腾讯云 知乎

How to Manage Any Layer-7 Traffic in an Istio Service Mesh?

Huabing Zhao@Tencent Cloud, Yang Tang@Zhihu

IstioCon

#IstioCon

Huabing Zhao

Software Engineer @ Tencent Cloud

- @zhaohuabing
- @zhaohuabing
- @zhaohuabing
- @zhaohuabing
- https://zhaohuabing.com

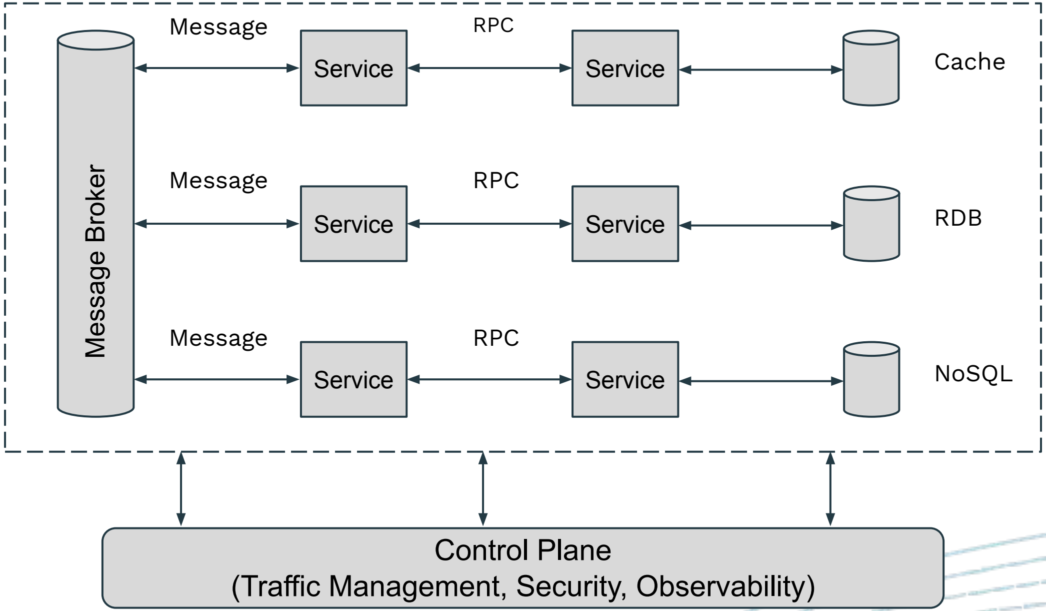
Agenda

- ❑ The Status Quo of Istio Traffic Management
- ❑ Possible Ways to Extend Istio’s Traffic Management Capability
- ❑ Aeraki - Manage Any Layer-7 Traffic in an Istio Service Mesh
- ❑ Demo - Thrift Traffic Splitting
- ❑ Aeraki Use Cases
  - ❑ Dev/Prod parity
  - ❑ More Security
  - ❑ Testing heterogeneous databases
  - ❑ Fault injection for other traffic

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

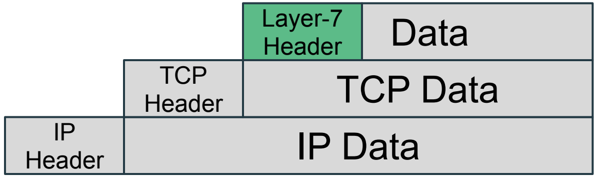
- **RPC**: HTTP, gRPC, Thrift, Dubbo, Proprietary RPC Protocol ...
- **Messaging**: Kafka, RabbitMQ ...
- **Cache**: Redis, Memcached ...
- **Database**: mySQL, PostgreSQL, MongoDB ...
- **Other Layer-7 Protocols**: ...



What Do We Expect From a Service Mesh?

Layer-7 Traffic Management

- Routing based on layer-7 header
  - Load balancing at requet level
  - HTTP host/header/url/method,
  - Thrift service name/method name
  - ...
- 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
  - ...



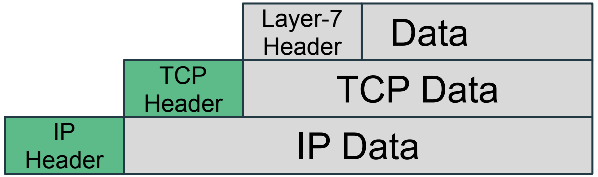
What Do We Get From Istio?

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

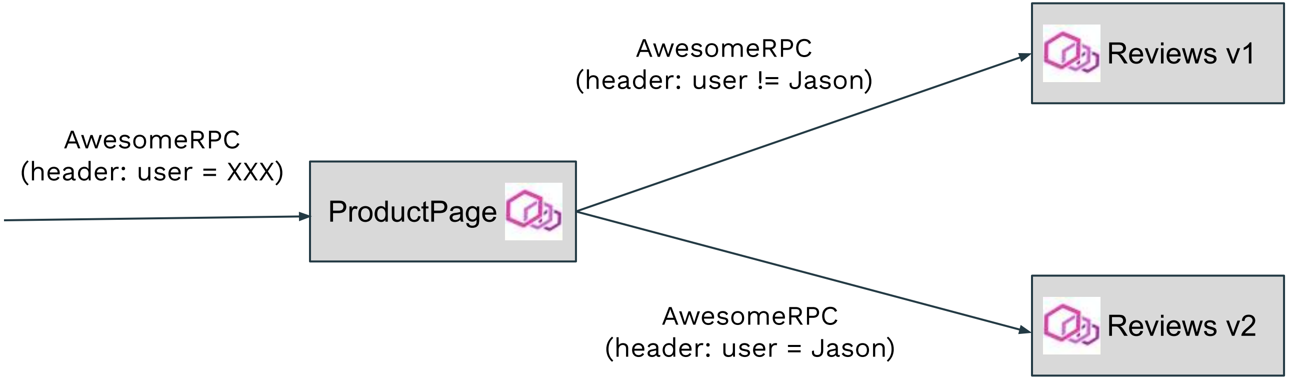
- Routing based on headers under layer-7
  - IP address
  - TCP Port
  - SNI
- Observability - only TCP metrics
  - TCP sent/received bytes
  - TCP opened/closed connections
- Security
  - Connection level authentication: mTLS
  - Connection level authorization: Identity/Source IP/ Dest Port
  - Request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



BookInfo Application - AwesomeRPC

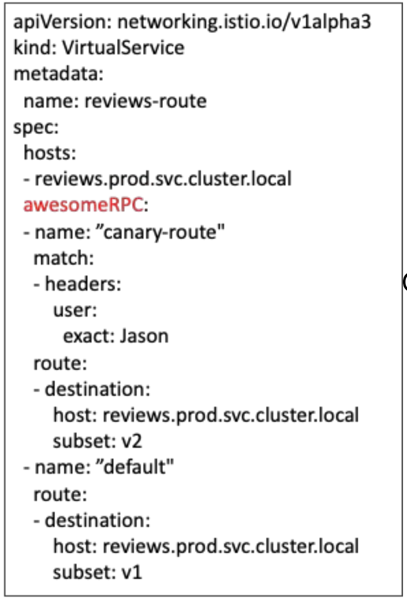
Let's say that we're running a bookinfo application in an Istio service mesh, but the inter-services communication are done by AwesomePRC, our own RPC protocol, instead of HTTP.

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

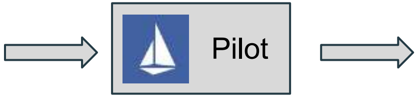


How to Manage AwesomeRPC Traffic in Istio?

Istio Config



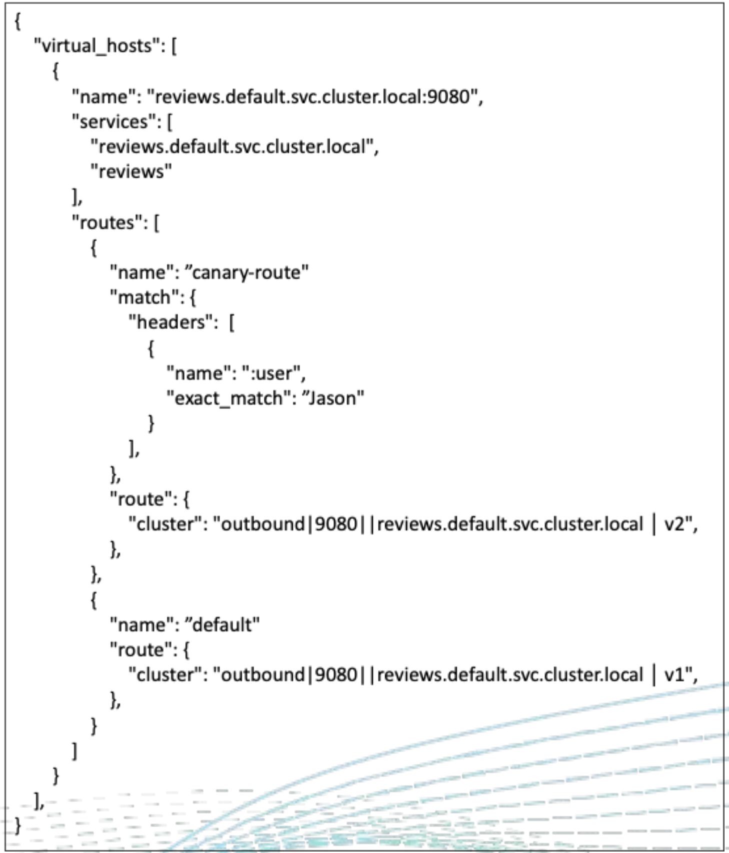
```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews-route
spec:
  hosts:
  - reviews.prod.svc.cluster.local
  awesomeRPC:
  - name: ”canary-route"
    match:
    - headers:
        user:
          exact: Jason
    route:
    - destination:
        host: reviews.prod.svc.cluster.local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



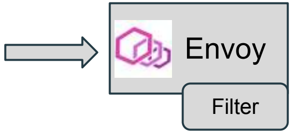
Code changes at the Pilot side:

- Add AwesomeRPC support in VirtualService API
- Generate xDS 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
- Parsing header
- Routing
- Load balancing
- Circuit breaker
- Fault injection
- Stats
- ...

Pros:

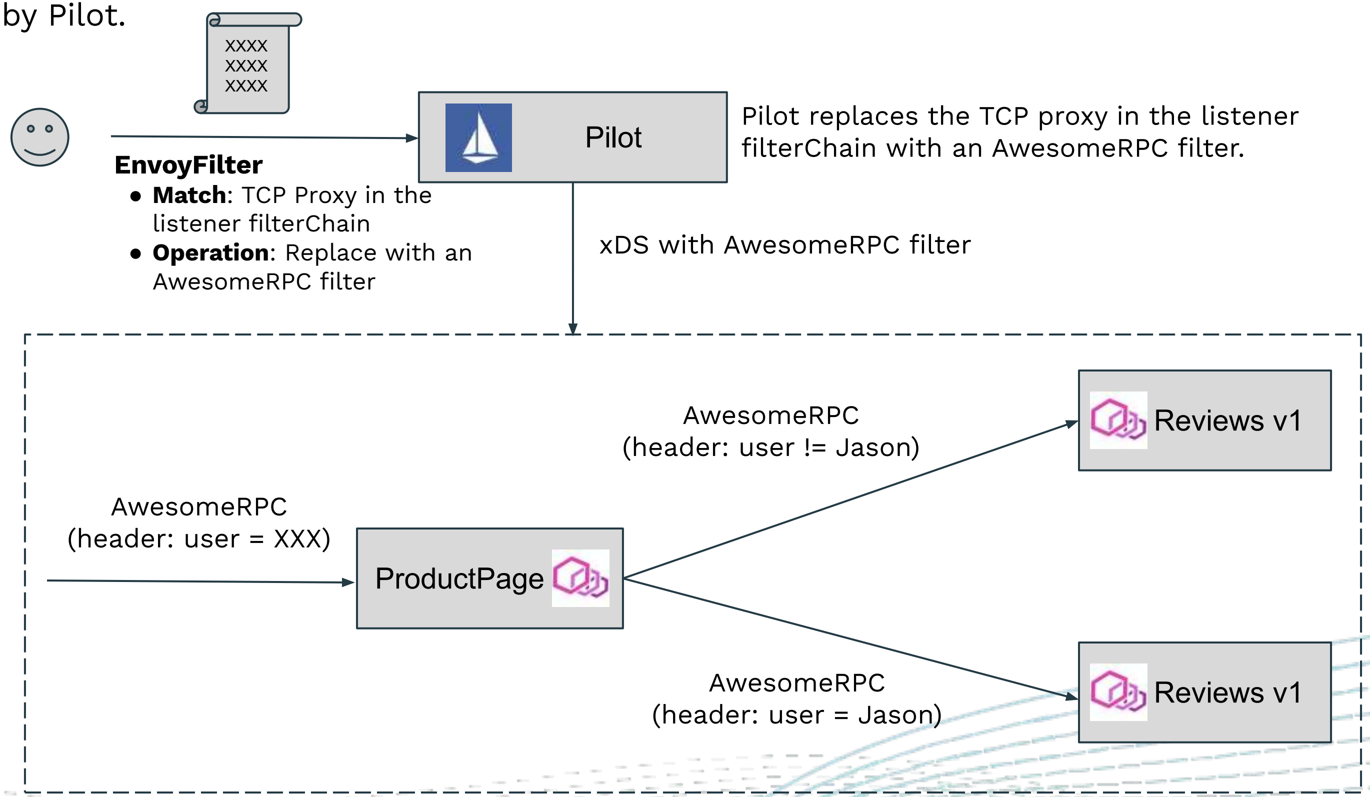
- It’s relatively easy to add support for a new protocol to the control plane, if envoy filter is already there

Cons:

- You have to maintain a fork of Istio, which makes upgrade 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.



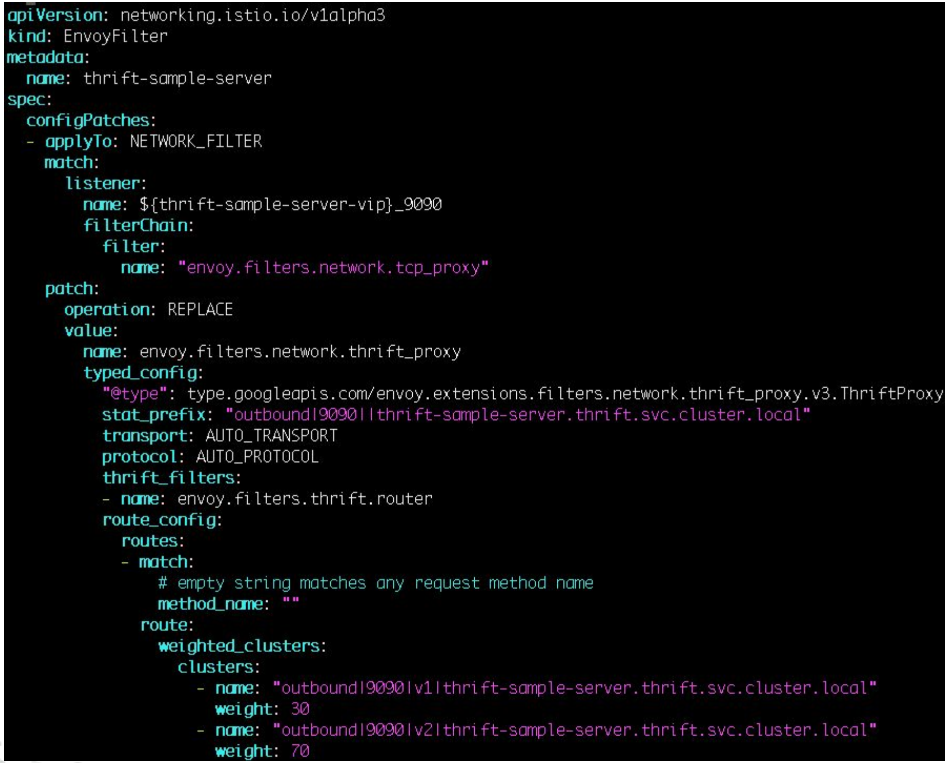
EnvoyFilter

- **Match**: TCP Proxy in the listener filterChain
- **Operation**: Replace with an AwesomeRPC filter

Pilot replaces the TCP proxy in the listener filterChain with an AwesomeRPC filter.

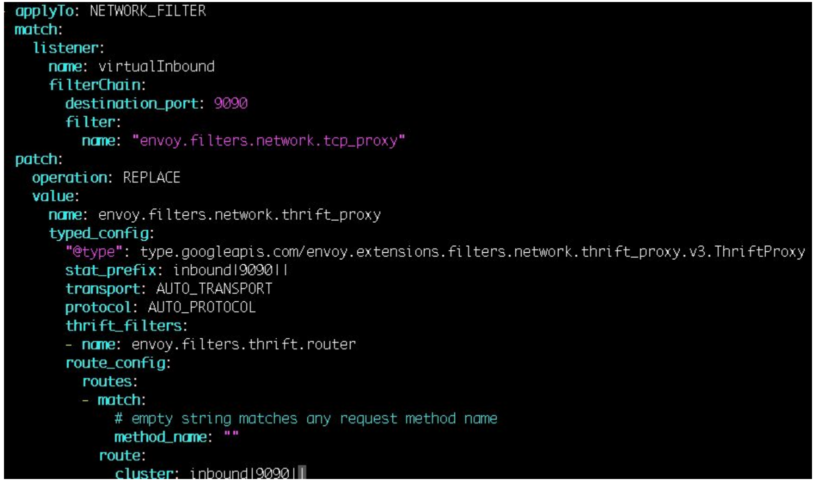
EnvoyFilter Example - Thrift Traffic Splitting

Replace TCP proxy in the outbound listener

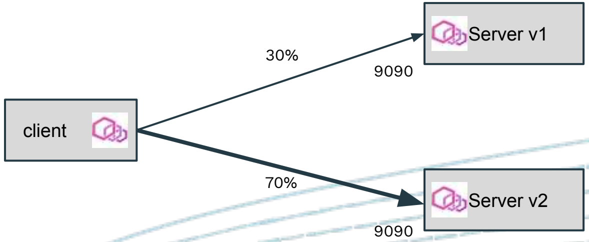


```yaml
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: thrift-sample-server
spec:
  configPatches:
  - applyTo: NETWORK_FILTER
    match:
      listener:
        name: ${thrift-sample-server-vip}_9090
        filterChain:
          filter:
            name: "envoy.filters.network.tcp_proxy"
    patch:
      operation: REPLACE
      value:
        name: envoy.filters.network.thrift_proxy
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.thrift_proxy.v3.ThriftProxy
          stat_prefix: "outbound|9090||thrift-sample-server.thrift.svc.cluster.local"
          transport: AUTO_TRANSPORT
          protocol: AUTO_PROTOCOL
          thrift_filters:
          - name: envoy.filters.thrift.router
          route_config:
            routes:
            - match:
                # empty string matches any request method name
                method_name: ""
              route:
                weighted_clusters:
                  clusters:
                    - name: "outbound|9090|v1|thrift-sample-server.thrift.svc.cluster.local"
                      weight: 30
                    - name: "outbound|9090|v2|thrift-sample-server.thrift.svc.cluster.local"
                      weight: 70
```

Replace TCP proxy in the inbound listener



```yaml
applyTo: NETWORK_FILTER
match:
  listener:
    name: virtualInbound
    filterChain:
      destination_port: 9090
      filter:
        name: "envoy.filters.network.tcp_proxy"
patch:
  operation: REPLACE
  value:
    name: envoy.filters.network.thrift_proxy
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.network.thrift_proxy.v3.ThriftProxy
      stat_prefix: inbound|9090||
      transport: AUTO_TRANSPORT
      protocol: AUTO_PROTOCOL
      thrift_filters:
      - name: envoy.filters.thrift.router
      route_config:
        routes:
        - match:
            # empty string matches any request method name
            method_name: ""
          route:
            cluster: inbound|9090||
```

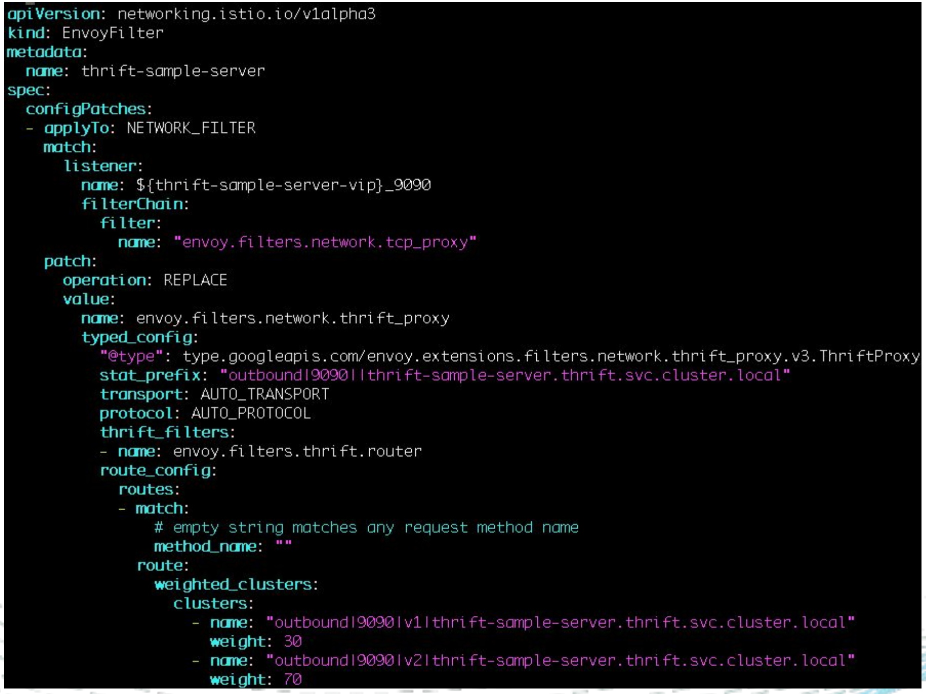


#IstioCon

EnvoyFilter is Powerful, But ...

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

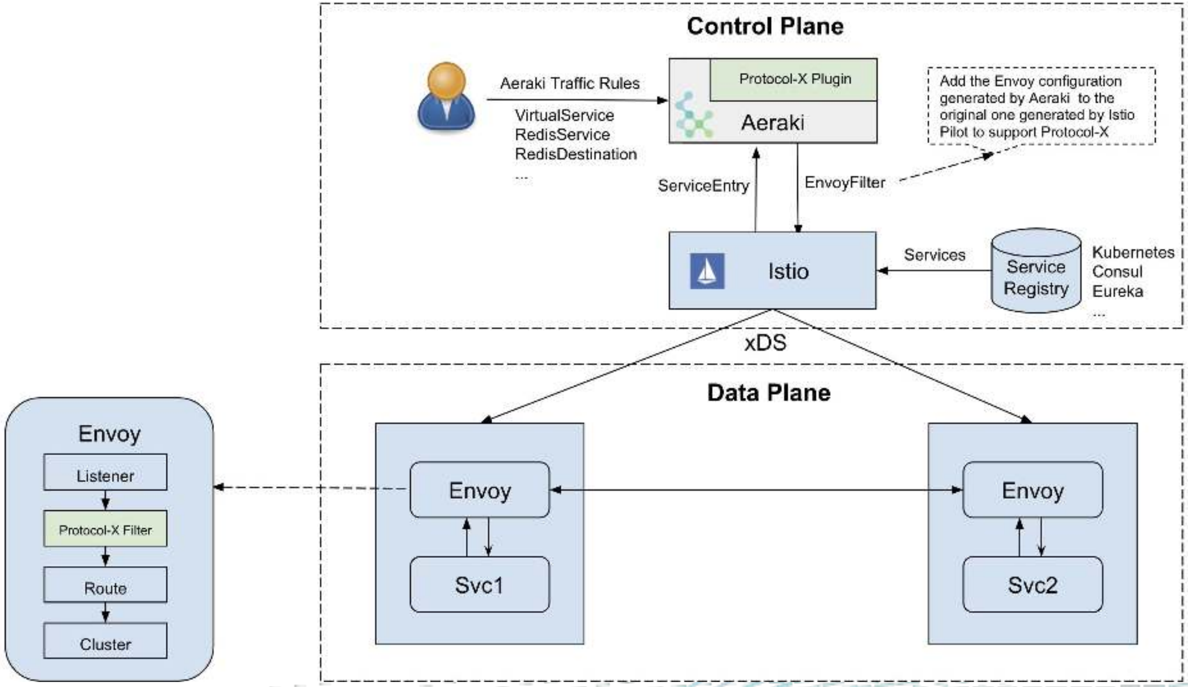
- It exposes low-level Envoy configurations to operation
- It depends on the structure/name convention of the generated xDS by Pilot
- It depends on some cluster-specific information such as service cluster IP
- We need to manually create tons of EnvoyFilter, one for each of the services



```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: thrift-sample-server
spec:
  configPatches:
  - applyTo: NETWORK_FILTER
    match:
      listener:
        name: ${thrift-sample-server-vip}_9090
        filterChain:
          filter:
            name: "envoy.filters.network.tcp_proxy"
    patch:
      operation: REPLACE
      value:
        name: envoy.filters.network.thrift_proxy
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.thrift_proxy.v3.ThriftProxy
          stat_prefix: "outbound|9090||thrift-sample-server.thrift.svc.cluster.local"
          transport: AUTO_TRANSPORT
          protocol: AUTO_PROTOCOL
          thrift_filters:
          - name: envoy.filters.thrift.router
          route_config:
            routes:
            - match:
                # empty string matches any request method name
                method_name: ""
              route:
                weighted_clusters:
                  clusters:
                    - name: "outbound|9090|v1|thrift-sample-server.thrift.svc.cluster.local"
                      weight: 30
                    - name: "outbound|9090|v2|thrift-sample-server.thrift.svc.cluster.local"
                      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



#IstioCon

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:

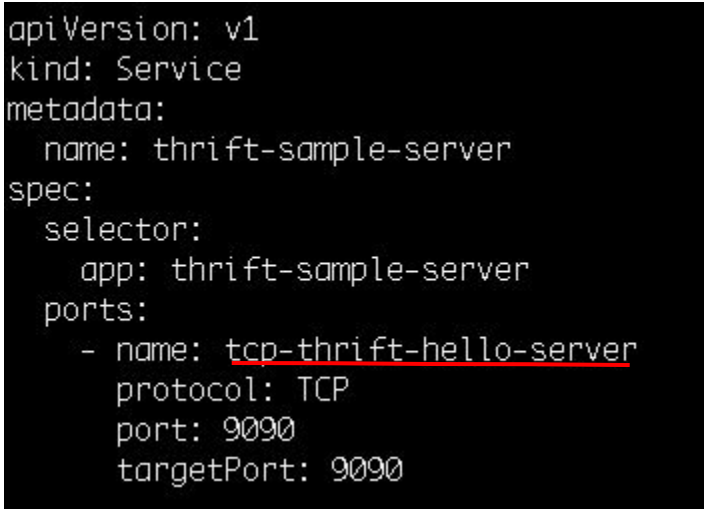
- PRC: Thrift, Dubbo, tRPC
- Others: Redis, Kafka, Zookeeper,
- More protocols are on the way ...

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

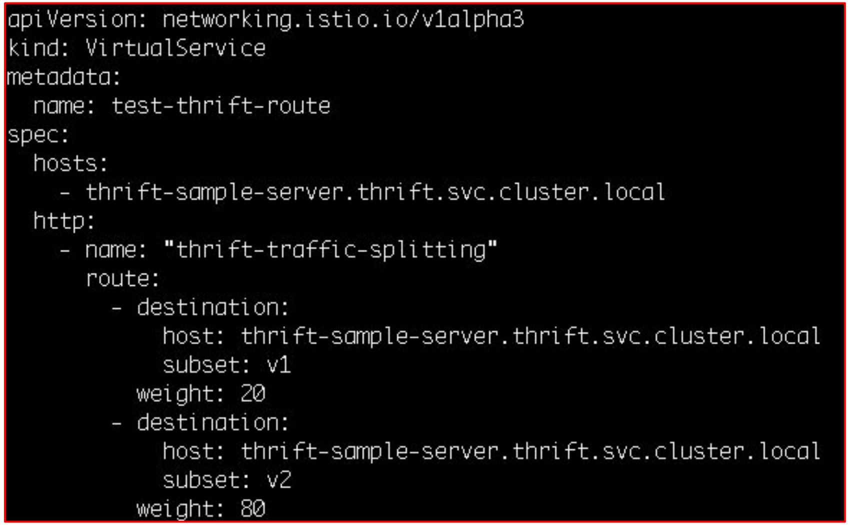
Aeraki Configuration Example: Thrift

Service definition



```
apiVersion: v1
kind: Service
metadata:
  name: thrift-sample-server
spec:
  selector:
    app: thrift-sample-server
  ports:
    - name: tcp-thrift-hello-server
      protocol: TCP
      port: 9090
      targetPort: 9090
```

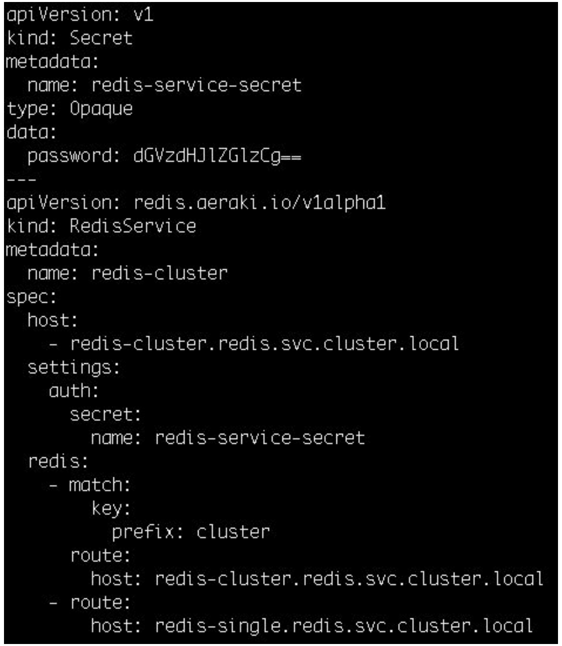
Traffic rules



```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: test-thrift-route
spec:
  hosts:
    - thrift-sample-server.thrift.svc.cluster.local
  http:
    - name: "thrift-traffic-splitting"
      route:
        - destination:
            host: thrift-sample-server.thrift.svc.cluster.local
            subset: v1
          weight: 20
        - destination:
            host: thrift-sample-server.thrift.svc.cluster.local
            subset: v2
          weight: 80
```

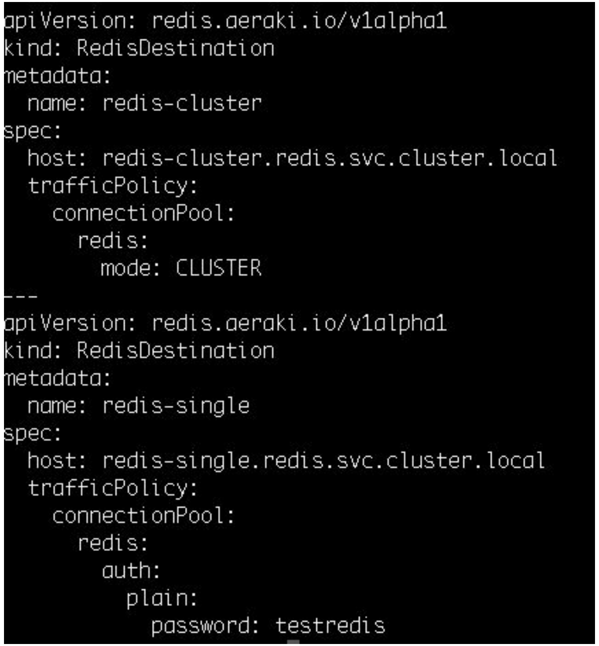
Aeraki Configuration Example: Redis

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: Thrift Traffic Management

Live Demo: kiali Dashboard

Live Demo: Service Metrics: Grafana

Live Demo: Service Metrics: Prometheus

Would like to give it a try? It's just as simple as two commands:

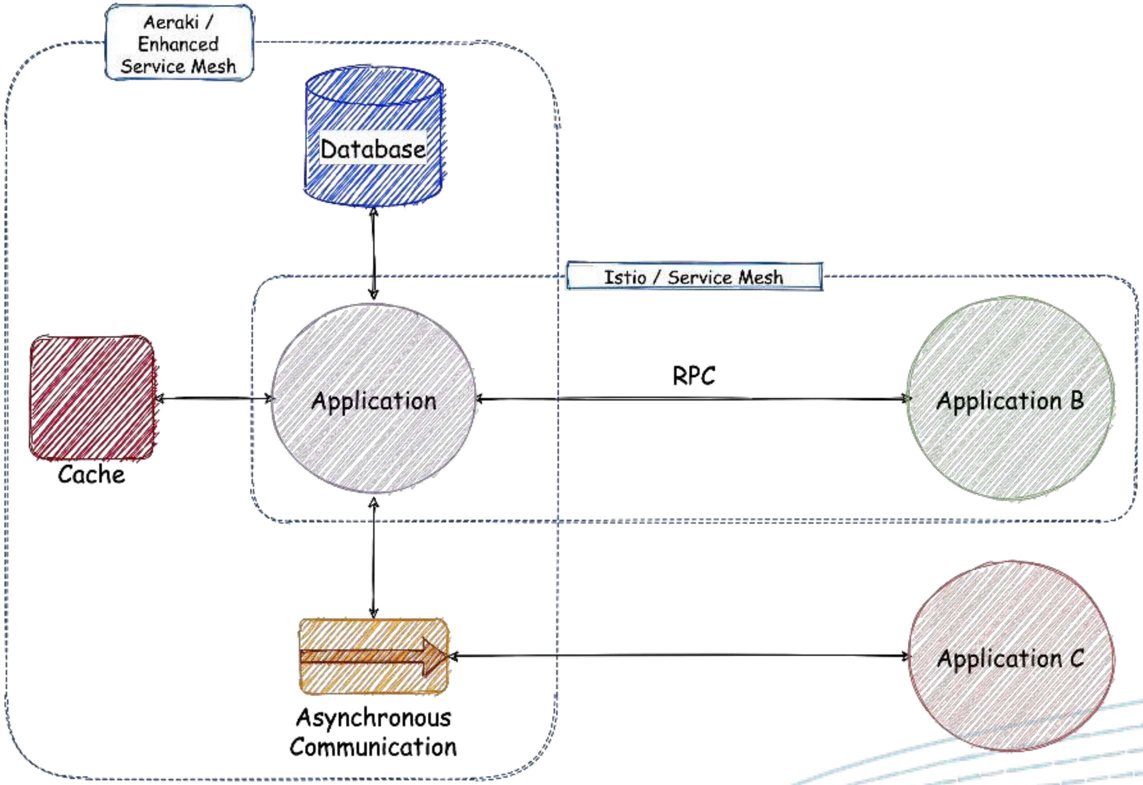
```
git clone https://github.com/aeraki-framework/aeraki.git
aeraki/demo/install-demo.sh
```

Yang Tang

Software Engineer @ zhihu

@cocotyty

Use Aeraki to enhance the Service Mesh

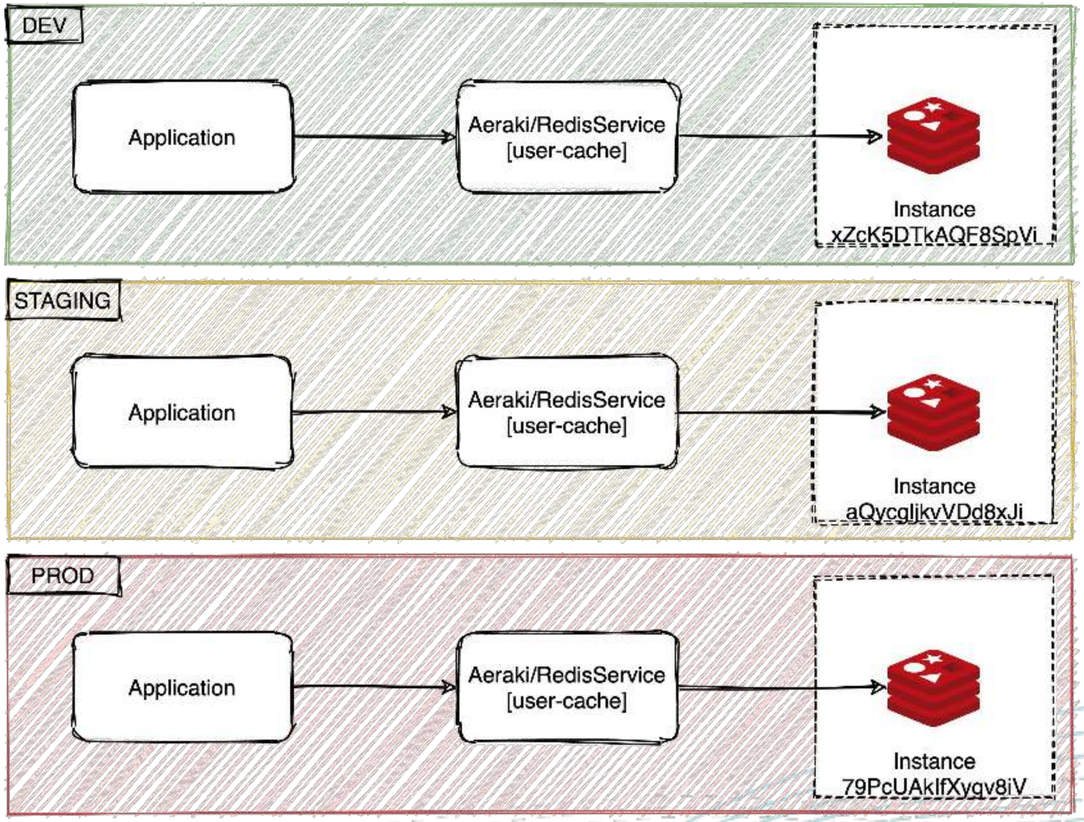


DEV/PROD Parity

Continuous integration / deploy your software without changing any code or configuration.

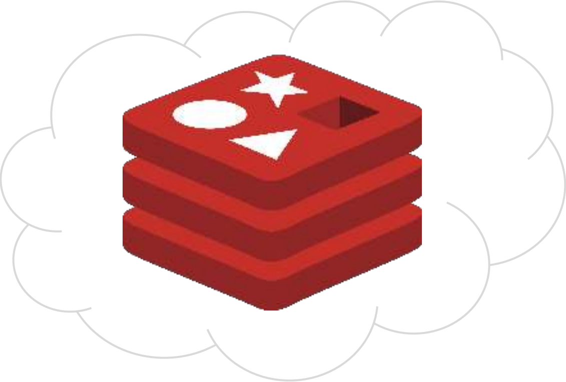
- Use the same DSN to access database between different environments.
- Use the same URL to access cache between different environments.
- Use the same X to access Y between different environments.

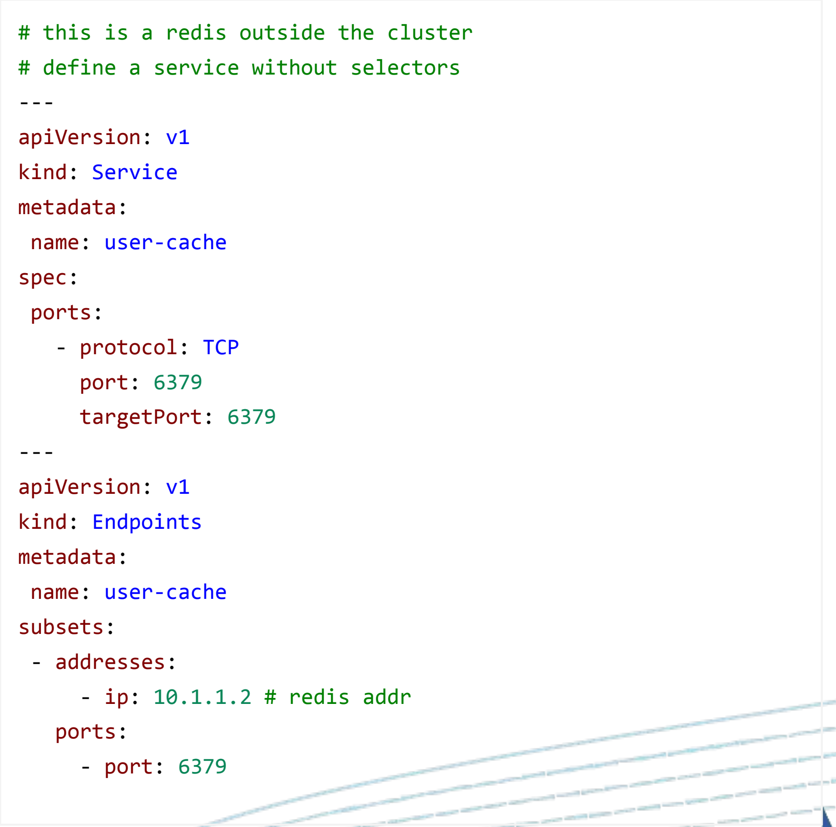
Example: Redis



Example: Redis

Eliminate the differences Redis hosts used in different environments by creating a no-selector service.

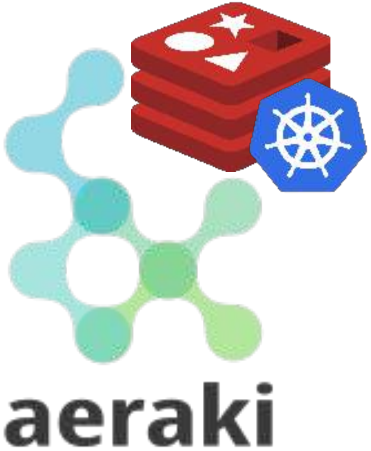


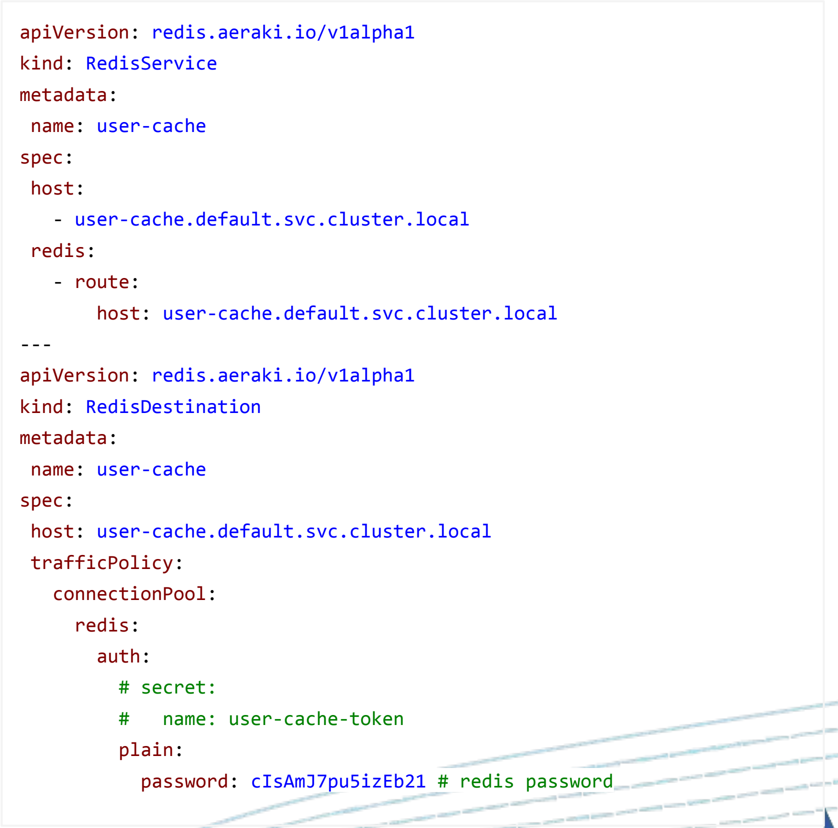


```yaml
# this is a redis outside the cluster
# define a service without selectors
---
apiVersion: v1
kind: Service
metadata:
 name: user-cache
spec:
 ports:
   - protocol: TCP
     port: 6379
     targetPort: 6379
---
apiVersion: v1
kind: Endpoints
metadata:
 name: user-cache
subsets:
 - addresses:
     - ip: 10.1.1.2 # redis addr
   ports:
     - port: 6379
```

Example: Redis

Use RedisService and RedisDestination to eliminate the difference between usernames or passwords.

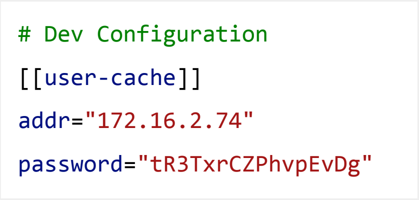




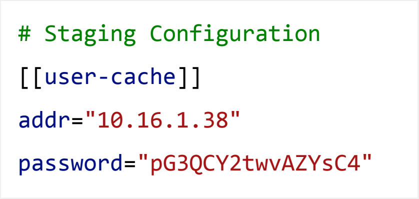
```yaml
apiVersion: redis.aeraki.io/v1alpha1
kind: RedisService
metadata:
 name: user-cache
spec:
 host:
   - user-cache.default.svc.cluster.local
 redis:
   - route:
       host: user-cache.default.svc.cluster.local
---
apiVersion: redis.aeraki.io/v1alpha1
kind: RedisDestination
metadata:
 name: user-cache
spec:
 host: user-cache.default.svc.cluster.local
 trafficPolicy:
   connectionPool:
     redis:
       auth:
         # secret:
         #   name: user-cache-token
         plain:
           password: cIsAmJ7pu5izEb21 # redis password
```

#IstioCon

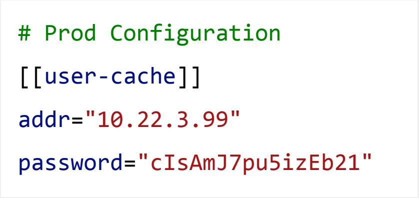
Example: Redis



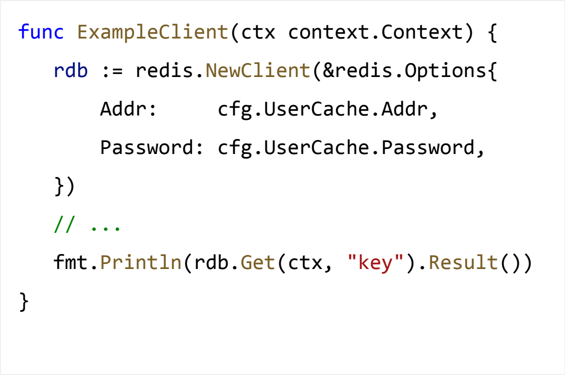
```
# Dev Configuration
[[user-cache]]
addr="172.16.2.74"
password="tR3TxrCZPhvpEvDg"
```



```
# Staging Configuration
[[user-cache]]
addr="10.16.1.38"
password="pG3QCY2twvAZYsC4"
```

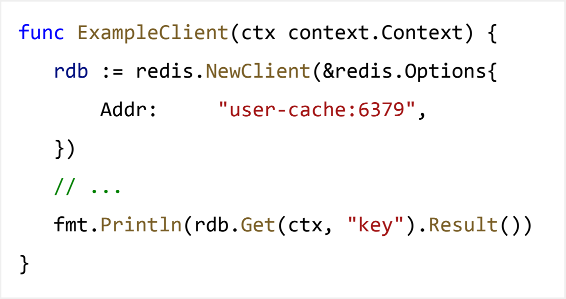


```
# Prod Configuration
[[user-cache]]
addr="10.22.3.99"
password="cIsAmJ7pu5izEb21"
```



```
func ExampleClient(ctx context.Context) {
    rdb := redis.NewClient(&redis.Options{
        Addr:     cfg.UserCache.Addr,
        Password: cfg.UserCache.Password,
    })
    // ...
    fmt.Println(rdb.Get(ctx, "key").Result())
}
```

BEFORE



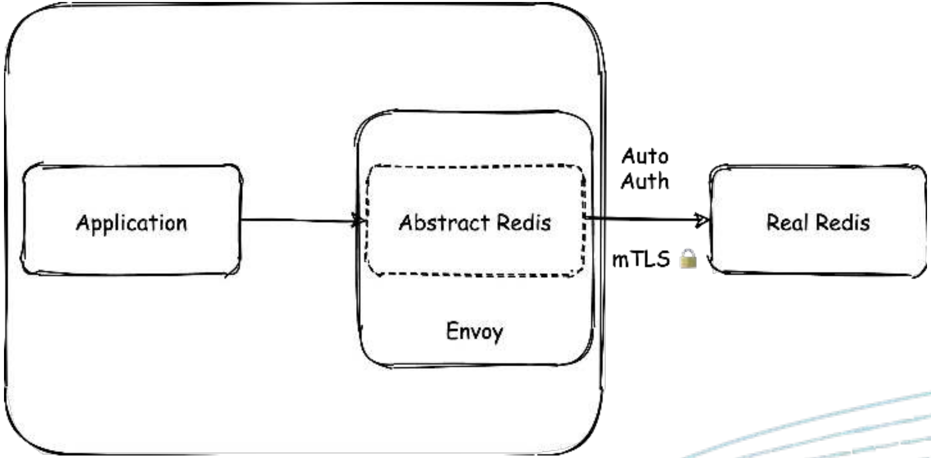
```
func ExampleClient(ctx context.Context) {
    rdb := redis.NewClient(&redis.Options{
        Addr:     "user-cache:6379",
    })
    // ...
    fmt.Println(rdb.Get(ctx, "key").Result())
}
```

AFTER

More security

Aeraki gives your application these protections:

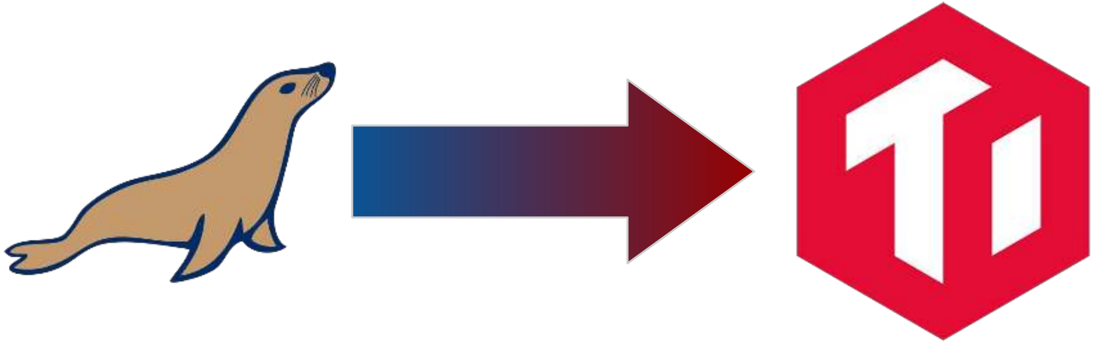
1. Manage authorization for other system, like databases.
2. Upgrade other traffic to mTLS
3. Avoid authentication in your code



Helpful for using heterogeneous databases

MySQL Protocol compatible:

- MariaDB
- TiDB
- Oceanbase
- Amazon Aurora
- KingShard
- ...

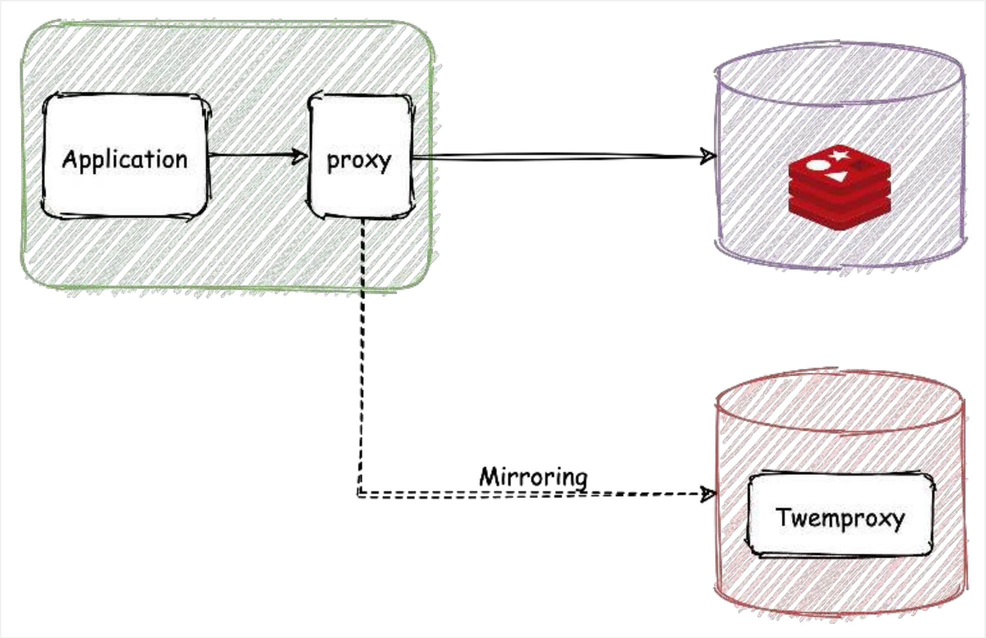


Redis Protocol compatible:

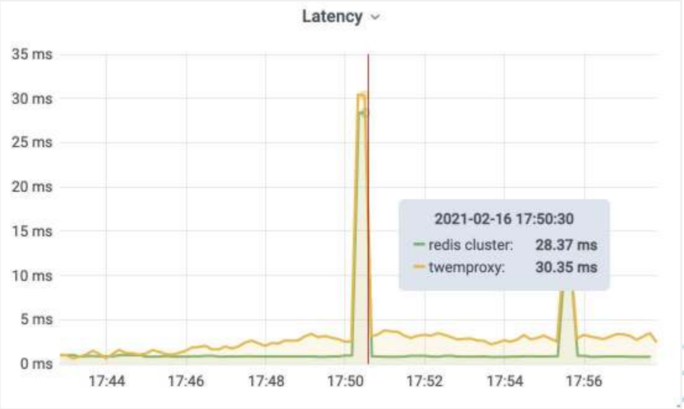
- Codis
- Tendis
- Pika
- Twemproxy
- ...

Performance
Compatibility
Easy Migration

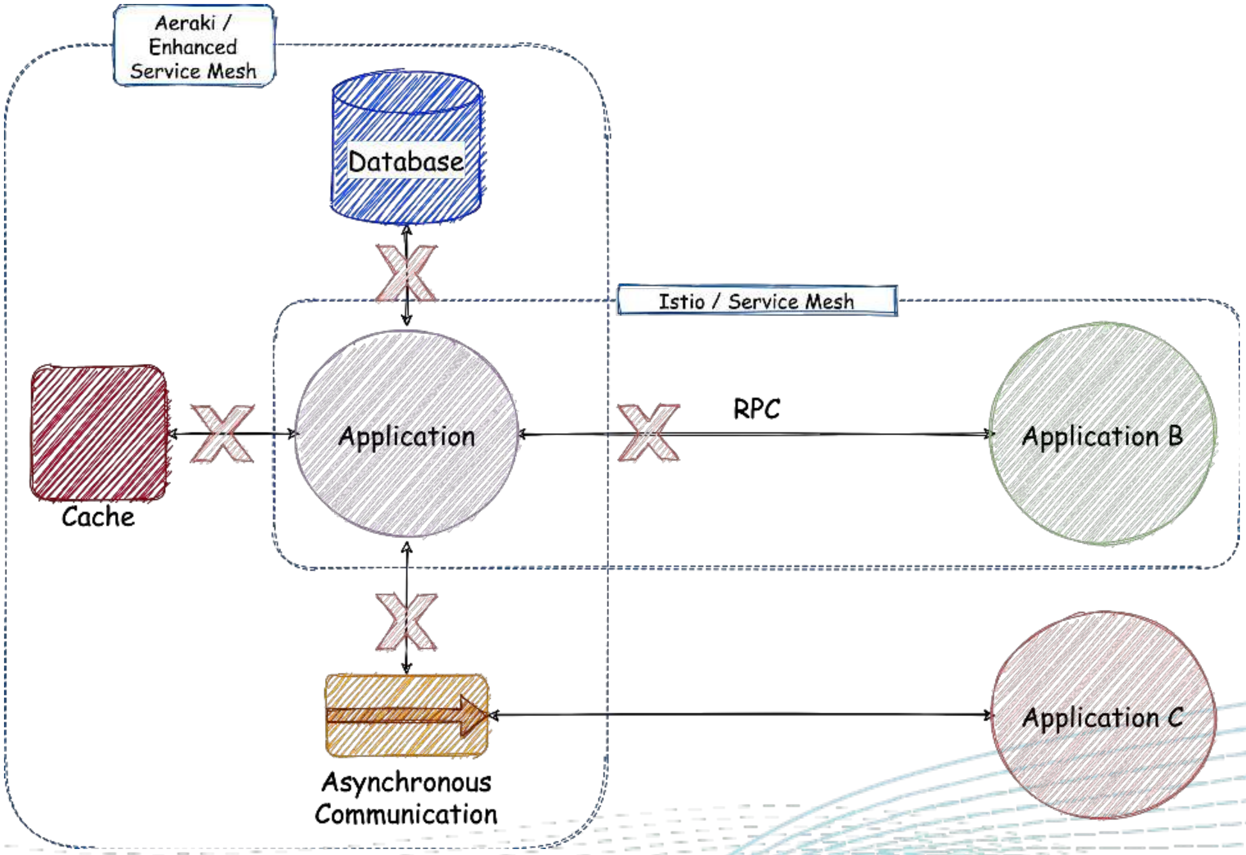
Helpful for using heterogeneous databases



Compare latency between Redis Cluster and Twemproxy



Fault injection for other traffic



Aeraki

Thank you!