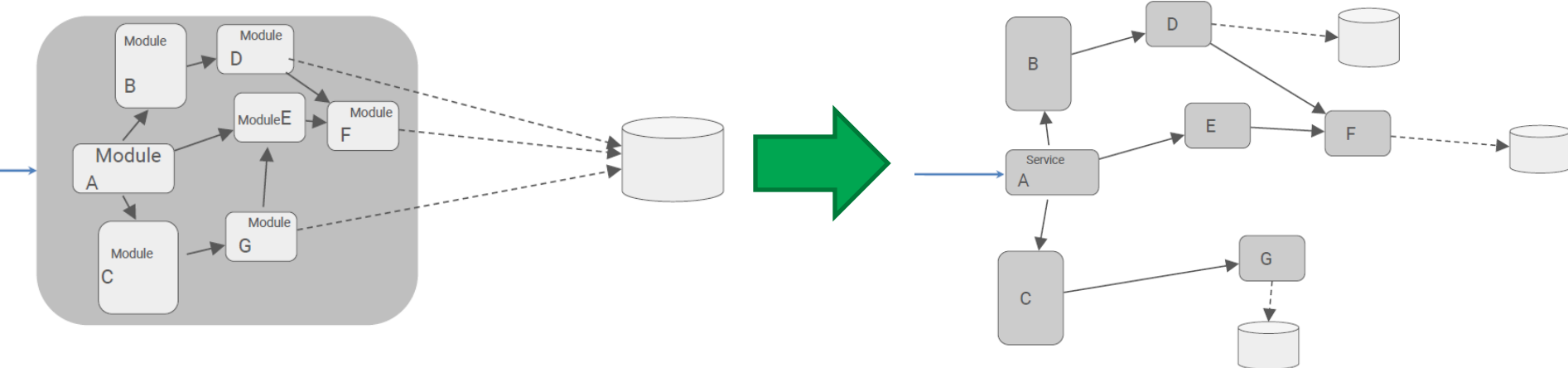


基于5G网络管理服务网络实践

# 服务网格技术介绍、实践 与发展趋势

赵化冰

# Monolith向Microservice 的演变



**An architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities.**

作为一种架构模式，微服务将复杂系统切分为数十乃至上百个小服务，每个服务负责实现一个独立的业务逻辑。

(对于程序的运行态而言，当应用以微服务架构的方式实现后，一个重要的变化是**进程内的方法调用变成了进程间的远程调用**。)

# 微服务架构带来的收益

- 更小的开发团队  
团队之间的交流更为顺畅，利于团队管理
- 语言、框架独立  
开发团队可针对业务特点和人员技能选择适合的技术路线
- 更小的代码库  
更容易理解服务的逻辑，新加入的成员可以很快上手
- 更短的开发迭代周期  
灵活应对市场变化，快速推进产品迭代
- 更小的内存和CPU资源占用  
方便各个服务的开发和测试
- 各服务可独立部署  
服务可根据负荷独立缩扩容  
隔离单个故障对整个系统的影响  
服务可独立升级，互不影响



# Microservice 架构带来的挑战

微服务架构引入了**分布式系统**带来的一系列复杂问题，包括：

## 问题：

- 客户端如何找到服务提供者
- 如何保证远程调用的可靠性
- 如何保证服务调用的安全性
- 如何保证系统的可见性
- 如何进行端到端流程调试
- 如何保证系统的健壮性
- 如何对系统进行容错性测试

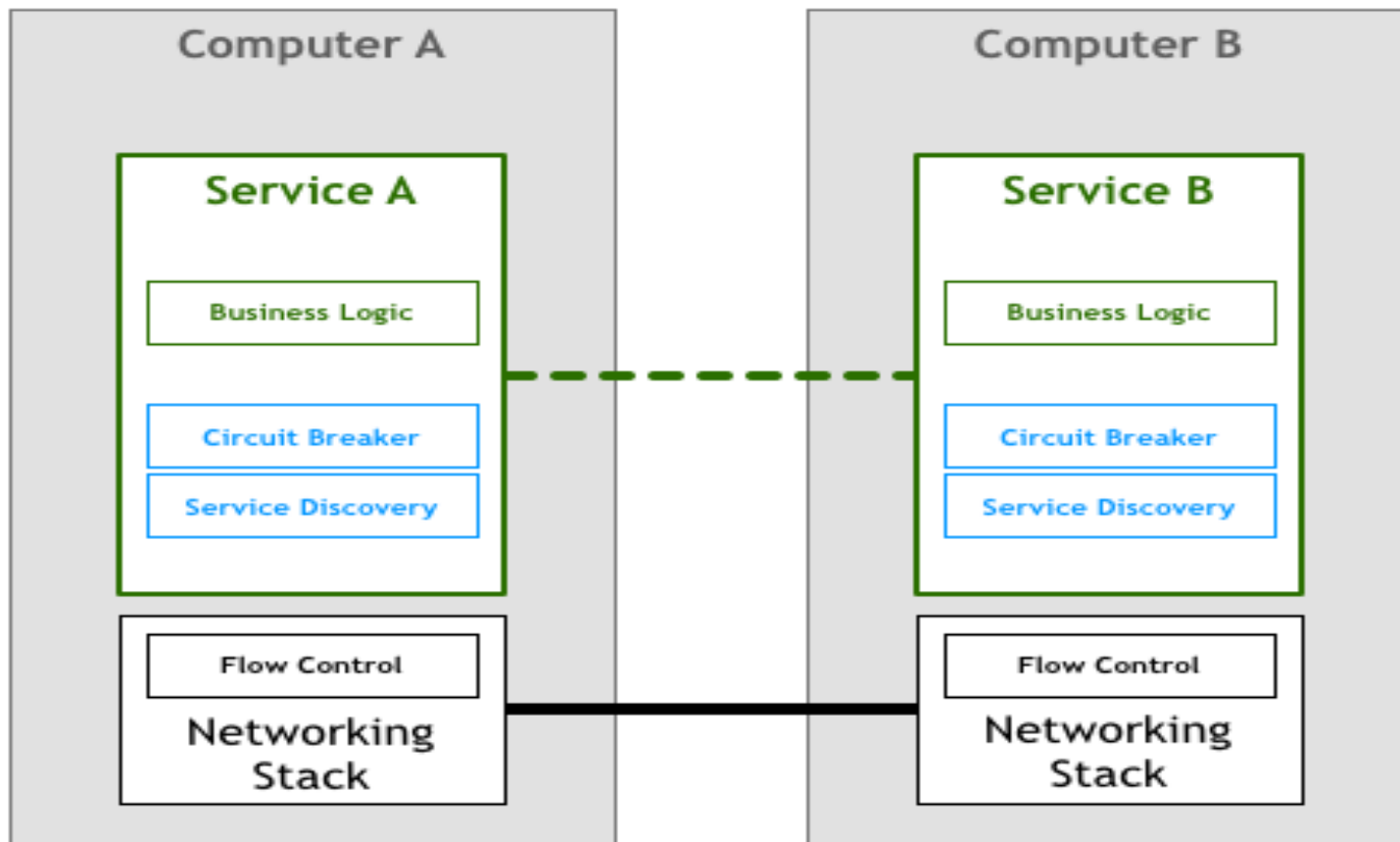
## 方案：

- 服务注册和发现
- 超时，重试，负载均衡
- 服务认证和鉴权
- 性能指标收集及分析/日志收集
- 分布式调用追踪
- 熔断，限流，故障恢复
- 故障模拟/Chaos测试

微服务架构在带来收益的同时提高了系统的复杂度，并加大了应用运维的难度。

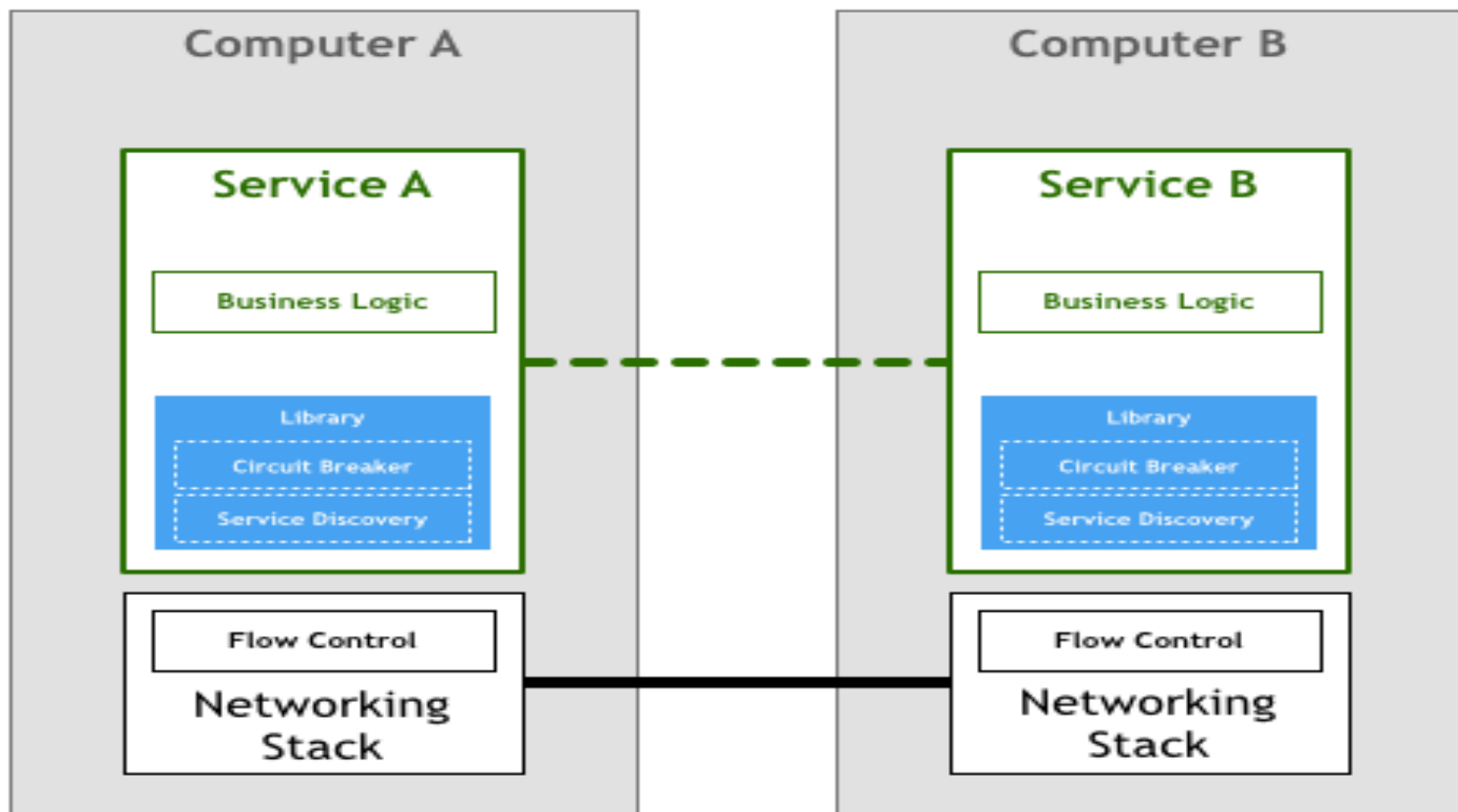
# 解决方案：原始时代

由不同微服务的开发人员各自处理服务之间的通讯，包括服务发现，重试，超时，容错，安全等逻辑。



# 解决方案：类库模式

针对不同的语言提取出类库，来解决微服务的通讯的一些共性问题。



# 类库带来的问题

微服务带来的一个巨大优势，就是开发团队可以根据业务特点和人员技能**灵活采用不同的技术栈**来实现一个微服务。但是，当我们将服务通讯和治理相关代码封装到类库和框架时，有个小问题冒出来了☺

## 主流编程语言

- Java
  - Scala
  - Groovy
  - Kotlin
- C
- C++
- C#
- Python
- PHP
- Ruby

## 新兴编程语言

- Golang
- Node.js
- Rust
- R
- Lua/OpenResty

- 需要针对不同的程序语言开发不同的代码库，反过来会影响微服务应用开发语言 和框架的选择，限制技术选择的灵活性。

- 随着时间的变化，代码库会存在不同的版本，不同版本代码库的兼容性和大量运行 环境中微服务的升级将成为一个难题。

- 类库的机制比较复杂，对开发人员而言有较高的学习成本，导致其不能将其全部精力聚焦于业务逻辑。

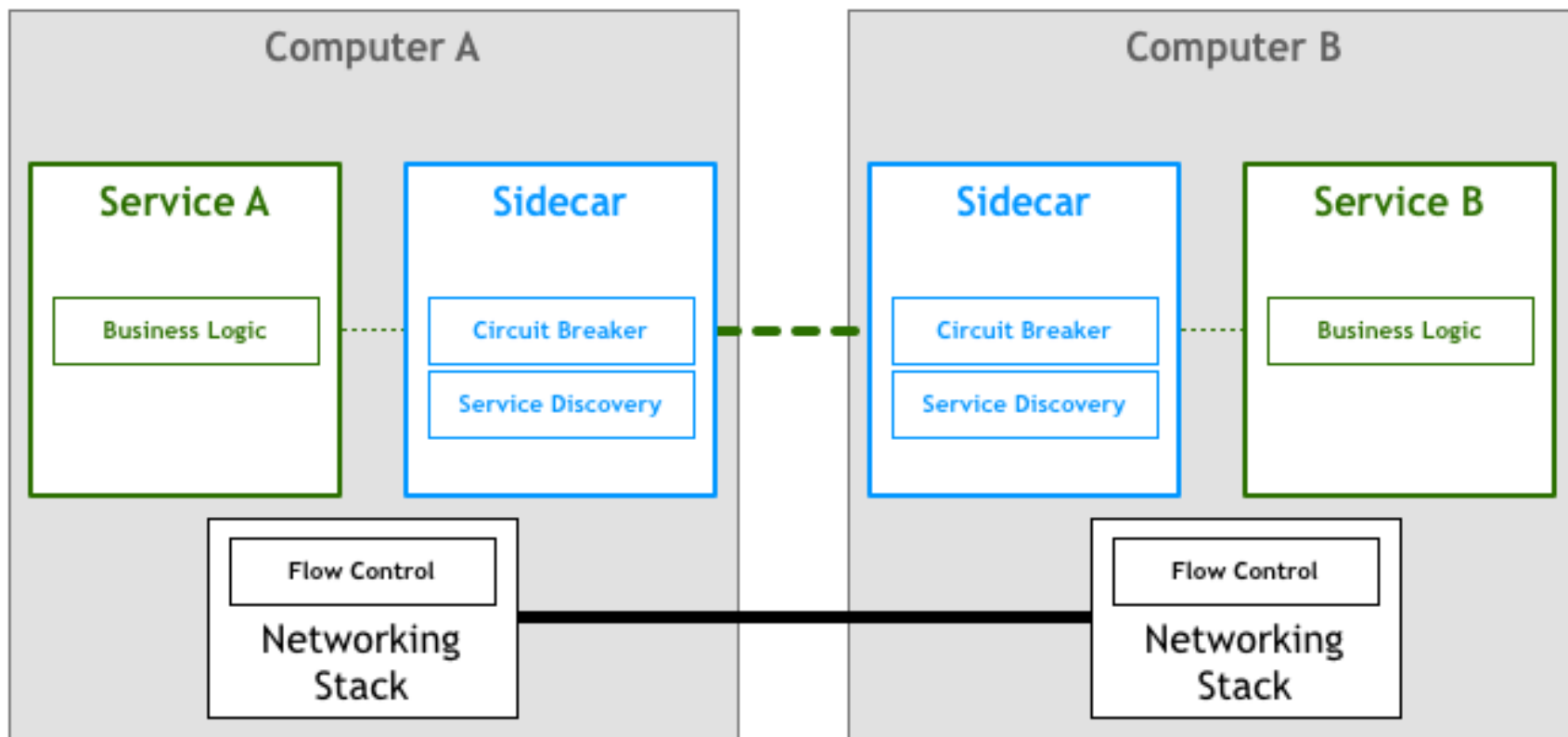
我们需要开发/  
维护多少种类  
库？



说好的多语言呢？

# 解决方案：边车模式

应用程序在进行网络通信时并不需要关注TCP/IP协议栈的实现，微服务需要关注服务间的通信细节吗？

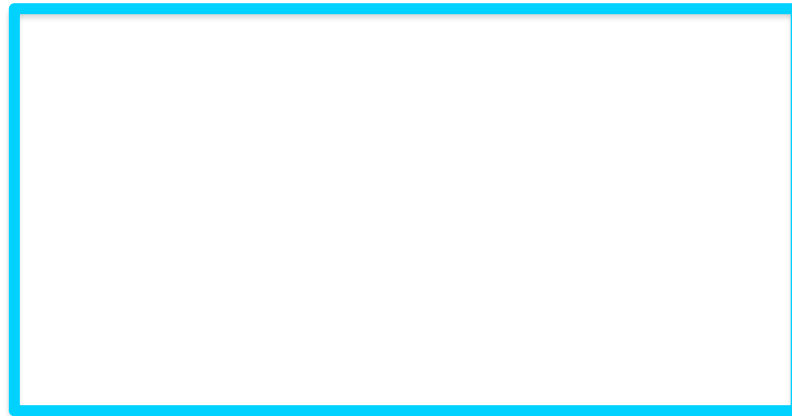


边车模式：将为微服务提供通信服务的这部分逻辑从应用程序进程中抽取出来，作为一个单独的进程进行部署，并将其作为服务间的通信代理。

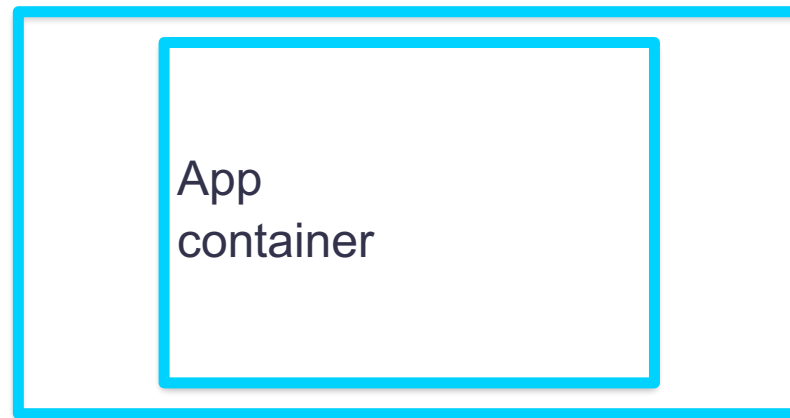


让我们来回顾一下微服务架构的演进过程

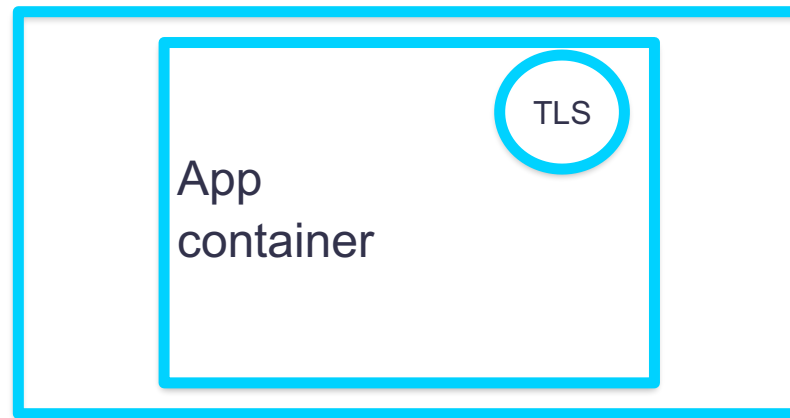
# Before



# Before



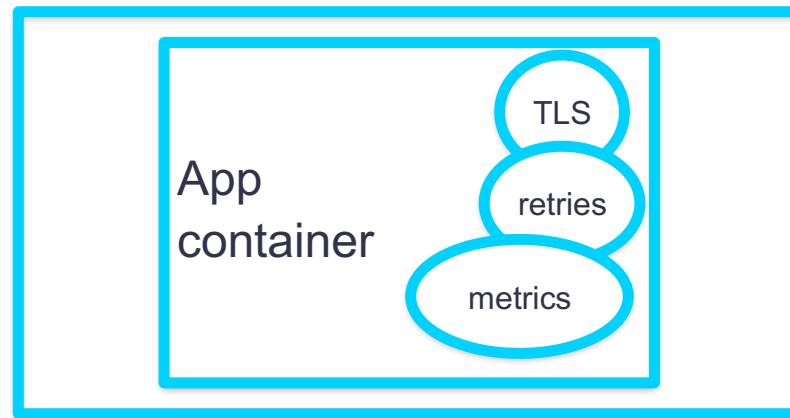
# Before



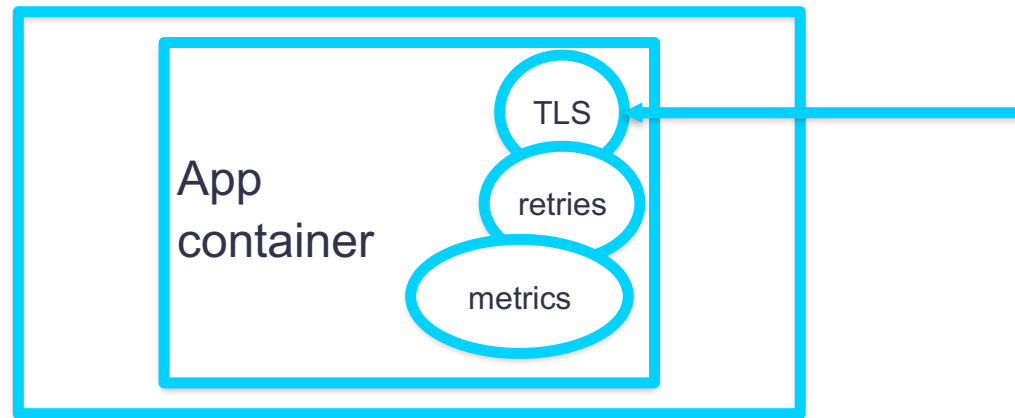
# Before



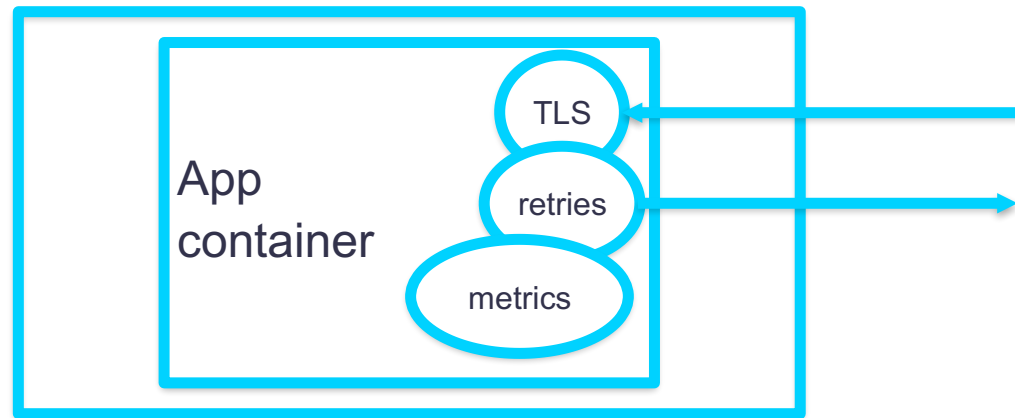
# Before



# Before

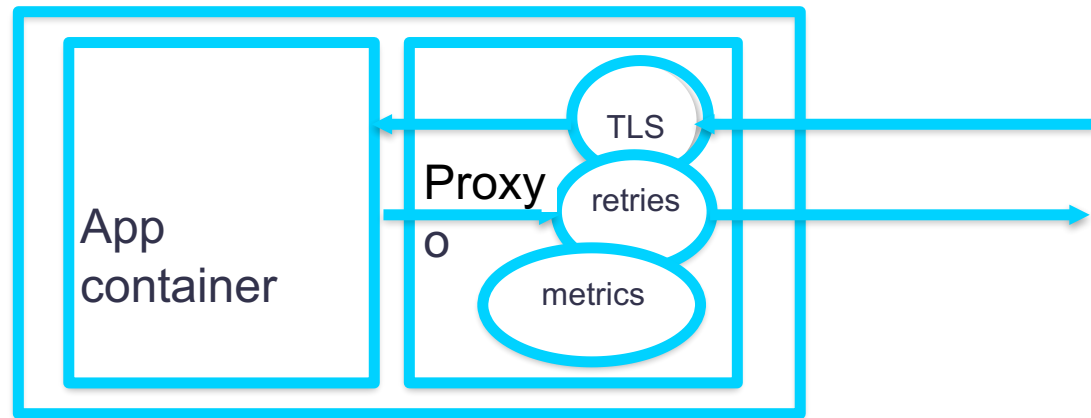


# Before

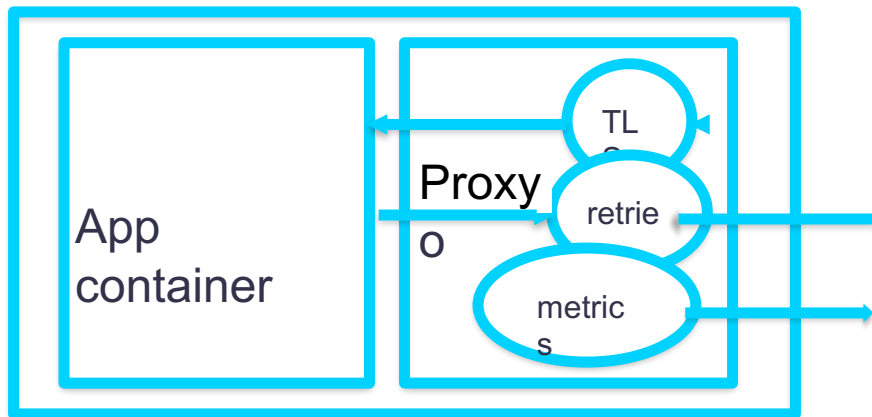




# After

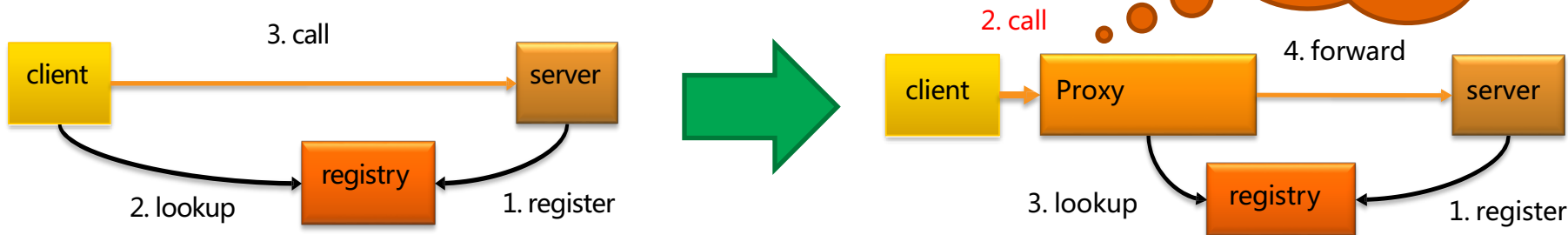


微服务通讯代理的这种部署方式被形象地称为“**Sidecar**”，即三轮摩托车的“挎斗”



Sidecar模式可用于插入其他横切面逻辑：  
安全、策略、  
Telemetry、分布式调用跟踪、缓存等等

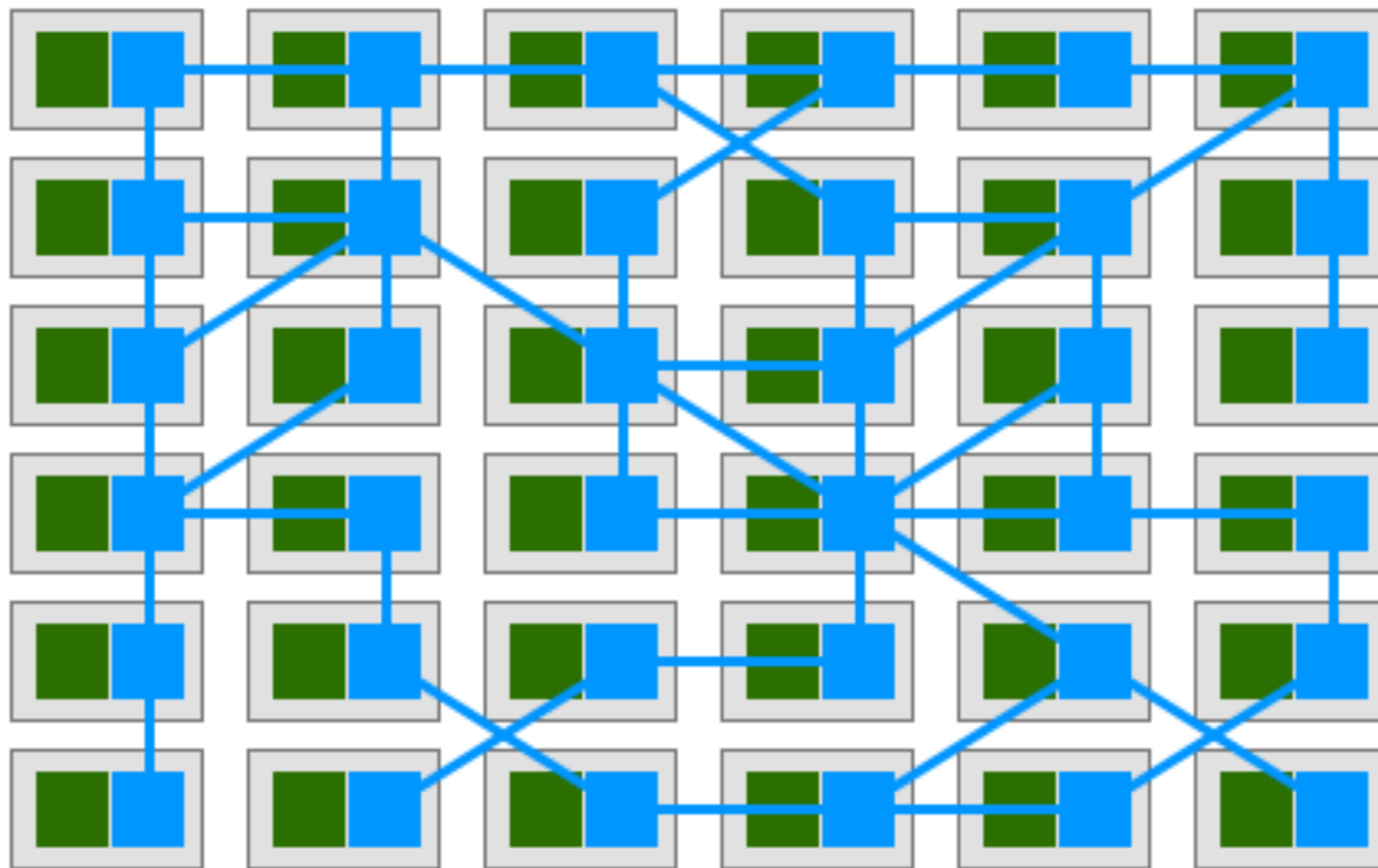
服务间的通信层处理被迁移到以“挎斗”方式部



Proxy对应用**无侵入**，应用对Proxy**不感知**

服务开发者只需要关注产品的商用价值重点：业务逻辑

在集群中部署的多个Proxy形成支撑服务间通信的一个“网格”



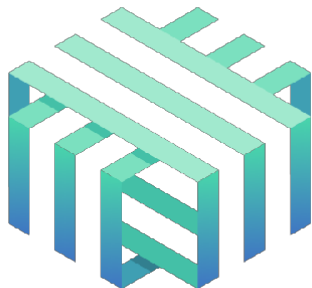
# 服务网格(Service Mesh)的定义

**Willian Morgan**（**Buoyant**的**CEO**）给出的定义：

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.

服务网格是一个**基础设施层**，用于处理服务间通信。云原生应用有着复杂的服务拓扑，服务网格负责在这些拓扑中**实现请求的可靠传递**。在实践中，服务网格通常实现为一组**轻量级网络代理**，它们与应用程序一起部署，但**对应用程序透明**。

# Service Mesh开源项目



## • Linkerd

- 来自buoyant, Scala语言
- Service Mesh名词的创造者
- 2016年1月15日, 0.0.7发布
- 2017年1月23日, 加入CNCF
- 2017年4月25日, 1.0版本发布
- 2018年, 2.0发布, 采用全新架构, 引入控制面(控制面:Go 数据面:Rust)



## • Istio

- 来自Google, IBM和Lyft, Go语言
- Service Mesh集大成者
- 2017年5月发布0.1版本
- 2018年3月发布0.7版本
- 2018年7月发布1.0版本
- .....

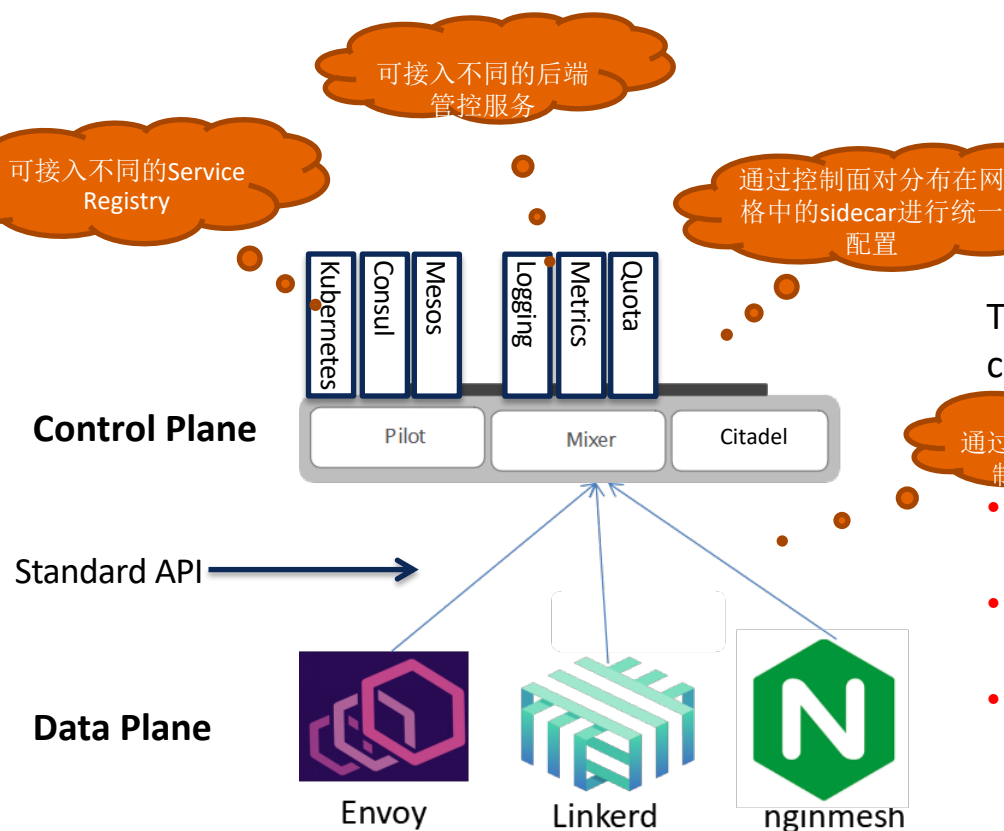
Linkerd在1.x版本宣布支持作为数据面和Istio控制面进行集成。在2.0版本中借鉴了Istio的架构, 也引入了控制面。

# 以Kubernetes为核心的微服务生态系统



Google已经通过CNCF基金会和Kubernetes生态圈掌握了PaaS容器部署和编排的事实标准。通过Istio，Google补充了Kubernetes生态圈的重要一环，是Google从容器编排向微服务领域的一个里程碑式的扩张。

# 为什么选择Istio



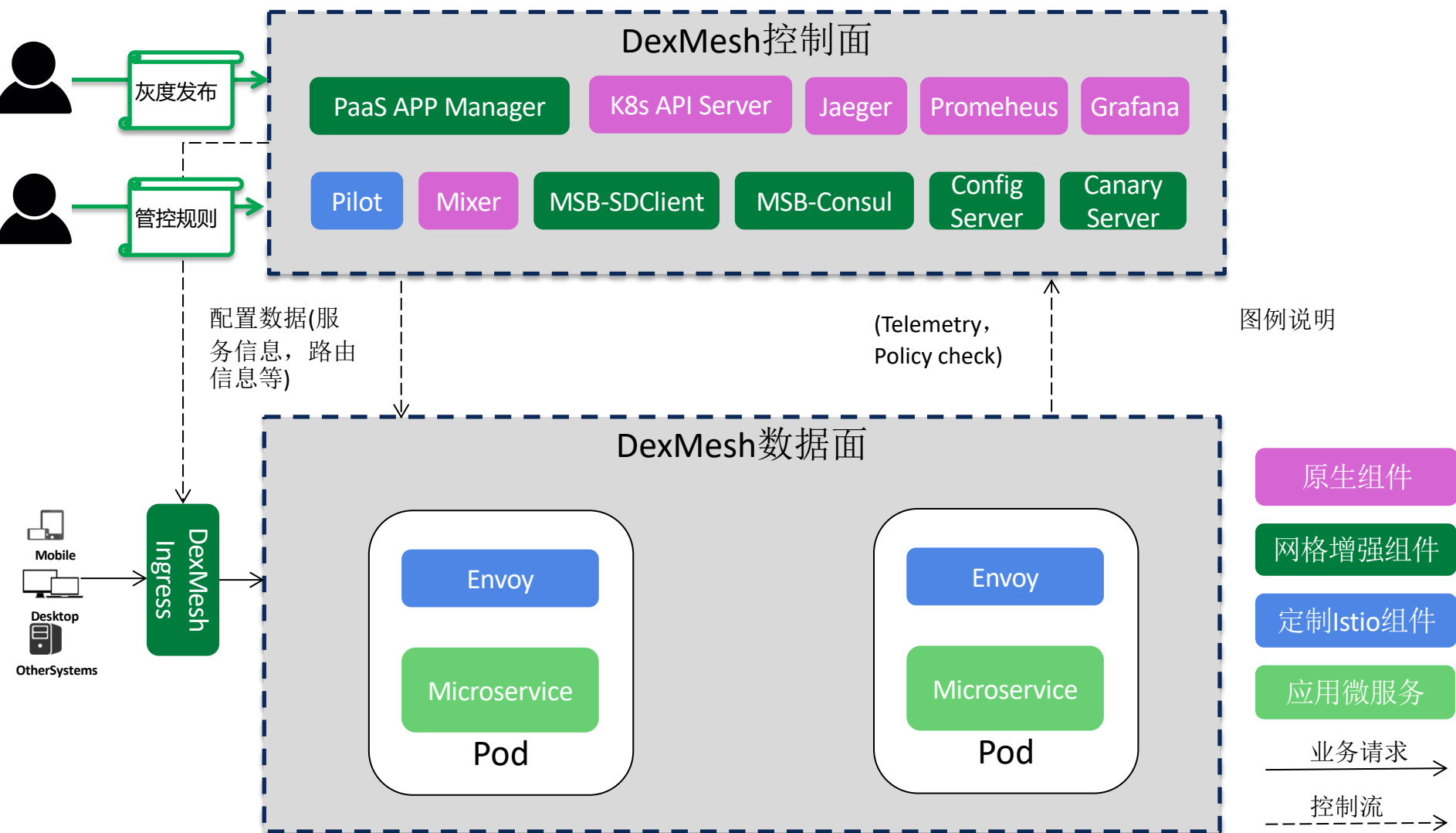
The main advantage of Istio is introducing a centralized **Control Plane** to manage the distributed mesh. The Control plane is a set of management utilities including:

- **Pilot**: routing tables, service discovery, and load balancing pools
- **Mixer**: Policy enforcement and telemetry collection
- **Citadel**: TLS mutual service authentication and fine-grained RBAC

The Istio control plane is **highly extendable by design**.

- Multiple adapters can be plugged into Pilot to populate the services: Kubernetes, Consul, Mesos...
- Different backends can be connected to Mixer without modification at the application side: Prometheus, Heapster, AWS CloudWatch...
- Standard API between Pilot and data plane for service discovery, LB pool and routing tables which decouples the sidecar implementation and Pilot: Envoy, Linkerd, Nginmesh are all support Istio now and can work along with Istio as sidecar

# 总体架构-高层视图

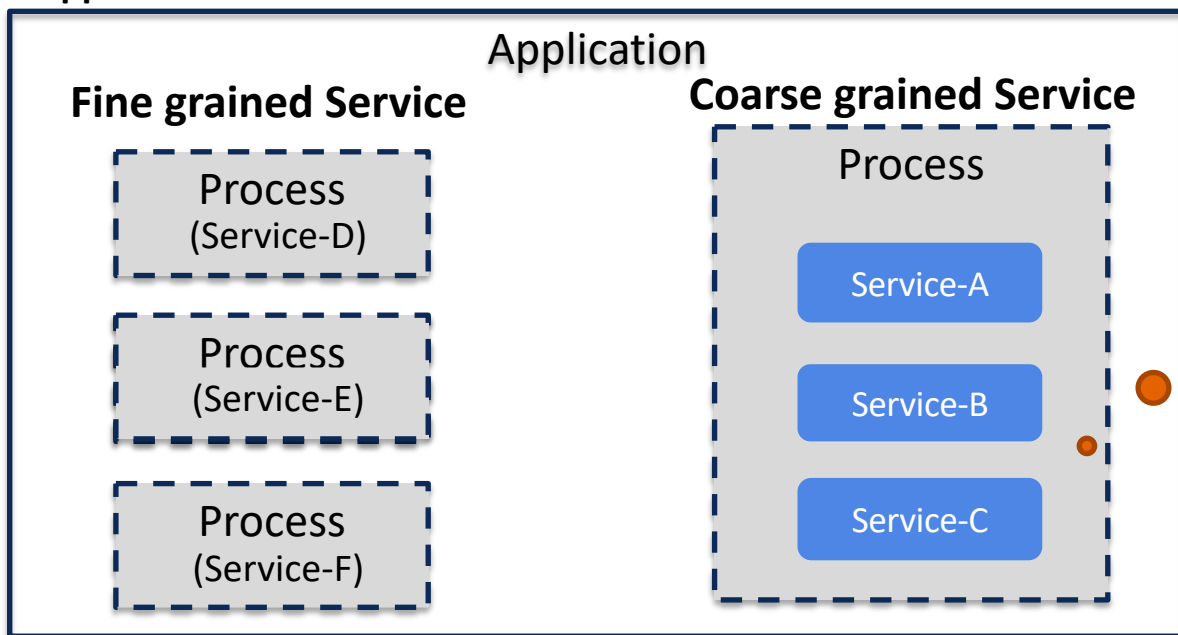




# 产品化增强-支持粗粒度的SOA类应用

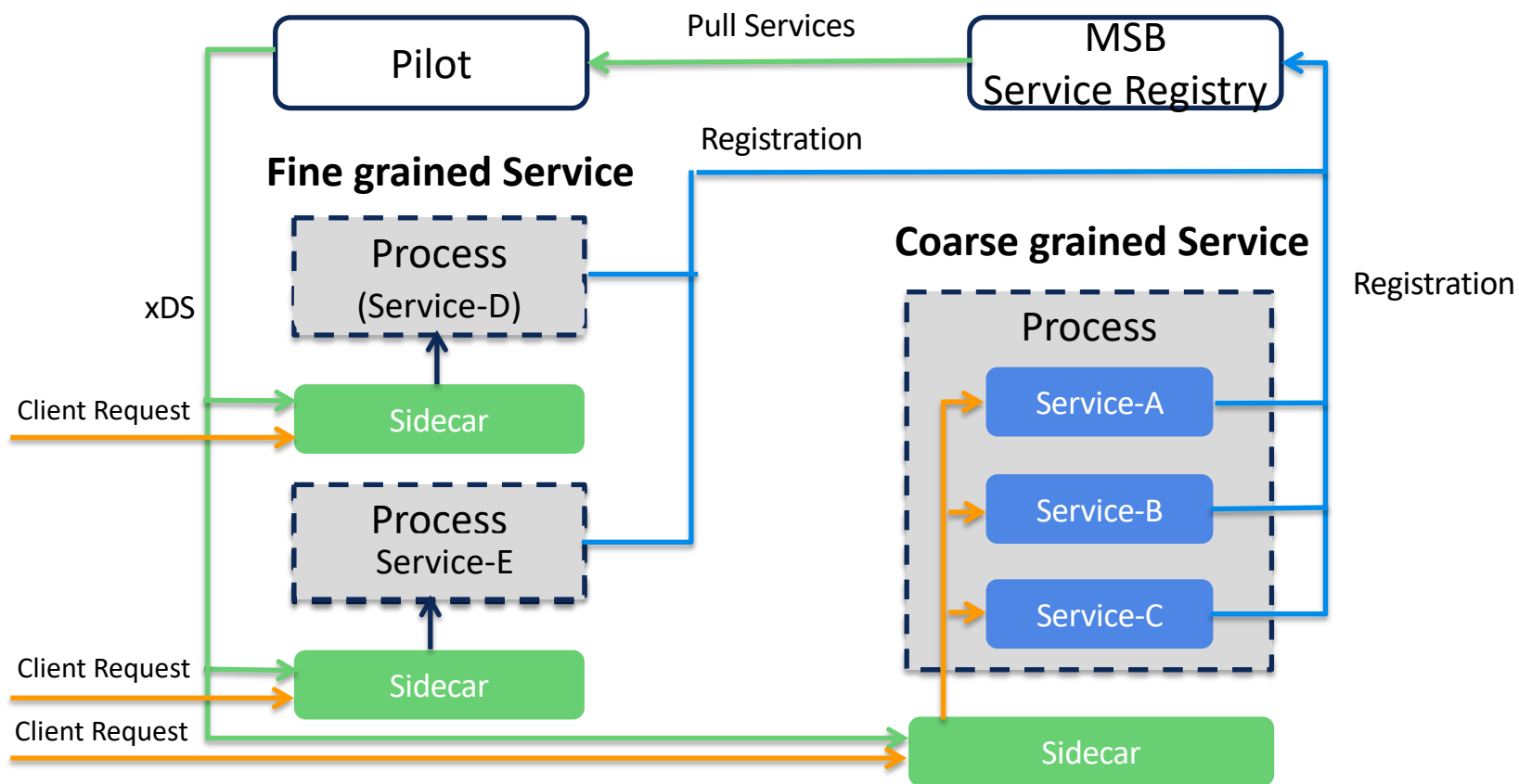
Kubernetes and Istio assumes that one process/container is a service. While this assumption is fine with “pure” Microservices architecture, it does have problems supporting “Coarse Grained” Service.

**An example application:** Combination of “fine Grained” and “Coarse Grained” Services



How to map multiple logic services inside one process to Istio service?

## 产品化增强-支持粗粒度的SOA类应用



# 产品化增强-Ingress API Gateway



## K8S Ingress

Load balancing  
SSL termination  
Virtual hosting

提供七层网关能力，  
但和服务网格是割裂的



## Istio Gateway

Load balancing  
SSL termination  
Virtual hosting  
Advanced traffic routing  
Fault injection  
Other benefits brought by Istio

提供七层网关和网格能力，  
但缺少API管理能力

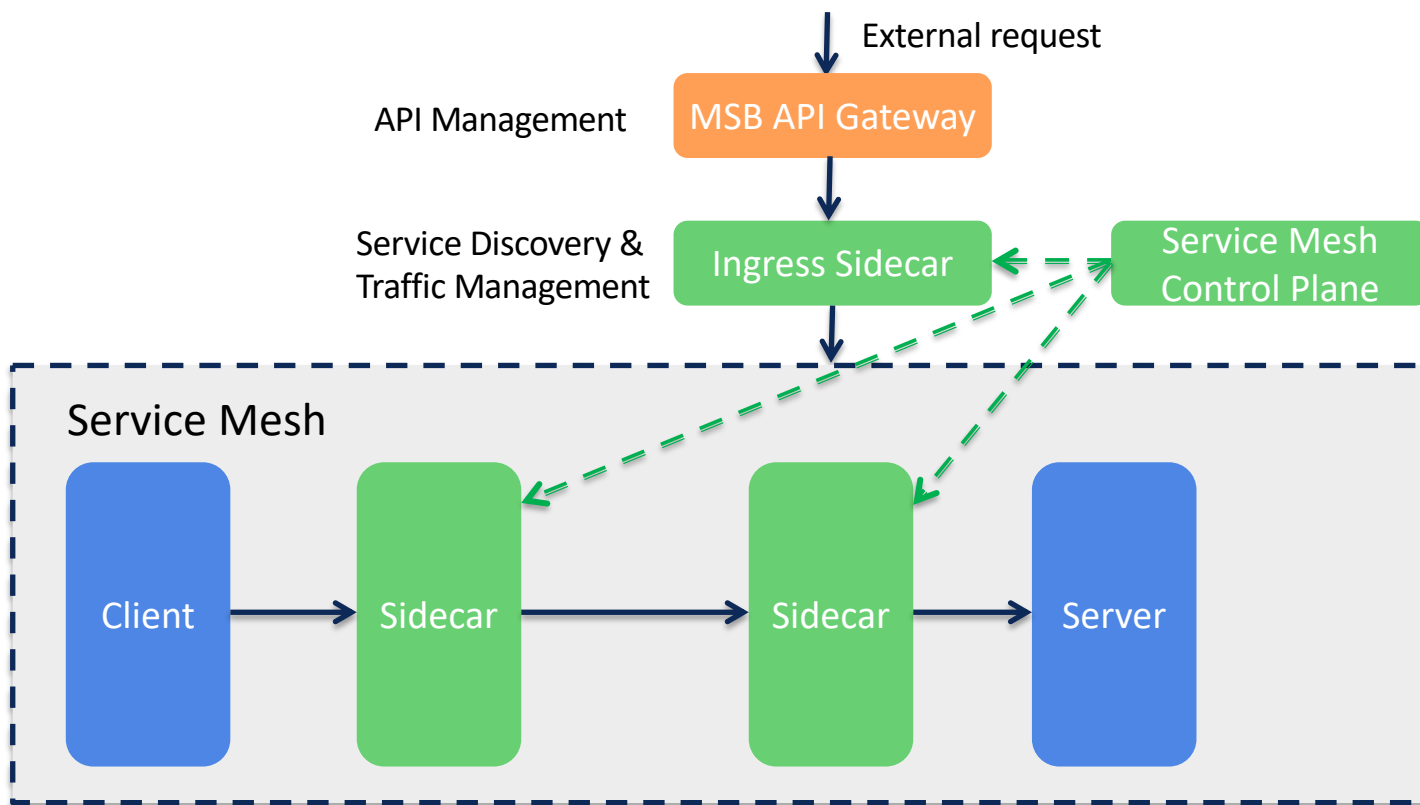


## API Gateway

Load balancing  
SSL termination  
Virtual hosting  
Traffic routing  
API lifecycle management  
API access monitoring  
API access authorization  
API key management  
API Billing and Rate limiting  
Other business logic ...

提供API管理能力，  
缺少服务网格能力

# 在DexMesh场景下Mesh和API Gateway的分工与协同



## API Gateway: 应用网关逻辑

- 使用不同端口为不同租户提供访问入口
- 租户间的隔离和访问控制
- 用户层面的访问控制
- 按用户的API访问限流
- API访问日志和计费

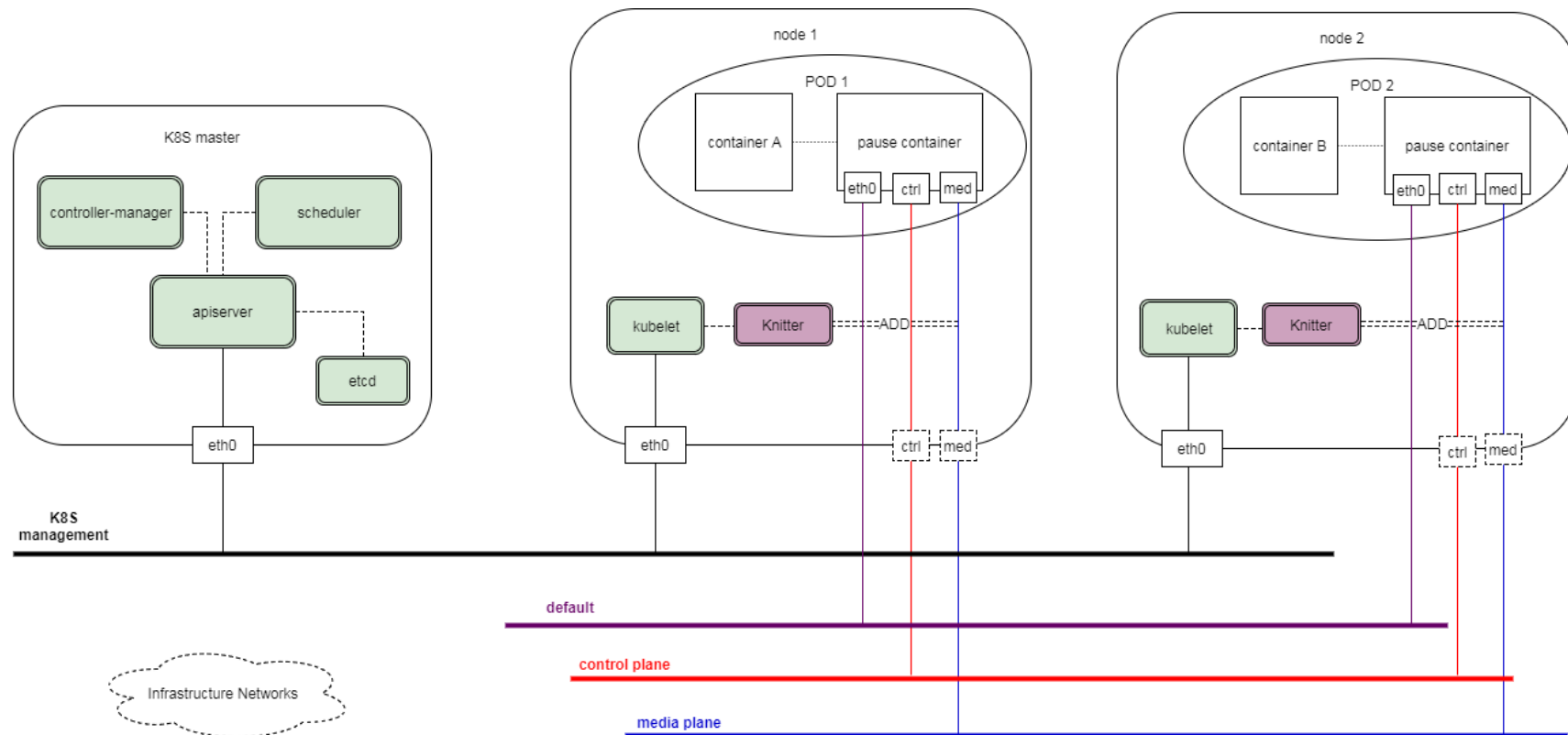
## Service Mesh: 统一的微服务通信管理

- 服务发现
- 负载均衡
- 重试，断路器
- 故障注入
- 分布式调用跟踪
- Metrics 收集

© ZTE All rights reserved

## 产品化增强-支持多网络平面

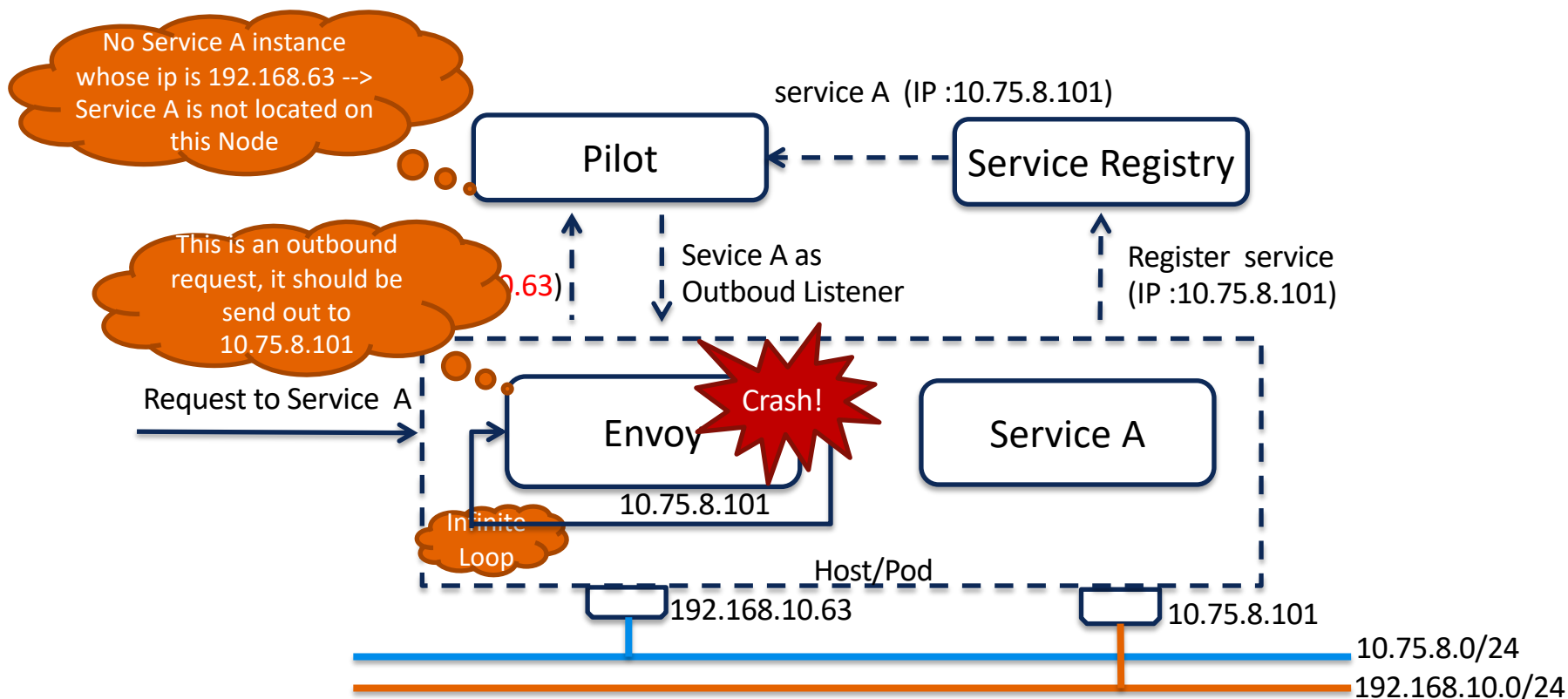
电信系统一般会有多个网络平面的，主要原因包括：避免不同功能的网络之间的相互影响；网络设计冗余，增强系统网络的健壮性；为不同的网络提供不同的SLA；通过网络隔离提高安全性；通过叠加多个网络增加系统带宽



上图中的Kubernetes集群使用了Knitter网络插件，部署了四个网络平面

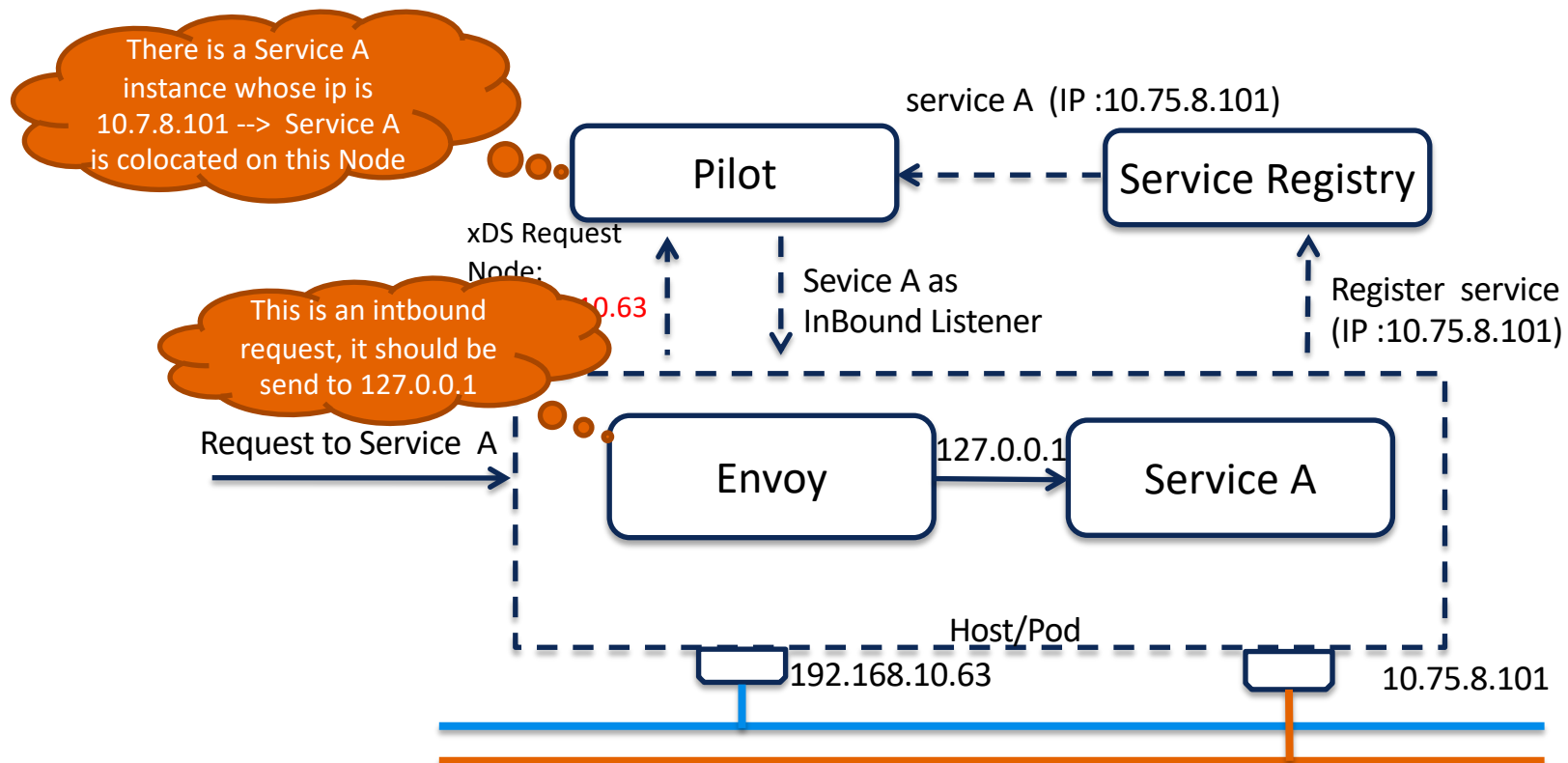
## 产品化增强-支持多网络平面

Istio1.0中不支持多网络平面，当服务地址和Envoy地址分别位于两个网络上时，会导致转发请求时发生死循环，导致socket耗尽，Envoy不停重启。



# 产品化增强-支持多网络平面

我们对Istio的代码进行了改造，增加了多网络平面支持。



# 产品化增强-TCP Service的处理

是否需要将**TCP Service**纳入**Service Mesh**管控？

- 收益

- **TCP Service**可以享受流量管理，可见性，策略控制等**Istio**承诺的益处

- 成本

- **Istio**不理解**TCP**上的应用层协议，其对**TCP Service**的缺省处理会影响应用层逻辑
  - 要求对应用进行逐一分析，将会导致应用逻辑受到影响的**TCP Service**排除
- **Istio**中和**HTTP Service** 端口冲突会的**TCP Service**请求会被**Envoy**直接丢弃
  - 要求对应用进行改造，避免端口冲突

由于系统中绝大部分的微服务间通讯都是基于**HTTP**的，并且存量应用难以改动，因此将**TCP**纳入**Service Mesh**管控的成本远大于收益



# 产品化增强-TCP Service的处理

在Service Mesh中 Bypass TCP流量，让TCP请求跳过Service Mesh的处理，直接发送到原始请求目的地。

- 方案一：通过IPtables bypass TCP流量  
通过IP段或者端口范围区分HTTP和其他TCP流量  
- 需要对应用进行改造
- 方案二：在Envoy中 bypass TCP 流量  
- 不需要对应用进行改造，但Envoy要具备区分TCP和HTTP流量的能力，需对Envoy进行改造

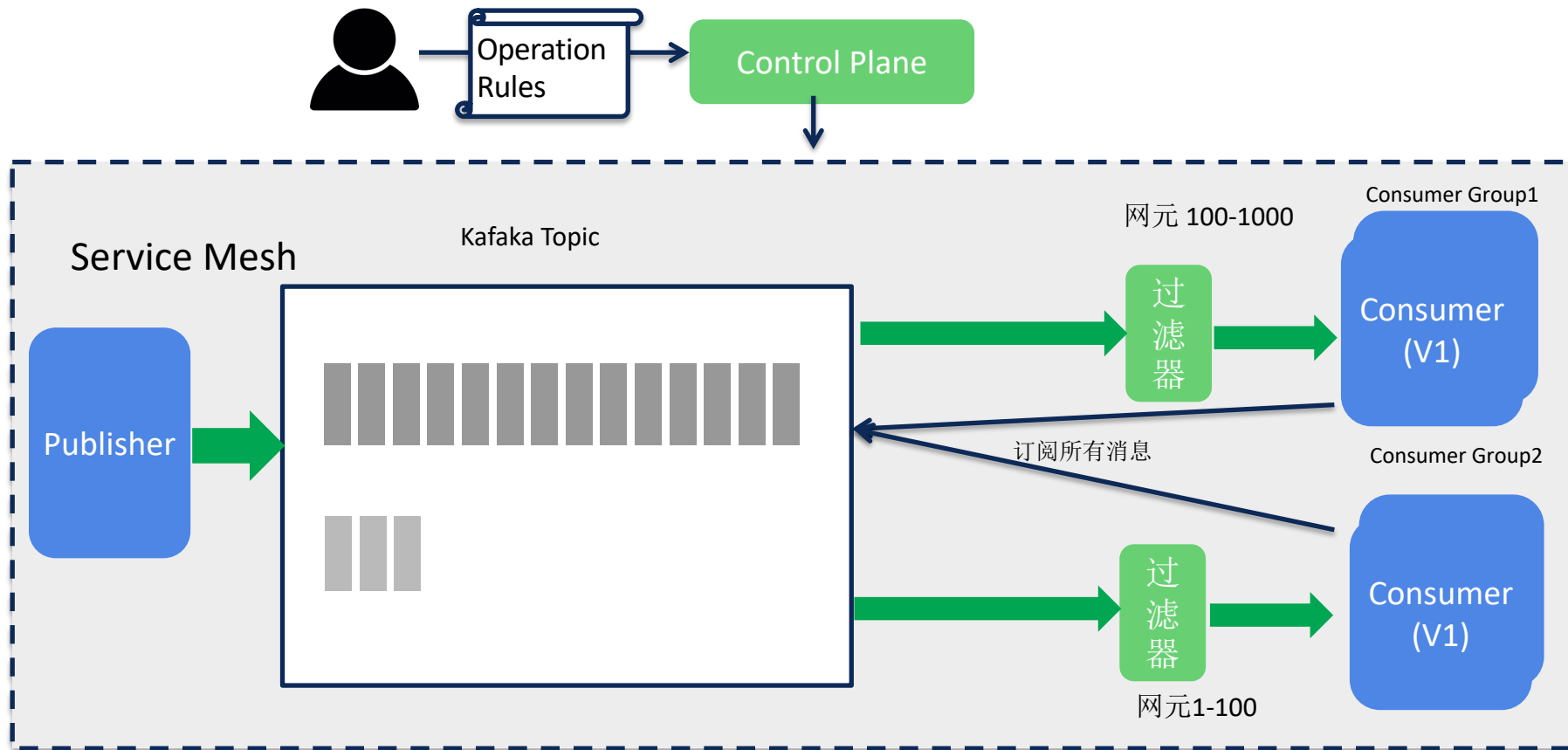
改造方案：

- Envoy侧：通过一个自定义的envoy listener filter区分HTTP和非HTTP的TCP流量
- Pilot侧：修改Pilot下发的LDS配置，将TCP流量转到一个指定的filter chain 处理，通过tcp\_proxy filter将TCP请求发往Passthroughfilter，以达到bypass TCP流量的目的。

```
listener
├── name : 0.0.0.0_12011
├── address
│   └── socket_address
│       ├── address : 0.0.0.0
│       └── port_value : 12011
├── filter_chains :
│   ├── 0
│   │   ├── filter_chain_match
│   │   │   └── server_names :
│   │   │       HTTP.DATA.COM
│   │   └── filters :
│   │       ├── 0
│   │       │   ├── name : envoy.http_connection_manager
│   │       │   └── config
│   │       └── 1
│   │           ├── filter_chain_match
│   │           │   └── server_names :
│   │           └── filters :
│   │               ├── 0
│   │               │   ├── name : envoy.tcp_proxy
│   │               │   └── config
│   │               │       ├── stat_prefix : PassthroughCluster
│   │               │       └── cluster : PassthroughCluster
│   └── deprecated_v1
└── listener_filters :
    ├── 0
    │   └── name : envoy.listener.http_inspector
```

# 产品化增强：异步通信的流量管理

微服务有两种通讯模式：同步RPC和异步消息。Istio目前缺少对异步消息流量的管理。























- 两个版本的应用实例分别属于不同的Consumer Group
- 通过客户端过滤器来进行消息分流
- 在升级过程中，消息会被重复消费一次（客户端过滤）
- 通过Kafka消息头和控制面下发的导流规则对Kafka消息进行过滤，以达到在不同应用中对Kafka消息进行导流的目的

## 产品化增强-其他

- 管理界面：服务编排、灰度发布及服务管控
- Consul Registry故障修复及效率优化
- IPV6支持
- 网络参数调优
- 容器镜像轻量化

# 上游开源社区参与情况

将产品化过程中的优化和改进以PR合入了上游Istio社区：包含一些通用的故障修复、性能优化及功能特性的

<input type="checkbox"/>	 <b>Fix: Consul high CPU usage (#15509)</b>  <span>approved</span> <span>area/perf and scalability</span> <span>cla: yes</span> <span>lgtn</span>   4
#15510 by zhaohuabing was merged 15 hours ago • Approved	
<input type="checkbox"/>	 <b>Remove warn log message of ignored Consul service tag</b>  <span>approved</span> <span>area/user experience</span> <span>cla: yes</span> <span>lgtn</span>   8
#15452 by zhaohuabing was merged 11 days ago • Approved	
<input type="checkbox"/>	 <b>Avoid unnecessary service change events(#11971)</b>  <span>cla: yes</span> <span>ok-to-test</span>   4
#12148 by zhaohuabing was merged on Mar 1 • Approved	
<input type="checkbox"/>	 <b>Use ServiceMeta to convey the protocol and other service properties</b>  <span>approved</span> <span>cla: yes</span> <span>lgtn</span>   13
#9713 by zhaohuabing was merged on Nov 10, 2018 • Approved	
<input type="checkbox"/>	 <b>Support multiple network interfaces(#9441)</b>  <span>approved</span> <span>cla: yes</span>   70
#9688 by zhaohuabing was merged on Dec 15, 2018 • Approved	

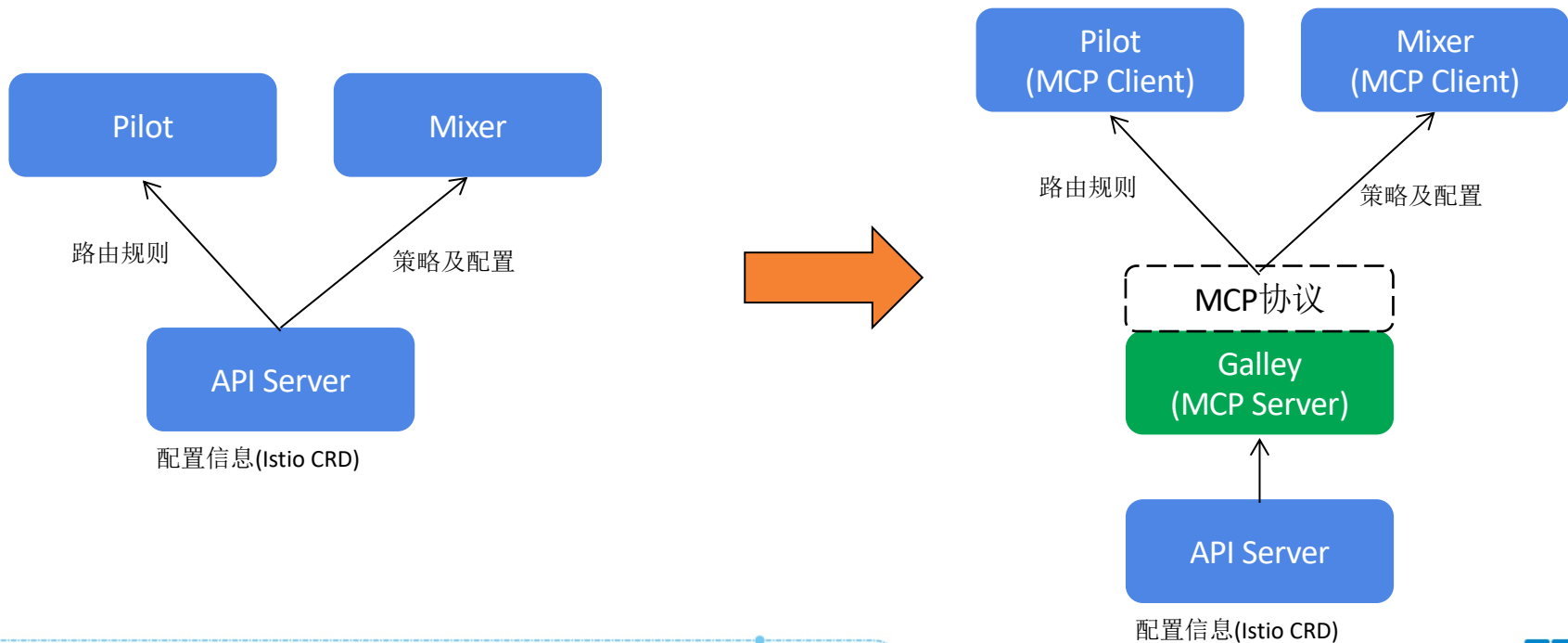
# Istio项目演进趋势-网格配置

## 现状及问题:

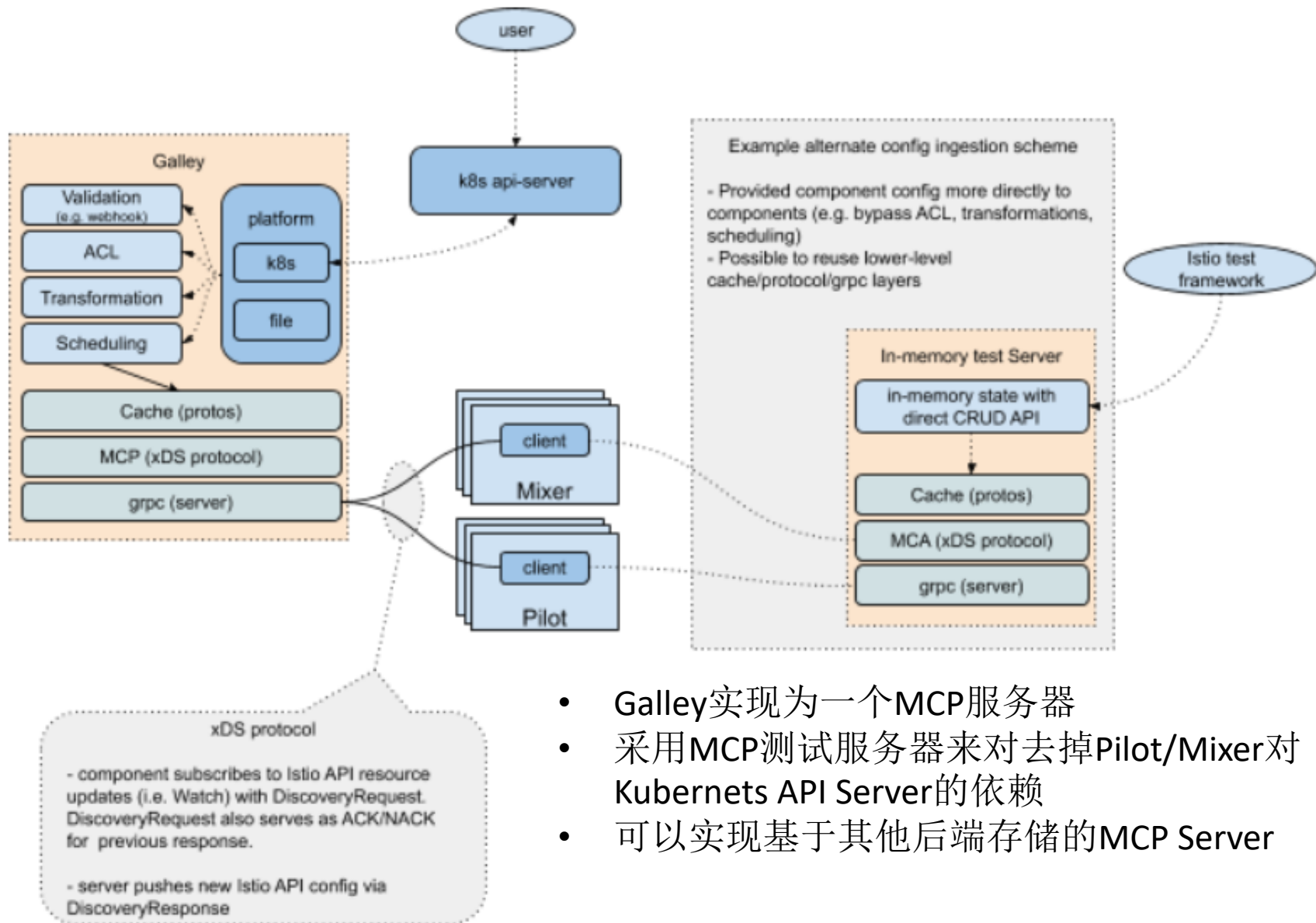
- 控制面组件从API Server直接获取配置数据（Istio CRD）
- 控制面组件和Kubernetes API Server耦合紧密，难以测试
- 缺乏统一的配置管理，各个组件各自订阅配置
- 缺乏配置数据的隔离机制，ACL控制

## 演进方式:

- 采用MCP（Mesh Configuration Protocol）标准接口来下发配置数据
- 采用galley来管理Kubernetes API Server中的配置数据，提供CRD的 validation，缓存，同步，ACL等配置数据的统一管理



# Istio项目演进趋势-网格配置



- Galley实现为一个MCP服务器
- 采用MCP测试服务器来对去掉Pilot/Mixer对Kubernetes API Server的依赖
- 可以实现基于其他后端存储的MCP Server

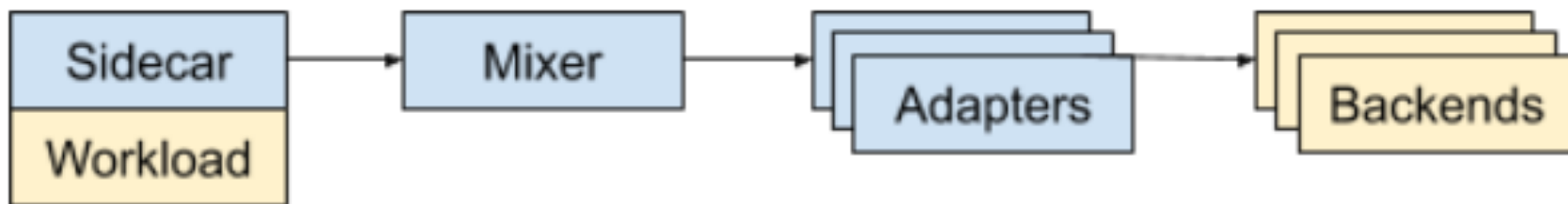
# Istio项目演进趋势-Mixer的演进

**Mixer V1架构：** Mixer作为控制面的一个独立组件

优势：灵活的适配器架构，可以方便地接入多个不同的Backend系统

劣势：

- Envoy在业务调用链中加入了对Mixer的远程调用，带来了较大的性能开销
- 网格中的大量Telemetry和policy check都需要经过Mixer，Mixer成为系统瓶颈
- 采用一个template来抽象不同Backend系统需要的数据，需要编写大量复杂的配置文件

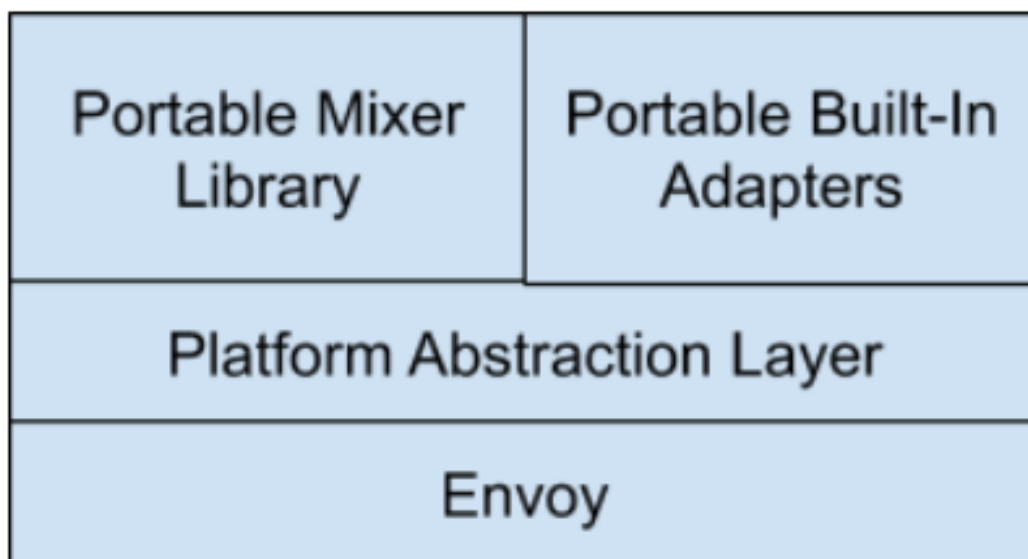


# Istio项目演进趋势-Mixer V2 Proposal

Mixer V2: 将Mixer作为插件内置到Envoy中

内置Mixer架构:

- **PAL:** 为各个Adapter实现基础平台，屏蔽不同Proxy之间的差异。将来会提供在web assembly runtime上运行插件的能力。
- **Adapter:** 实现各种业务功能的插件。



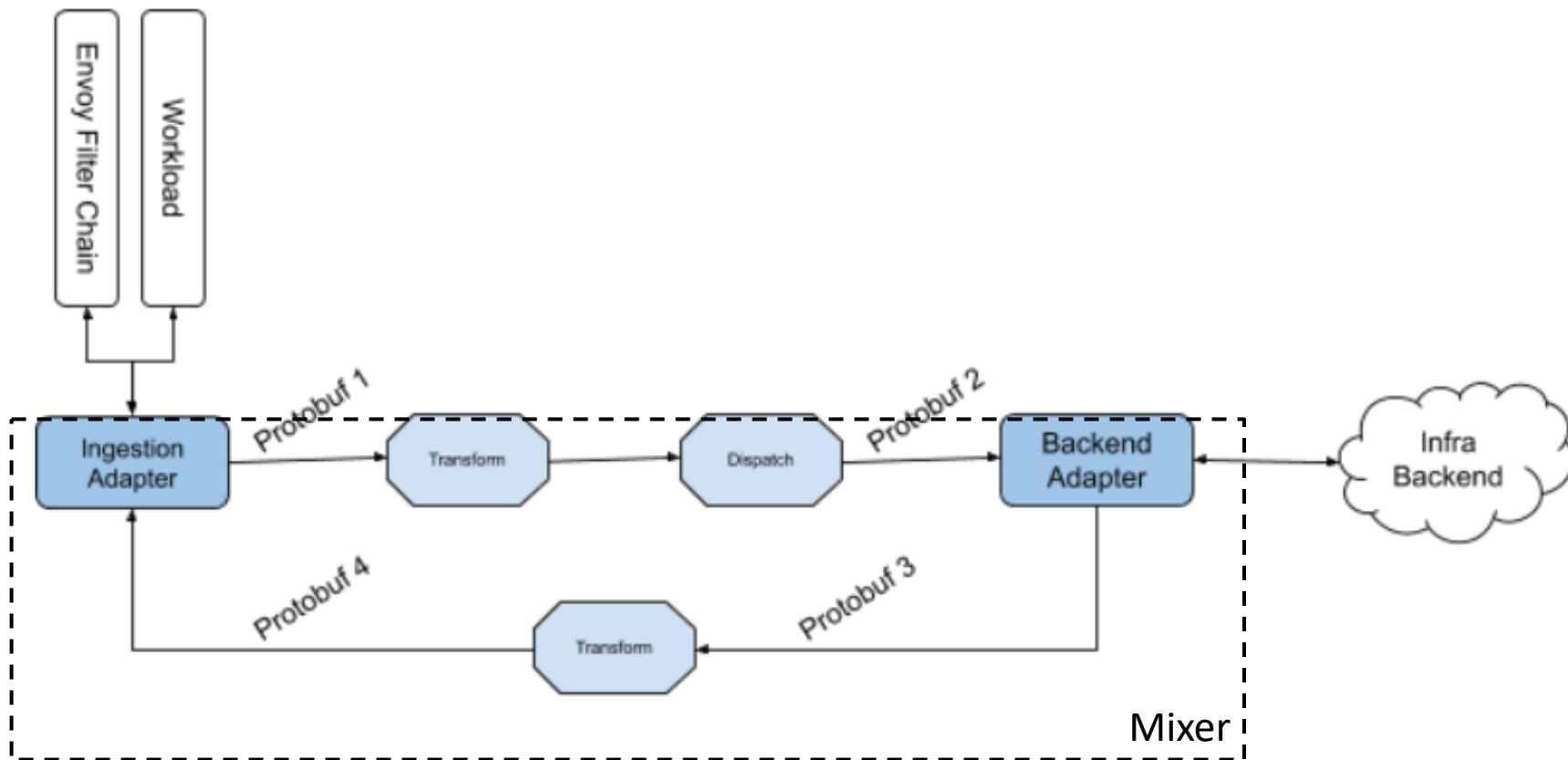


# Istio项目演进趋势-Mixer V2 Proposal

Mixer V2 业务处理流程:

Mixer V2由一系列Adapter组成，Adapter分为两类

- Ingestion Adapter: 拦截请求，并将请求解析为一个Backend Adapter可以理解的数据结构
- Backend Adapter: 消费解析的数据结构，并结合后端系统实现特定功能，例如

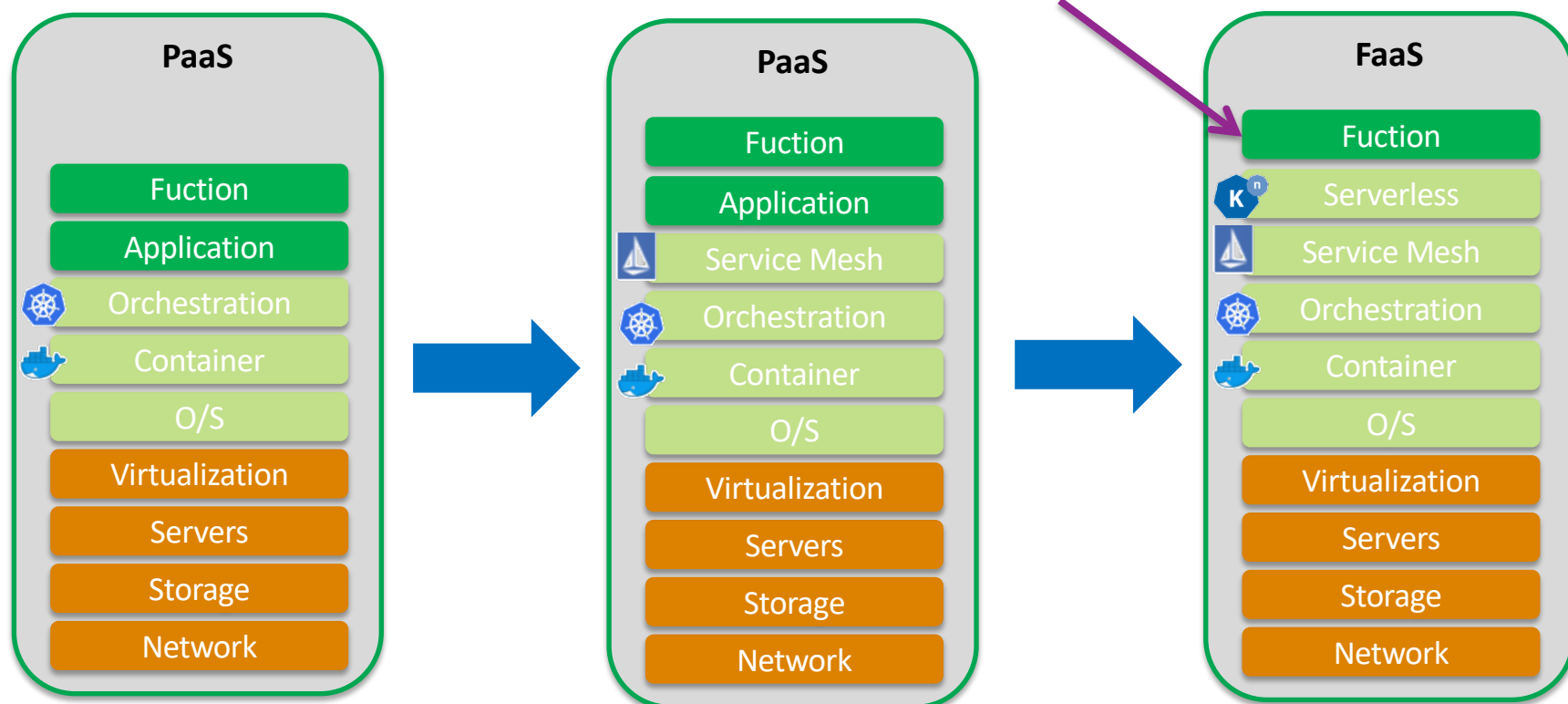


# Service Mesh 发展趋势-下沉为公有云基础设施

Service Mesh简化了微服务的应用开发，但Service Mesh层自身的技术复杂度较高，实现和运维都比较困难。要享用Service Mesh带来的便利，最方便的方式便是使用云提供上的Service Mesh托管服务，将底层的复杂性交给云提供商。

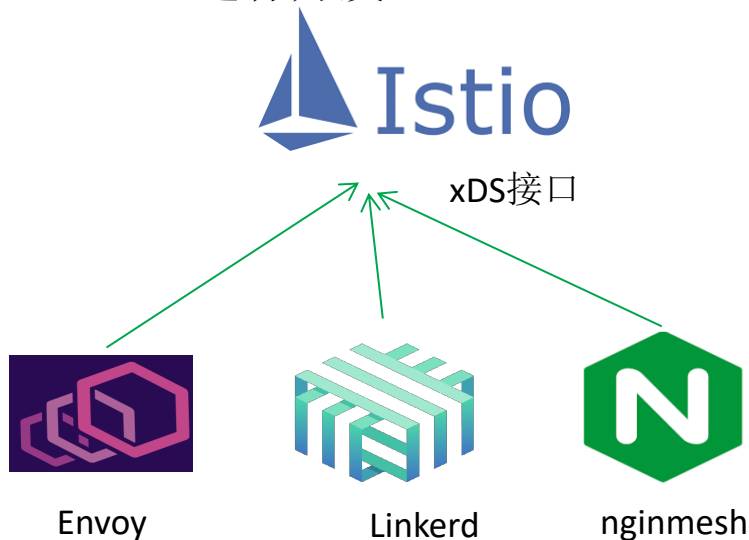
目前Google，AWS，微软，阿里，腾讯，华为等公有云提供商都已经提供了公有云上的Service Mesh托管服务。

Leave all the others to the platform except **business logic**



# Service Mesh 发展趋势-标准化

Istio 1.0及以前：Istio横空出世，其他项目向Istio的大旗靠拢，要么作为数据面和其集成，要么基于Istio之上进行开发。



- 2017年初 Buoyant CEO William Morgan正式提出Service Mesh的定义
- 2017.1 Linkerd作为第一个Service Mesh项目加入CNCF
- 2017.4 Google, IBM和Lyft公开Istio项目：Istio提出了控制面和标准数据面接口（xDS）的概念，占领Service Mesh架构高度
- 2017年5月发布Istio 0.1版本
- 2017年7月 Linkerd宣布其1.1.1版本支持和Istio集成
- 2017年8月Nginx开源基于Istio的数据面项目Nignmesh

# Service Mesh 发展趋势-标准化

2018-2019: Service Mesh生态蓬勃发展，多个Service Mesh开源及闭源项目相继涌现：

- Buoyant暗渡陈仓，发布Linkerd2, 包括数据面和控制面组件，不再支持和Istio集成
- Consul 发布connect, 支持Service mesh功能，支持采用envoy作为数据面
- 各公有云厂商纷纷发布Service Mesh管理服务
- Solo.io发布Service Mesh管理平台SuperGloo，支持多云，多Mesh的管理

随着Service Mesh的应用，**跨Service Mesh之间的互通和兼容性**开始成为一个亟待解决的问题。

多Mesh管理:



控制面项目:



Istio



Linkerd2



Consul



Traffic Director



Azure Service  
Fabric Mesh



AWS App Mesh

数据面项目:



Envoy



Linkerd2-proxy



Consul connect



Gloo



AMBASSADOR

# Service Mesh 发展趋势-标准化

- **数据面标准化**: Google主导在CNCF成立Universal Data Plane API（通用数据平面API）工作组，基于xDS为基础指定L4/L7的数据面标准接口,意在建立Istio的生态系统。
- **控制面标准化**: 微软牵头，联合 Linkerd, HashiCorp, Solo等发起Service Mesh Interface（SMI），定义控制面的标准规范，以期望实现各个Service Mesh的互通性，挑战目前Istio在控制面上的垄断地位。

## 基于Mesh的应用:

(跨Mesh的管理、流量调整、全局策略、全局可见性等)



## Service Mesh Interface

控制面项目



Linkerd2



Consul



Traffic Director



Azure Service  
Fabric Mesh



AWS App Mesh

## Universal Data Plane API

数据面项目



Envoy



Linkerd2-proxy



Consul connect



Gloo



AMBASSADOR

谢谢！

- 国内Service Mesh社区  
<http://servicemesher.com>
- 博客  
<http://zhaohuabing.com>
- <https://medium.com/@zhaohuabing>
- Github  
<https://github.com/zhaohuabing>
-