

# 全栈服务网格 - Aeraki 助你在 Istio 服务网格中管理任何七层流量

赵化冰@腾讯云

# Huabing Zhao


Software Engineer @ Tencent Cloud

 @zhaohuabing

 @zhaohuabing

 @zhaohuabing

 @zhaohuabing

 <https://zhaohuabing.com>



# Agenda

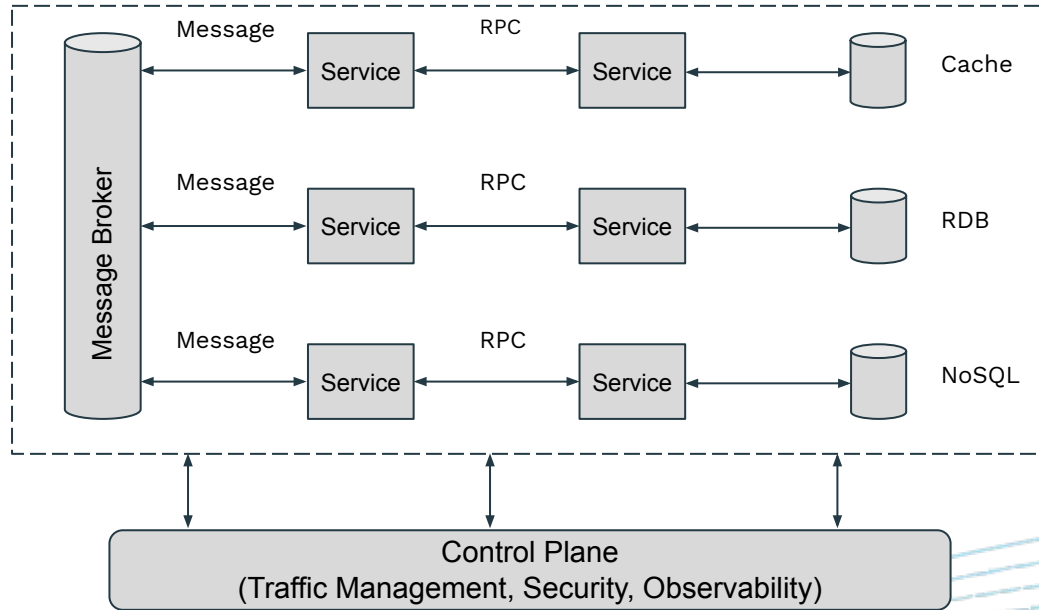
- ❑ Service Mesh 中的七层流量管理能力
- ❑ 几种扩展 Istio 流量管理能力的方法
- ❑ Aeraki - 在 Istio 服务网格中管理所有七层流量
- ❑ Demo - Dubbo Traffic Management
- ❑ MetaProtocol - Service Mesh 通用七层协议框架



# Protocols in a Typical Microservice Application

We need to manage multiple types of layer-7 traffic in a service mesh, not just HTTP and gRPC

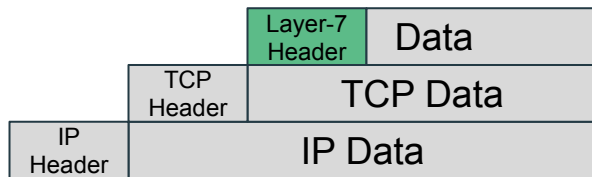
- **RPC:** HTTP, gRPC, Thrift, Dubbo, Proprietary RPC Protocol ...
- **Messaging:** Kafka, RabbitMQ ...
- **Cache:** Redis, Memcached ...
- **Database:** MySQL, PostgreSQL, MongoDB ...
- **Other Layer-7 Protocols:** ...



# What Do We Expect From a Service Mesh?

为了将基础设施的运维管理从应用代码中剥离，我们需要七层的流量管理能力：

- Routing based on layer-7 header
  - Load balancing at request level
  - HTTP host/header/url/method,
  - Thrift service name/method name
  - Dubbo Interface/method/attachment
  - ...
- Fault Injection with application layer error codes
  - HTTP status code
  - Redis Get error
  - ...
- Observability with application layer metrics
  - HTTP status code
  - Thrift request latency
  - ...
- Application layer security
  - HTTP JWT Auth
  - Redis Auth
  - ...



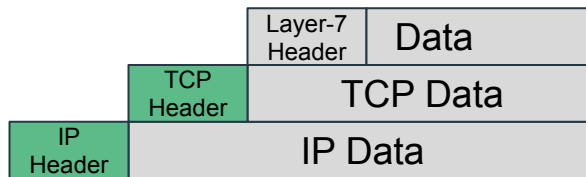
# What Do We Get From Istio?

## Traffic Management for HTTP/gRPC - all good

- We get all the capabilities we mentioned on the previous slide

## Traffic Management for non-HTTP/gRPC - only layer-3 to layer-6

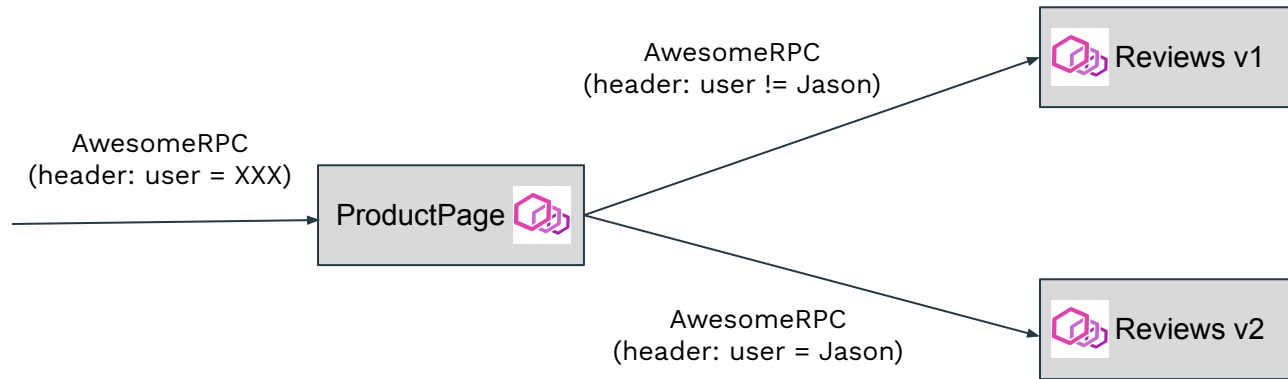
- Routing based on headers under layer-7
  - IP address
  - TCP Port
  - SNI
- Observability - only TCP metrics
  - TCP sent/received bytes
  - TCP opened/closed connections
- Security
  - Connection level authentication: mTLS
  - Connection level authorization: Identity/Source IP/ Dest Port
  - Request level auth is impossible



# BookInfo Application - AwesomeRPC

Let's say that we're running a bookinfo application in an Istio service mesh, but the inter-services communication are done by AwesomeRPC, our own RPC protocol, instead of HTTP.

So, how could we achieve layer-7 traffic management for AwesomeRPC in Istio?



# How to Manage AwesomeRPC Traffic in Istio?

## Istio Config

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews-route
spec:
  hosts:
  - reviews.prod.svc.cluster.local
  awesomeRPC:
  - name: "canary-route"
    match:
    - headers:
      user:
        exact: Jason
    route:
    - destination:
        host: reviews.prod.svc.cluster.local
        subset: v2
  - name: "default"
    route:
    - destination:
        host: reviews.prod.svc.cluster.local
        subset: v1
```

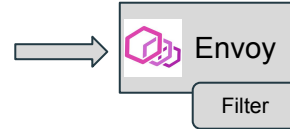


Code changes at the Pilot side:

- Add AwesomeRPC support in VirtualService API
- Generate LDS/RDS for Envoy

## Envoy Config

```
{
  "virtual_hosts": [
    {
      "name": "reviews.default.svc.cluster.local:9080",
      "services": [
        "reviews.default.svc.cluster.local",
        "reviews"
      ],
      "routes": [
        {
          "name": "canary-route"
          "match": {
            "headers": [
              {
                "name": ":user",
                "exact_match": "Jason"
              }
            ],
          },
          "route": {
            "cluster": "outbound|9080||reviews.default.svc.cluster.local | v2",
          },
        },
        {
          "name": "default"
          "route": {
            "cluster": "outbound|9080||reviews.default.svc.cluster.local | v1",
          },
        }
      ]
    }
  ],
}
```



AwesomeRPC Filter

- Decoding/Encoding
- Routing
- Load balancing
- Circuit breaker
- Fault injection
- Stats
- ...

## Pros:

- It's relatively easy to add support for a new protocol to the control plane, given that Envoy filter is already there

## Cons:

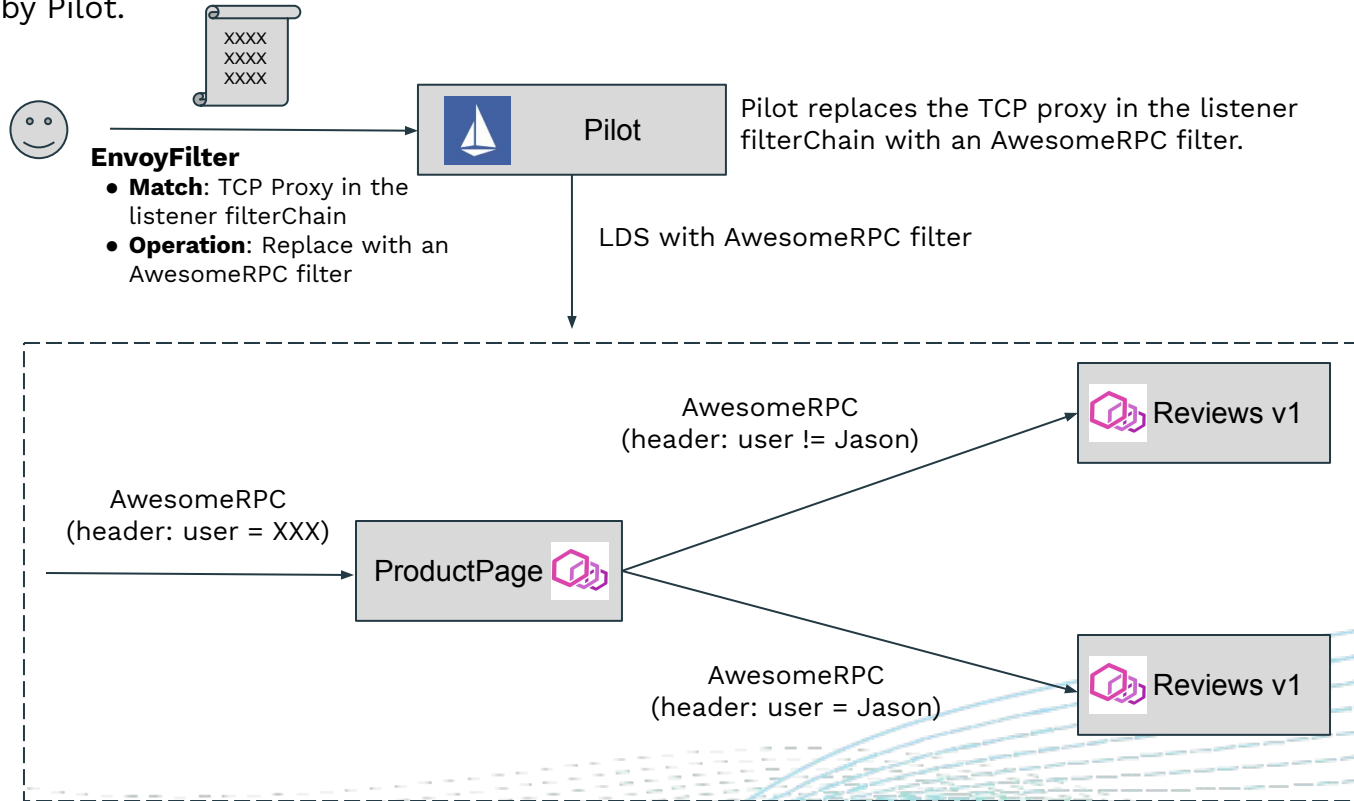
- You have to maintain a fork of Istio, which makes upgrade painful
- Writing an Envoy Filter for AwesomeRPC is painful





# Manage AwesomeRPC Traffic in Istio With EnvoyFilter

EnvoyFilter is an Istio configuration CRD, by which we can apply a “patch” to the Envoy configuration generated by Pilot.



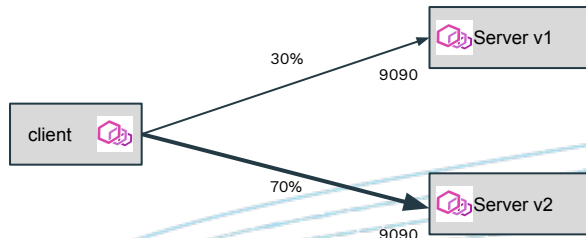
# EnvoyFilter Example - Dubbo Traffic Splitting

Replace TCP proxy in the outbound listener

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: envoyfilter-dubbo-proxy
  namespace: istio-system
spec:
  configPatches:
  - applyTo: NETWORK_FILTER
    match:
      listener:
        name: 193.193.192.192_20880
    filterChain:
      filter:
        name: "envoy.filters.network.tcp_proxy"
    patch:
      operation: REPLACE
      value:
        name: envoy.filters.network.dubbo_proxy
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.dubbo_proxy.v3.DubboProxy
          stat_prefix: outbound|20880||org.apache.dubbo.samples.basic.api.demoservice
          protocol_type: Dubbo
          serialization_type: Hessian2
          route_config:
            - name: outbound|20880||org.apache.dubbo.samples.basic.api.demoservice
              interface: org.apache.dubbo.samples.basic.api.DemoService
              routes:
                - match:
                    method:
                      name:
                        exact: sayHello
                  route:
                    weighted_clusters:
                      clusters:
                        - name: "outbound|20880|v1|org.apache.dubbo.samples.basic.api.demoservice"
                          weight: 30
                        - name: "outbound|20880|v2|org.apache.dubbo.samples.basic.api.demoservice"
                          weight: 70
```

Replace TCP proxy in the inbound listener

```
- applyTo: NETWORK_FILTER
match:
  listener:
    name: virtualInbound
  filterChain:
    destination_port: 20880
  filter:
    name: "envoy.filters.network.tcp_proxy"
patch:
  operation: REPLACE
  value:
    name: envoy.filters.network.dubbo_proxy
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.network.dubbo_proxy.v3.DubboProxy
      stat_prefix: inbound|20880||
      protocol_type: Dubbo
      serialization_type: Hessian2
      route_config:
        - name: inbound|20880||
          interface: org.apache.dubbo.samples.basic.api.DemoService
          routes:
            - match:
                method:
                  name:
                    exact: sayHello
              route:
                @listener: inbound|20880||
```



# EnvoyFilter is Powerful, But ...

It's very difficult if not possible to manually create and maintain these EnvoyFilters, especially in a large service mesh:

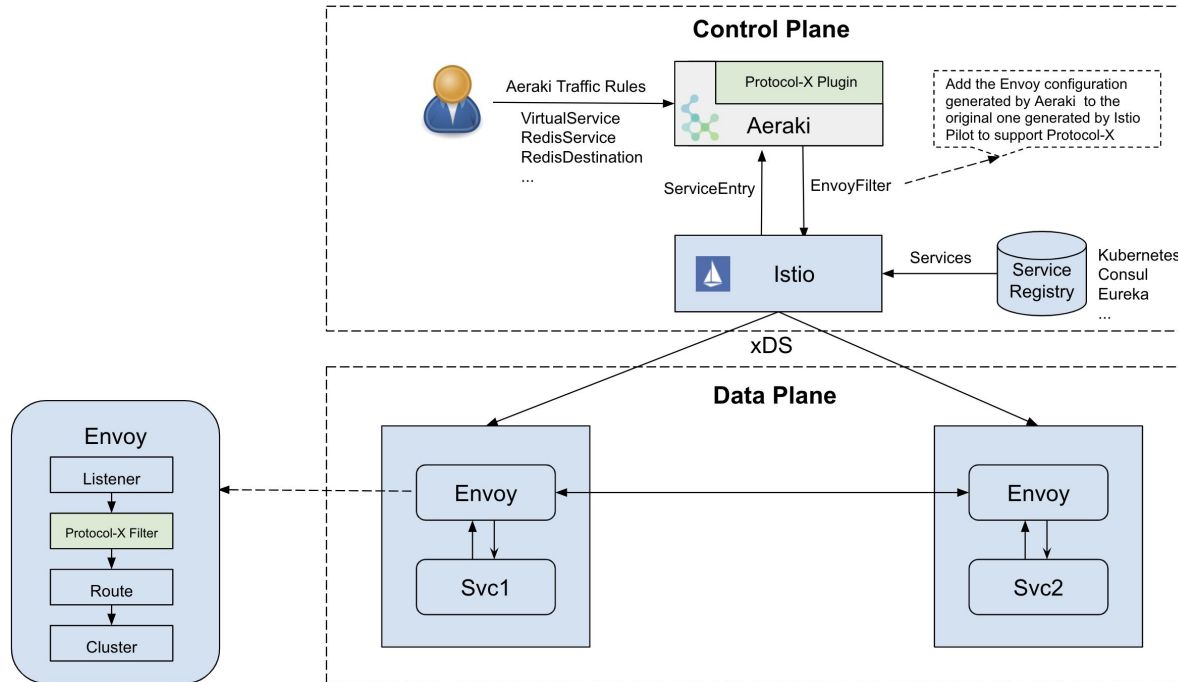
- It exposes low-level Envoy configurations to operation
- It depends on the structure/name convention of the generated xDS by Pilot
- It depends on some cluster-specific information such as service cluster IP
- We need to manually create tons of EnvoyFilter, one for each of the services

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: envoyfilter-dubbo-proxy
  namespace: istio-system
spec:
  configPatches:
  - applyTo: NETWORK_FILTER
    match:
      listener:
        name: 193.193.192.192_20880
        filterChain:
          filter:
            name: "envoy.filters.network.tcp_proxy"
    patch:
      operation: REPLACE
      value:
        name: envoy.filters.network.dubbo_proxy
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.dubbo_proxy.v3.DubboProxy
          stat_prefix: outbound|20880||org.apache.dubbo.samples.basic.api.demoservice
          protocol_type: Dubbo
          serialization_type: Hessian2
          route_config:
            - name: outbound|20880||org.apache.dubbo.samples.basic.api.demoservice
              interface: org.apache.dubbo.samples.basic.api.DemoService
              routes:
                - match:
                    method:
                      name:
                        exact: sayHello
                  route:
                    weighted_clusters:
                      clusters:
                        - name: "outbound|20880|v1|org.apache.dubbo.samples.basic.api.demoservice"
                          weight: 30
                        - name: "outbound|20880|v2|org.apache.dubbo.samples.basic.api.demoservice"
                          weight: 70
```



# Aeraki: Manage any layer-7 traffic in an Istio service mesh

Aeraki [Air-rah-ki] is the Greek word for 'breeze'. We hope that this breeze can help Istio sail a little further - to manage any layer-7 protocols other than just HTTP and gRPC. You can think of Aeraki as the “Controller” to automate the creation of envoy configuration for layer-7 protocols



# Aeraki: Manage any layer-7 traffic in an Istio service mesh

Aeraki has the following advantages compared with current approaches:

- Zero-touch to Istio codes, you don't have to maintain a fork of Istio
- Easy to integrate with Istio, deployed as a stand-alone component
- Provides an abstract layer with Aeraki CRDs, hiding the trivial details of the low-level envoy configuration from operation
- Protocol-related envoy configurations are now generated by Aeraki, significantly reducing the effort to manage those protocols in a service mesh
- Easy to control traffic with Aeraki CRDs (Aeraki reuses VR and DR for most of the RPC protocols, and defines some new CRDs for other protocols)

Supported Protocols:

- PRC: Thrift, Dubbo, tRPC
- Others: Redis, Kafka, Zookeeper,
- More protocols are on the way ...

Similar to Istio, protocols are identified by service port prefix in this pattern: tcp-protocol-xxxx. For example, a Thrift service port is named as "tcp-thrift-service". Please keep "tcp" at the beginning of the port name because it's a TCP service from the standpoint of Istio.

Visit Github to get more information <https://github.com/aeraki-framework/aeraki>



+



# Aeraki Configuration Example: Dubbo

## Service definition

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: test-dubbo-service
  annotations:
    interface: org.apache.dubbo.samples.basic.api.DemoService
spec:
  hosts:
    - org.apache.dubbo.samples.basic.api.demoservice
  addresses:
    - 193.193.192.192
  ports:
    - number: 20880
      name: tcp-dubbo
      protocol: TCP
  workloadSelector:
    labels:
      app: dubbo-sample-provider
  resolution: STATIC
```

## Traffic rules

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: test-dubbo-route
spec:
  hosts:
    - org.apache.dubbo.samples.basic.api.demoservice
  http:
    - name: "reviews-traffic-splitting"
      route:
        - destination:
            host: org.apache.dubbo.samples.basic.api.demoservice
            subset: v1
          weight: 30
        - destination:
            host: org.apache.dubbo.samples.basic.api.demoservice
            subset: v2
          weight: 70
```



# Aeraki Configuration Example: Redis

## RedisServie

```
apiVersion: v1
kind: Secret
metadata:
  name: redis-service-secret
type: Opaque
data:
  password: dGVzdHJlZGlzCg==
---
apiVersion: redis.aeraki.io/v1alpha1
kind: RedisService
metadata:
  name: redis-cluster
spec:
  host:
    - redis-cluster.redis.svc.cluster.local
  settings:
    auth:
      secret:
        name: redis-service-secret
  redis:
    - match:
        key:
          prefix: cluster
      route:
        host: redis-cluster.redis.svc.cluster.local
    - route:
        host: redis-single.redis.svc.cluster.local
```

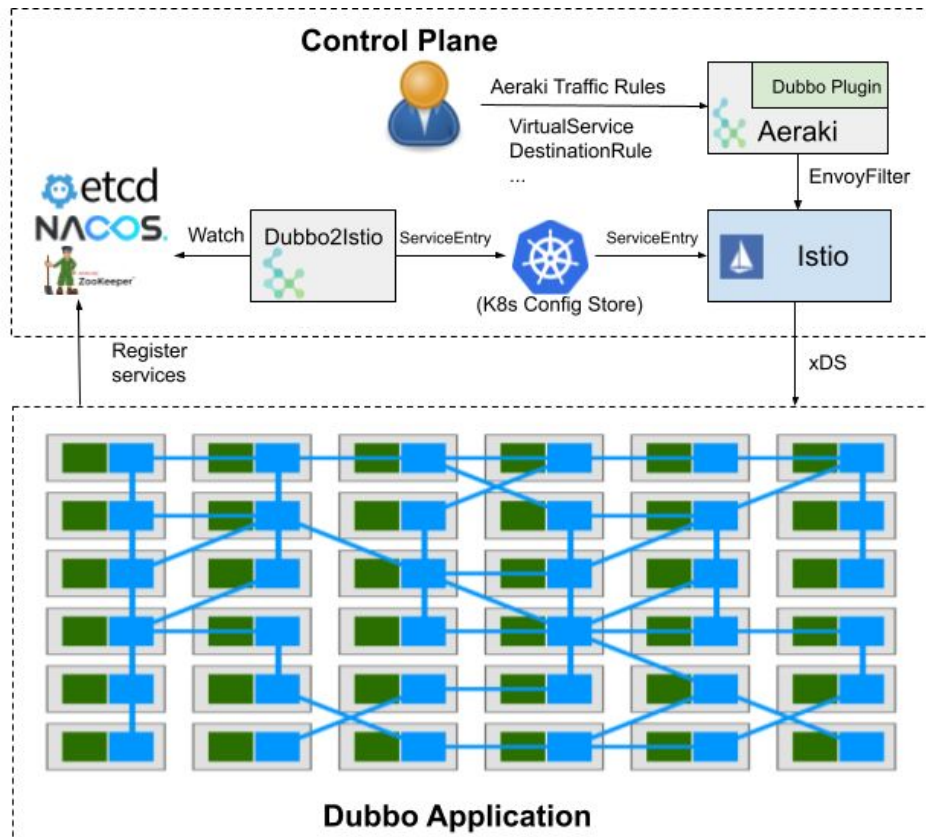
## RedisDestination

```
apiVersion: redis.aeraki.io/v1alpha1
kind: RedisDestination
metadata:
  name: redis-cluster
spec:
  host: redis-cluster.redis.svc.cluster.local
  trafficPolicy:
    connectionPool:
      redis:
        mode: CLUSTER
---
apiVersion: redis.aeraki.io/v1alpha1
kind: RedisDestination
metadata:
  name: redis-single
spec:
  host: redis-single.redis.svc.cluster.local
  trafficPolicy:
    connectionPool:
      redis:
        auth:
          plain:
            password: testredis
```



# Aeraki Demo: Dubbo 协议支持

- Dubbo2Istio 连接 Dubbo 服务注册表，支持：
  - ZooKeeper
  - Nacos
  - Etcd
- Aeraki Dubbo Plugin 实现了控制面的管理，支持下述能力：
  - 流量管理：
    - 七层（请求级别）负载均衡
    - 地域感知负载均衡
    - 熔断
    - 基于版本的路由
    - 基于 Method 的路由
    - 基于 Header 的路由
  - 可观测性：七层（请求级别）Metrics
  - 安全：基于 Interface/Method 的服务访问控制

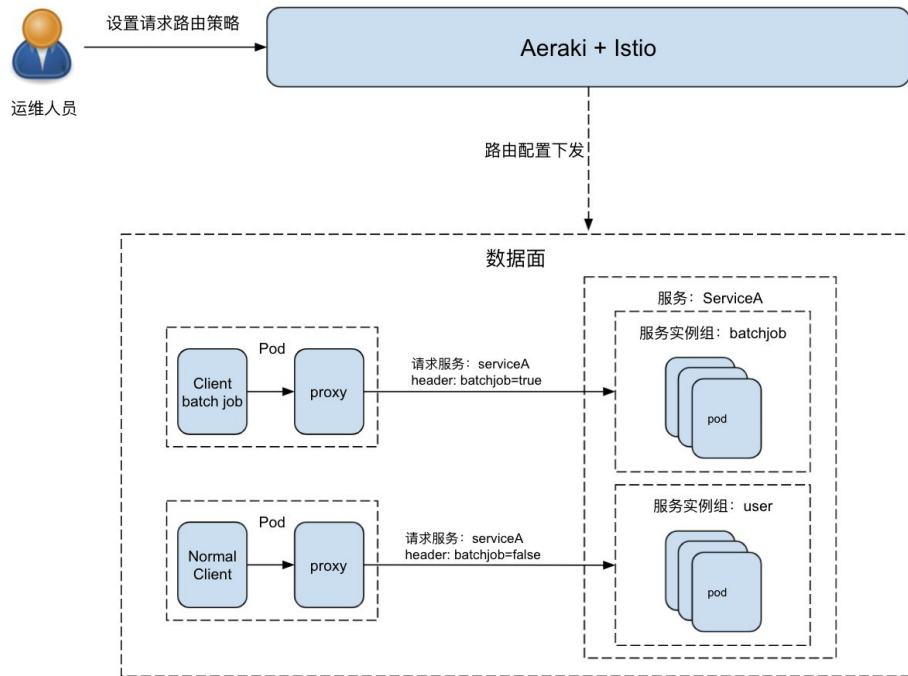




# Aeraki Demo: 用户请求和批处理任务隔离(Dubbo)

场景: 隔离处理用户请求和批处理任务的服务实例, 为用户请求留出足够的处理能力, 避免批处理任务的压力影响到用户体验。

- 将服务端划分为两个服务实例组, 分别用于处理批处理任务和用户请求。
- 客户端发起请求时通过一个“batchjob” header 标明请求的来源, batchjob=true表示该请求来自于批处理任务; batchjob=false表示该请求来自于用户请求。
- 运维人员设置请求路由规则, 将不同来源的请求路由到不同的服务实例组进行处理。



# Aeraki Demo: 用户请求和批处理任务隔离(Dubbo)

1. 在 dubbo: application 配置中为 Provider 增加 service\_group 自定义属性

```
<dubbo:application name="dubbo-sample-provider">  
  <dubbo:parameter key="service_group" value="${SERVICE_GROUP}" />  
</dubbo:application>
```

2. 通过 Provider 的 deployment 设置 SERVICE\_GROUP 环境变量
3. 在 consumer 发起调用时设置 batchJob header

```
RpcContext.getContext().setAttachment("batchJob", "true");  
DemoService demoService = (DemoService) context.getBean("demoService");  
String hello = demoService.sayHello("Aeraki");
```

4. 设置相应的 DR 和 VS 流量规则

```
apiVersion: networking.istio.io/v1beta1  
kind: DestinationRule  
metadata:  
  name: dubbo-sample-provider  
  namespace: dubbo  
spec:  
  host: org.apache.dubbo.samples.basic.api.demoservice  
  subsets:  
  - labels:  
    service_group: batchjob  
    name: batchjob  
  - labels:  
    service_group: user  
    name: user
```

```
apiVersion: networking.istio.io/v1beta1  
kind: VirtualService  
metadata:  
  name: test-dubbo-route  
  namespace: dubbo  
spec:  
  hosts:  
  - org.apache.dubbo.samples.basic.api.demoservice  
  http:  
  - name: route-batchjob  
    match:  
    - headers:  
      batchJob:  
        exact: "true"  
    route:  
    - destination:  
      host: org.apache.dubbo.samples.basic.api.demoservice  
      subset: batchjob  
  - name: route-user  
    match:  
    - headers:  
      batchJob:  
        exact: "false"  
    route:  
    - destination:  
      host: org.apache.dubbo.samples.basic.api.demoservice  
      subset: user
```

<https://docs.qq.com/doc/DVnlqUVB1ek1laFBQ>



# Aeraki Demo: 地域感知负载均衡(Dubbo)

场景:在开通地域感知负载均衡功能时, consumer 的请求将缺省被发送到本区域的 provider实例, 当本地域无可用的provider实例, 请求会被分发到其他区域。

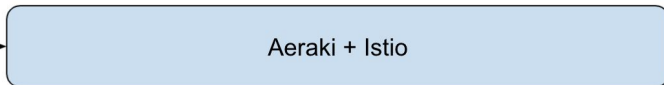
实现原理: 控制面根据 consumer 和 provider 的地域属性为provider实例配置不同的LB优先级, 优先级的判断顺序如下:

1. 最高: 相同 region, 相同zone
2. 其次: 相同 region, 不同zone
3. 再次: 不同 region

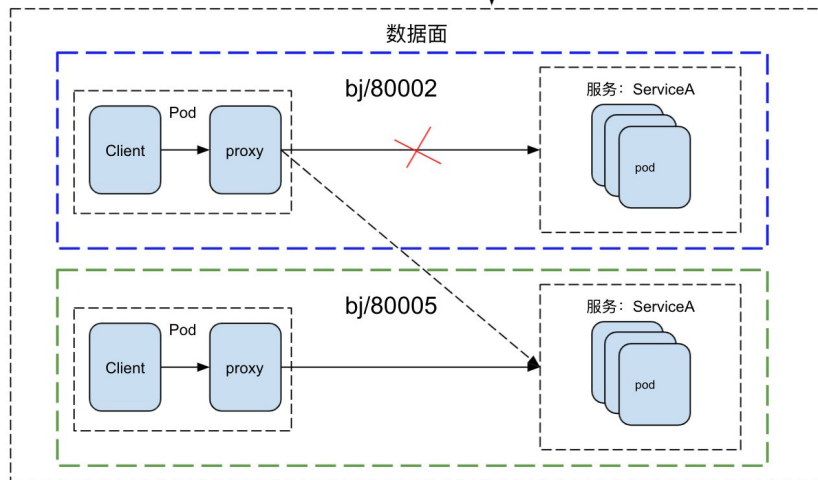


运维人员

设置请求路由策略



路由配置下发



# Aeraki Demo: 地域感知负载均衡(Dubbo)

1. 在 dubbo: application 配置中为 Provider 增加 aeraki\_meta\_locality 自定义属性

```
<dubbo:application name="dubbo-sample-provider">  
  <dubbo:parameter key="aeraki_meta_locality" value="${AERAKI_META_LOCALITY}" />  
</dubbo:application>
```

2. 在 provider 的 deployment 中通过环境变量设置其所属地域
3. 在 consumer 的 deployment 中通过 label 声明其所处的 region 和 zone
4. 通过 dr 规则启用 locality load balancing

```
apiVersion: networking.istio.io/v1alpha3  
kind: DestinationRule  
metadata:  
  name: dubbo-circuit-breaker-rule  
spec:  
  host: org.apache.dubbo.samples.basic.api.demoservice  
  trafficPolicy:  
    loadBalancer:  
      localityLbSetting:  
        enabled: true  
  outlierDetection:  
    baseEjectionTime: 5m  
    consecutive5xxErrors: 3  
    interval: 30s
```

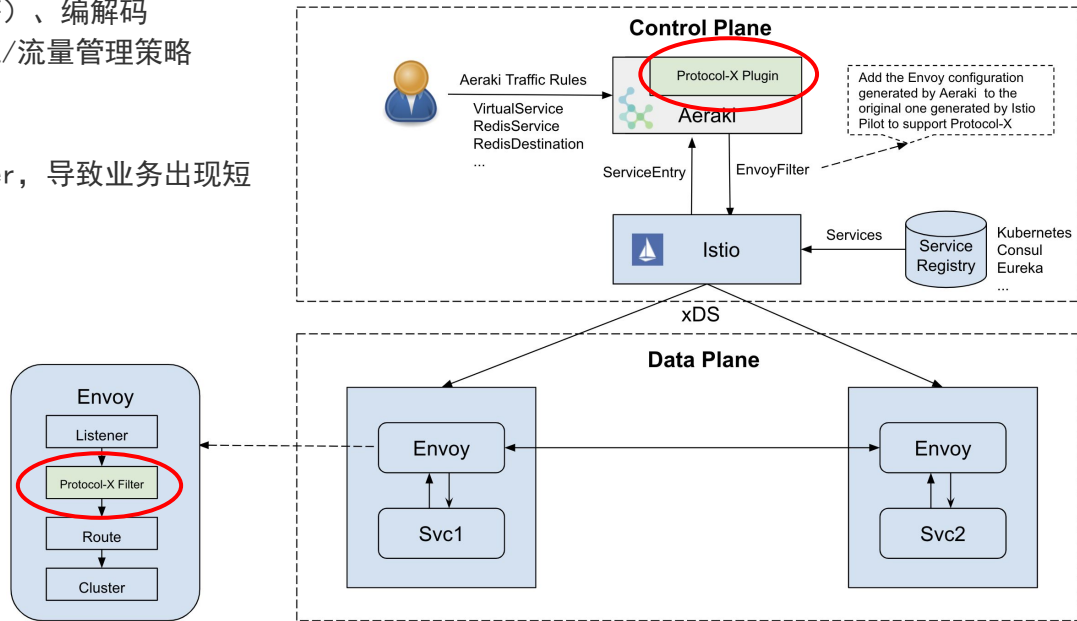
<https://docs.qq.com/doc/DVnlqUVB1ek1laFBQ>



# What's next ?

## 现阶段协议扩展方案面临的挑战：

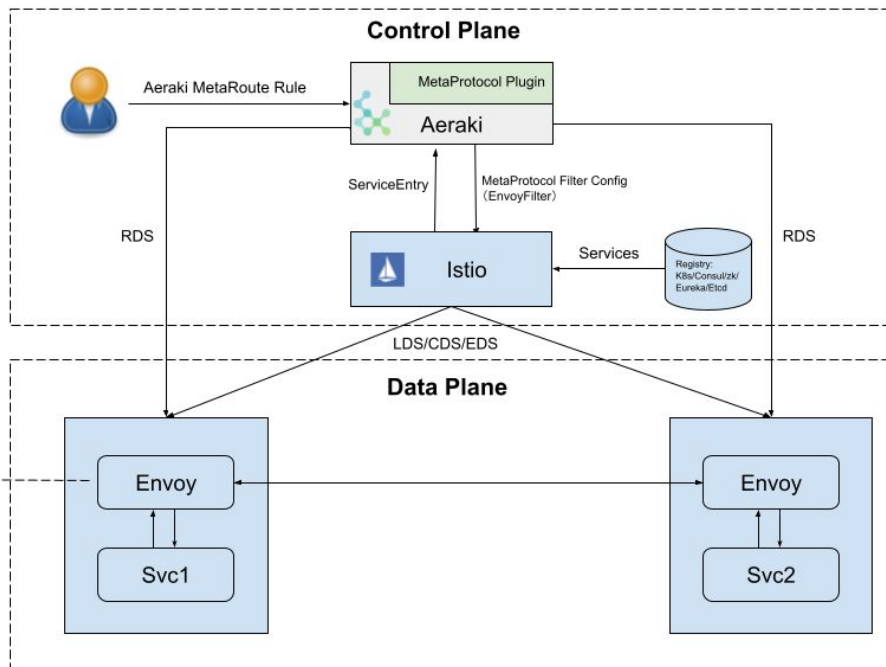
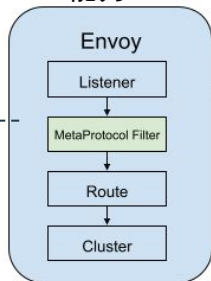
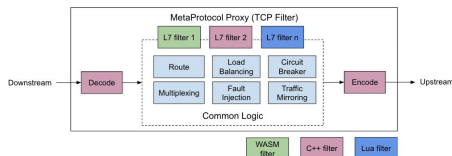
- 在 Mesh 中支持一个七层协议的工作量较大：
  - 数据面：编写一个 Envoy filter 插件——流量管理（RDS、负载均衡、熔断、流量镜像、故障注入等）、编解码
  - 控制面：编写一个 Aeraki 插件——运维/流量管理策略
- 非 HTTP 协议缺少 RDS 支持：
  - Listener 内嵌路由
  - 修改内嵌路由后，Envoy 会重建 listener，导致业务出现短暂中断。



# MetaProtocol: Service Mesh 通用七层协议框架

大部分七层协议的路由、熔断、负载均衡等能力的实现逻辑是类似的，没有必要每个协议都全部从头实现，重复造轮子。

- MetaProtocol 框架在 Service Mesh 提供七层协议的通用能力：
  - 数据面：MetaProtocol Proxy 实现 RDS、负载均衡、熔断等公共的基础能力
  - 控制面：Aeraki + Istio 提供控制面管理，实现按请求 header 路由、灰度发布、地域感知LB、流量镜像等高级流量管理能力。
- 基于 MetaProtocol 框架扩展开发，只需要实现 encode、decode 的少量接口即可在 Istio 中支持一个新的七层协议
- 为七层协议如 Dubbo、Thrift 等等添加 RDS 能力

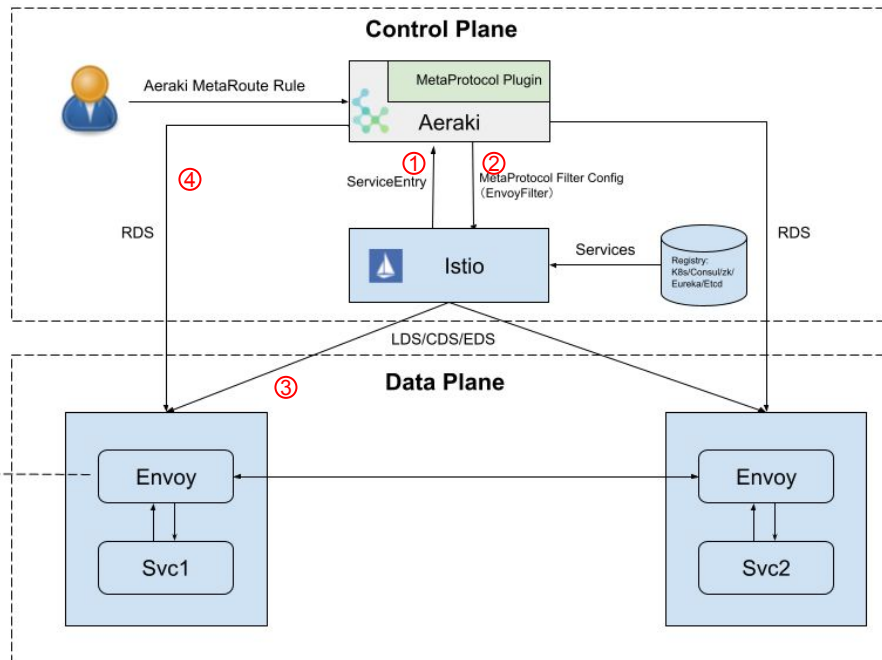
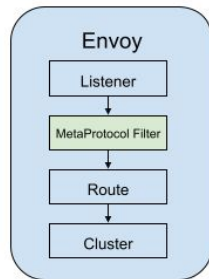


# MetaProtocol: 控制面

通过 Aeraki MetaProtocol Plugin 实现控制面的流量管理规则下发

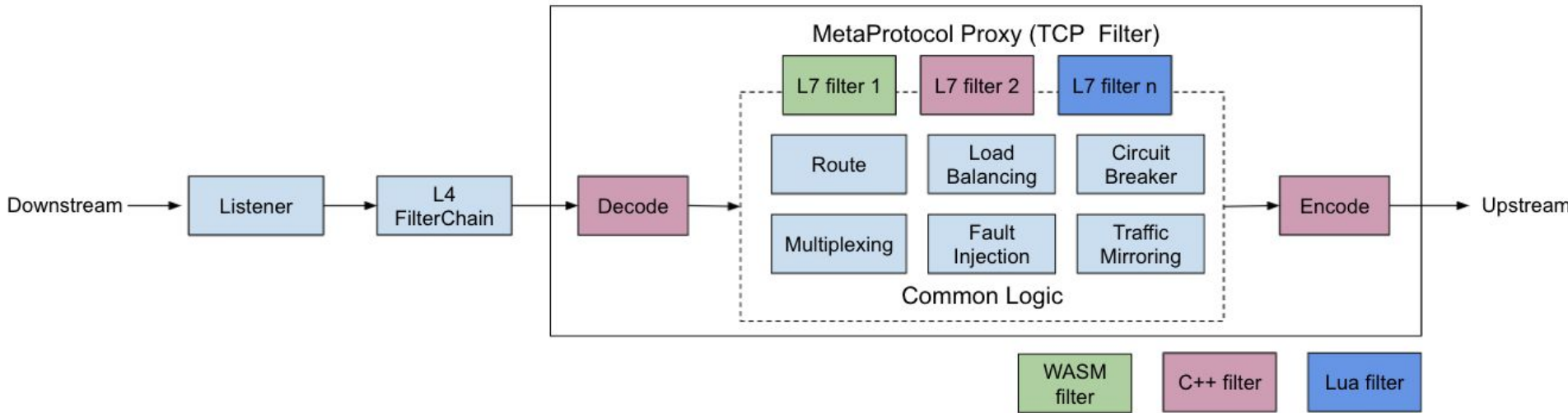
:

1. Aeraki 从 Istio 中获取 ServiceEntry, 通过端口命名判断协议类型 (如 tcp-metaprotocol-thrift)
2. 为 MetaProtocol 服务生成数据面所需的 Filter 配置, Filter 配置中将 RDS 指向 Aeraki
3. Istio 下发 LDS (Patch)/CDS/EDS 给 Envoy
4. Aeraki 根据缺省路由或者用户设置的路由规则下发 RDS 给 Envoy



# MetaProtocol: 数据面

- MetaProtocol Proxy 中实现七层协议的通用逻辑：路由、Header Mutation、负载均衡、断路器、多路复用、流量镜像等。
- 基于 MetaProtocol 实现一个自定义协议时，只需要实现 Decode 和 Encode 扩展点的少量代码 (C++)。
- 提供基于 WASM 和 Lua 的 L7 filter 扩展点。用户可以实现一些灵活的自定义协议处理逻辑，例如认证授权等。





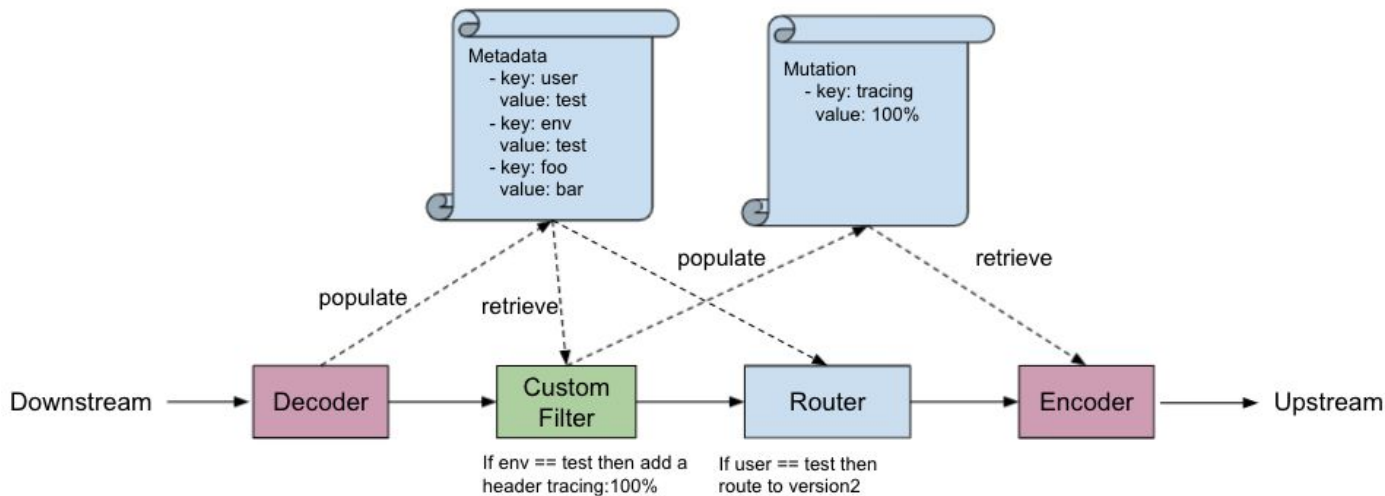
# MetaProtocol: 请求处理路径

处理流程:

1. Decoder 解析 Downstream 请求, 填充 Metadata
2. L7 filter 从 Metadata 获取所需的数据, 进行请求方向的业务处理
3. L7 filter 将需要修改的数据放入 Mutation 结构中
4. Router 根据 RDS 配置的路由规则选择 Upstream Cluster
5. Encoder 根据 Mutation 结构封包
6. 将请求发送给 Upstream

L7 filter 共享数据结构:

- Metadata: decode 时填充的 key:value 键值对, 用于 L7 filter 的处理逻辑中
- Mutation: L7 filter 填充的 key:value 键值对, 用于 encode 时修改请求数据包



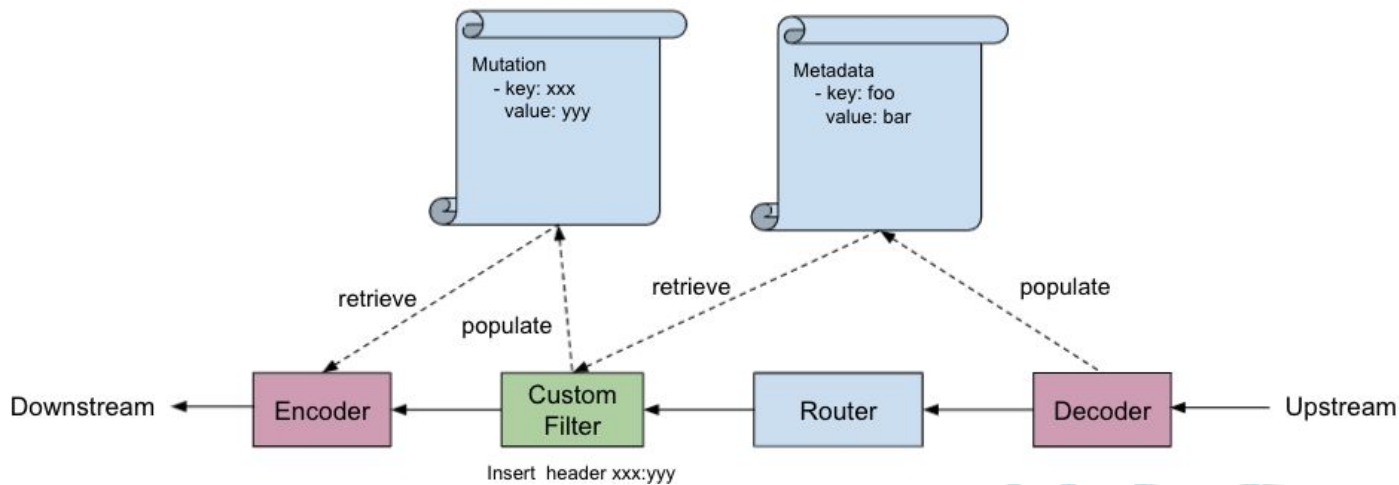
# MetaProtocol: 响应处理路径

处理流程:

1. Decoder 解析 Upstream 的响应, 填充 Metadata
2. Router 根据 connection/stream 对应关系找到响应的 Downstream 连接
3. L7 filter 从 Metadata 获取所需的数据, 进行响应方向的业务处理
4. L7 filter 将需要修改的数据放入 Mutation 结构中
5. Encoder 根据 Mutation 结构封包
6. 将响应发送到 Downstream

L7 filter 共享数据结构:

- Metadata: decode 时填充的 key:value 键值对, 用于 L7 filter 的处理逻辑中
- Mutation: L7 filter 填充的 key:value 键值对, 用于 encode 时修改响应数据包



# MetaProtocol: 流量管理示例 (Canary + Header Mutation)

- 路由规则协议无关：七层协议名是路由规则中的字段值，而不是字段名称
- 采用通用的 key:value 键值对来配置路由匹配条件

```
apiVersion: networking.aeraki.io/v1alpha1
kind: VirtualService
metadata:
  name: dubbo-demo-route-canary
spec:
  host:
  - org.apache.dubbo.samples.basic.api.demoservice
  protocol: dubbo
  Route:
  - name: "v2-route"
    match:
    - property:
      key: user
      value:
        exact: "test"
    headers:
      request:
        set:
          tracing: "100"
    destination:
      host: org.apache.dubbo.samples.basic.api.demoservice
      subset: v2
  - name: "v1-route"
    destination:
      host: org.apache.dubbo.samples.basic.api.demoservice
      subset: v1
```



# Aeraki 后续开源计划

- Istio 增强工具集
  - 协议扩展: Dubbo、Thrift、Redis、MetaProtocol
  - 性能优化: LazyXDS
  - 注册表对接: dubbo2istio、consul、Eureka
  - ...
- 独立组件、非侵入、厂商中立
- 助力 Istio 服务网格产品化



+



**Thank you!**

