

Alibaba Cloud

Alibaba Cloud Service Mesh Best Practices

Document Version: 20220712

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.
2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.
3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.
4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).
5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.
6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
 Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	 Danger: Resetting will result in the loss of user configuration data.
 Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	 Warning: Restarting will cause business interruption. About 10 minutes are required to restart an instance.
 Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	 Notice: If the weight is set to 0, the server no longer receives new requests.
 Note	A note indicates supplemental instructions, best practices, tips, and other content.	 Note: You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click Settings > Network > Set network type .
Bold	Bold formatting is used for buttons, menus, page names, and other UI elements.	Click OK .
Courier font	Courier font is used for commands	Run the <code>cd /d C:/window</code> command to enter the Windows system folder.
<i>Italic</i>	Italic formatting is used for parameters and variables.	<code>bae log list --instanceid</code> <i>Instance_ID</i>
[] or [a b]	This format is used for an optional value, where only one item can be selected.	<code>ipconfig [-all -t]</code>
{ } or {a b}	This format is used for a required value, where only one item can be selected.	<code>switch {active stand}</code>

Table of Contents

1.Workloads -----	05
1.1. Use an ingress gateway to access a gRPC service in an AS...-----	05
1.2. Implement auto scaling for workloads by using ASM metri...-----	12
2.Traffic Management -----	20
2.1. Use ASM to deploy an application in blue-green release m...-----	20
2.2. Use ASM and Wasm to implement end-to-end A/B testing...-----	25
2.3. Use ASM and KubeVela to implement a canary release -----	38
2.4. Use an ASM instance of a commercial edition to implemen...-----	46
3.Security -----	64
3.1. Implement CORS in ASM -----	64
3.2. Enable Multi-Buffer for TLS acceleration -----	66
4.Use gRPC in ASM -----	70
4.1. Design principle of the gRPC practice -----	70
4.2. Implement the communication models of gRPC -----	73
4.3. Implement load balancing among gRPC servers -----	86
4.4. Shape traffic to gRPC servers -----	93
4.5. Redirect traffic for gRPC-based applications -----	98
5.Use Flagger in ASM -----	104
5.1. Use Mixerless Telemetry to observe ASM instances -----	104
5.2. Use Mixerless Telemetry to scale the pods of an applicatio...-----	108
5.3. Use Mixerless Telemetry to implement a canary release -----	114
6.Authorize and control services in namespaces -----	121
6.1. Use an authorization policy to control service access across...-----	121
6.2. Use an authorization policy to control access traffic from ...-----	126

1. Workloads

1.1. Use an ingress gateway to access a gRPC service in an ASM instance over HTTP

Ingress gateways in an Alibaba Cloud Service Mesh (ASM) instance support protocol transcoding. This feature allows you to send HTTP requests that use the JSON data format from your browser or client to access gRPC services in an ASM instance. This topic describes how to use an ingress gateway to access a gRPC service in an ASM instance over HTTP.

Prerequisites

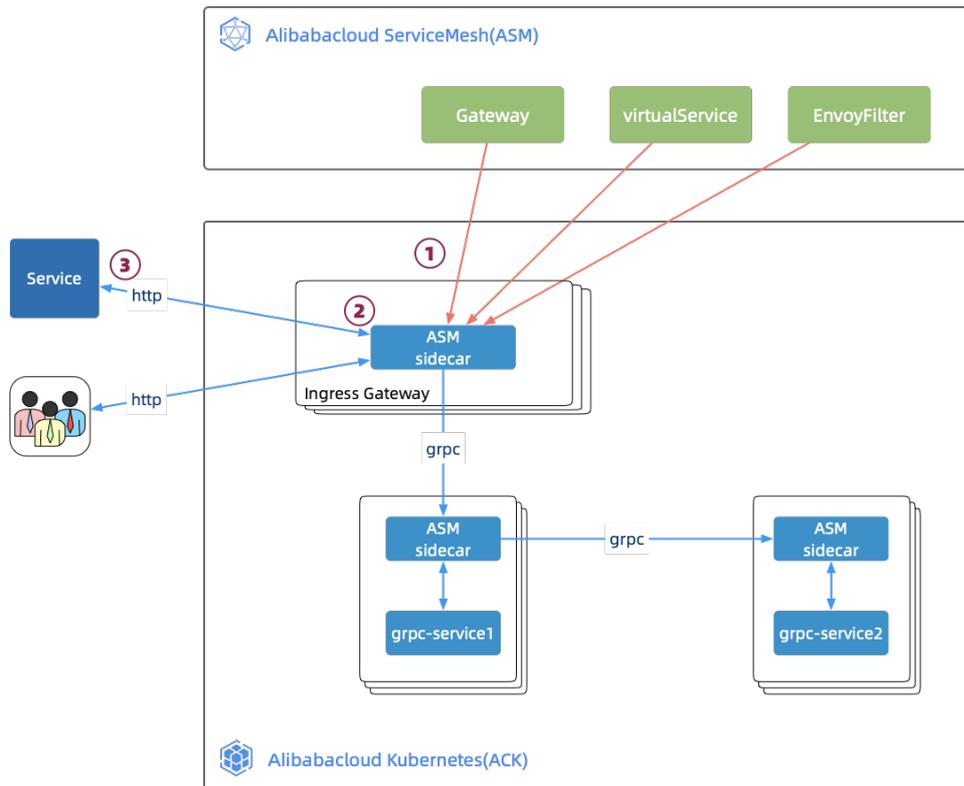
- The `grpc-transcoder` tool is installed. The tool is used to automatically generate Envoy filters. For more information, visit the [grpc-transcoder](#) page on GitHub.
- Protocol Buffers is installed. For more information, visit the [Protocol Buffers](#) page on GitHub.

Context

Envoy is a proxy service that composes the data plane of an ASM instance. Envoy contains various built-in HTTP filter extensions, including the gRPC-JSON transcoder. To enable the gRPC-JSON transcoder, Envoy defines relevant filter protocols. For more information, see [gRPC-JSON transcoder](#). Accordingly, the control plane of an ASM instance must define an Envoy filter to declare the specific phase in which the gRPC-JSON transcoder is enabled. Then, the defined Envoy filter is applied to enable the gRPC-JSON transcoder in the specific phase.

Transcoding process

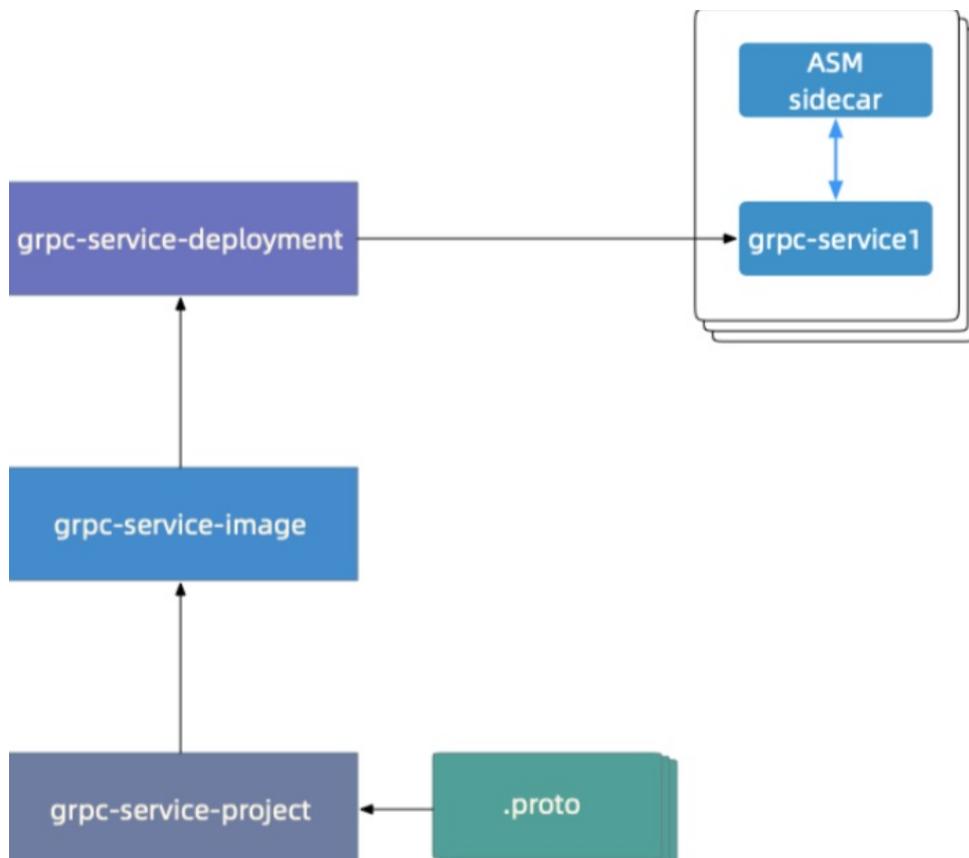
Ingress gateways in an ASM instance can transcode HTTP/JSON to gRPC. The following figure shows the transcoding process.



No.	Description
1	The control plane of an ASM instance applies the following configurations to an ingress gateway: an Envoy filter that is used for gRPC transcoding, and an Istio gateway and a virtual service that are used to configure rules to route traffic to a gRPC service port. After the ingress gateway receives the configurations, the ingress gateway immediately loads the configurations for the configurations to take effect.
2	After an HTTP request is received from your browser or client, the ingress gateway matches routing rules. Then, the ingress gateway transcodes the HTTP request to a gRPC request and sends the request to the destination gRPC service in the ASM instance.
3	After a gRPC response is received from the backend service, the ingress gateway transcodes the gRPC response to an HTTP response and returns the HTTP response to you.

Step 1: Add a transcoding declaration

To create a gRPC service, you must first define a *.proto* file in the Protocol Buffers format. The gRPC service project encapsulates a gRPC API. You must build an image, compile a Deployment, and then deploy the gRPC service as a pod to a Container Service for Kubernetes (ACK) cluster by using an ASM instance.



To enable transcoding from HTTP/JSON to gRPC, you must add the following transcoding declaration to the method definition in the *.proto* file:

```
option(google.api.http) = {
  get: "/v1/talk/{data}/{meta}"
};
```

The *.proto* file in the `hello-servicemesh-grpc` sample project is used as an example. The following code shows the content of the *.proto* file to which a transcoding declaration is added. For more information, visit the [hello-servicemesh-grpc](#) page on GitHub.

```
import "google/api/annotations.proto";
service LandingService {
  //Unary RPC
  rpc talk (TalkRequest) returns (TalkResponse) {
    option(google.api.http) = {
      get: "/v1/talk/{data}/{meta}"
    };
  }
  ...
}
message TalkRequest {
  string data = 1;
  string meta = 2;
}
```

Step 2: Generate a .proto-descriptor file

Run the following Protoc command in Protocol Buffers to generate the *landing.proto-descriptor* file from the *landing.proto* file:

```
# https://github.com/AliyunContainerService/hello-servicemesh-grpc
proto_path={path/to/hello-servicemesh-grpc}/grpc/proto
# https://github.com/googleapis/googleapis/tree/master/
proto_dep_path={path/to/googleapis}
protoc \
  --proto_path=${proto_path} \
  --proto_path=${proto_dep_path} \
  --include_imports \
  --include_source_info \
  --descriptor_set_out=landing.proto-descriptor \
  "${proto_path}"/landing.proto
```

Step 3: Generate a YAML file for creating an Envoy filter

Enter the following code in the command window on your computer to call the gRPC API. Then, the `grpc-transcoder` tool is automatically started to generate a YAML file for creating an Envoy filter.

```
grpc-transcoder \
  --version 1.7 \
  --service_port 9996 \
  --service_name grpc-server-svc \
  --proto_pkg org.feuyeux.grpc \
  --proto_svc LandingService \
  --descriptor landing.proto-descriptor
```

- `version` : the Istio version of the ASM instance.
- `service_port` : the port of the gRPC service.
- `service_name` : the name of the gRPC service.
- `proto_pkg` : the definition of the package name for the *.proto* file of the gRPC service.
- `proto_svc` : the definition of the service name in the *.proto* file of the gRPC service.

- `descriptor` : the path of the *.proto-descriptor* file.

The following content for creating an Envoy filter is automatically generated after you run the preceding code. Copy the following content to the *grpc-transcoder-envoyfilter.yaml* file:

```
#Generated by ASM(http://servicemesh.console.aliyun.com)
#GRPC Transcoder EnvoyFilter[1.7]
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: grpc-transcoder-grpc-server-svc
spec:
  workloadSelector:
    labels:
      app: istio-ingressgateway
  configPatches:
    - applyTo: HTTP_FILTER
      match:
        context: GATEWAY
        listener:
          portNumber: 9996
          filterChain:
            filter:
              name: "envoy.filters.network.http_connection_manager"
              subFilter:
                name: "envoy.filters.http.router"
      proxy:
        proxyVersion: ^1\.7.*
      patch:
        operation: INSERT_BEFORE
        value:
          name: envoy.grpc_json_transcoder
          typed_config:
            '@type': type.googleapis.com/envoy.extensions.filters.http.grpc_json_transcoder
            .v3.GrpcJsonTranscoder
            proto_descriptor_bin: Ct14ChVnb29nbGUvYXBpL2h0dHAucHJ...
            services:
              - org.feueux.grpc.LandingService
            print_options:
              add_whitespace: true
              always_print_primitive_fields: true
              always_print_enums_as_ints: false
              preserve_proto_field_names: false
```

Step 4: Create the Envoy filter in the ASM console

- 1.
- 2.
- 3.
- 4.
5. On the Create page, select a namespace from the **Namespace** drop-down list and copy the content of the *grpc-transcoder-envoyfilter.yaml* file that is edited in [Step 3: Generate a YAML file](#)

for creating an Envoy filter to the code editor. Then, click Create.

Step 5: Verify the Envoy configuration

Run the following commands in sequence to check whether the dynamic Envoy configuration contains the gRPC-JSON transcoder:

```
# Obtain the name of the ingress gateway pod.
ingressgateway_pod=$(kubectl get pod -l app="istio-ingressgateway" -n istio-system -o jsonpath='{.items[0].metadata.name}')
# Obtain the timestamp.
timestamp=$(date "+%Y%m%d-%H%M%S")
# Obtain the dynamic Envoy configuration and save the configuration to the dynamic_listener_s-"$timestamp".json file.
kubectl -n istio-system exec $ingressgateway_pod \
  -c istio-proxy \
  -- curl -s "http://localhost:15000/config_dump?dynamic_listeners" >dynamic_listeners-"$timestamp".json
# Check whether the configuration contains the gRPC-JSON transcoder.
grep -B3 -A7 GrpcJsonTranscoder dynamic_listeners-"$timestamp".json
```

If the following content appears in the output, the dynamic Envoy configuration contains the gRPC-JSON transcoder:

```
{
  "name": "envoy.grpc_json_transcoder",
  "typed_config": {
    "@type": "type.googleapis.com/envoy.extensions.filters.http.grpc_json_transcoder.v3.GrpcJsonTranscoder",
    "services": [
      "org.feuyeux.grpc.LandingService"
    ],
    "print_options": {
      "add_whitespace": true,
      "always_print_primitive_fields": true
    },
    ...
  }
}
```

Step 6: Check whether the gRPC service in the ASM instance can be accessed over HTTP

The *.proto* file defines the request API and response declaration of the gRPC service. When you call the gRPC service by using the request API, the defined response declaration is returned. The following content shows the request API and response declaration that are defined in the *.proto* file:

- Request API that is defined in the *.proto* file:

```
rpc talk (TalkRequest) returns (TalkResponse) {
  option (google.api.http) = {
    get: "/v1/talk/{data}/{meta}"
  };
}
```

- Response declaration that is defined in the *.proto* file:

```
message TalkResponse {
  int32 status = 1;
  repeated TalkResult results = 2;
}
message TalkResult {
  //timestamp
  int64 id = 1;
  //enum
  ResultType type = 2;
  // id:result uuid
  // idx:language index
  // data: hello
  // meta: serverside language
  map<string, string> kv = 3;
}
enum ResultType {
  OK = 0;
  FAIL = 1;
}
```

Run the following commands to use an ingress gateway to call the gRPC service over HTTP:

```
# Obtain the IP address of the ingress gateway.
INGRESS_IP=$(k -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
# Send an HTTP request to access port 9996 of the ingress gateway. The path is /v1/talk/{data}/{meta}.
curl http://$INGRESS_IP:9996/v1/talk/0/java
```

Expected output:

```
{
  "status": 200,
  "results": [
    {
      "id": "699882576081691",
      "type": "OK",
      "kv": {
        "data": "Hello",
        "meta": "JAVA",
        "id": "8c175d5c-d8a3-4197-a7f8-6e3e0ab1fe59",
        "idx": "0"
      }
    }
  ]
}
```

If the return result is as expected after you use an ingress gateway to call the gRPC service over HTTP, the call is successful, and transcoding from HTTP/JSON to gRPC is successful.

1.2. Implement auto scaling for workloads by using ASM metrics

Alibaba Cloud Service Mesh (ASM) collects telemetry data for Container Service for Kubernetes (ACK) clusters in a non-intrusive manner, which makes the service communication in the clusters observable. This telemetry feature makes service behaviors observable and helps O&M staff troubleshoot, maintain, and optimize applications without increasing maintenance costs. Based on the four key monitoring metrics, including latency, traffic, errors, and saturation, ASM generates a series of metrics for the services that it manages. This topic describes how to implement auto scaling for workloads by using ASM metrics.

Prerequisites

- An ACK cluster is created. For more information, see [Create an ACK managed cluster](#).
- An ASM instance is created. For more information, see [Create an ASM instance](#).
- A Prometheus instance and a Grafana instance are deployed in the ACK cluster. For more information, see [Use Prometheus to monitor an ACK cluster](#).
- A Prometheus instance is deployed to monitor the ASM instance. For more information, see [Deploy a self-managed Prometheus instance to monitor ASM instances](#).

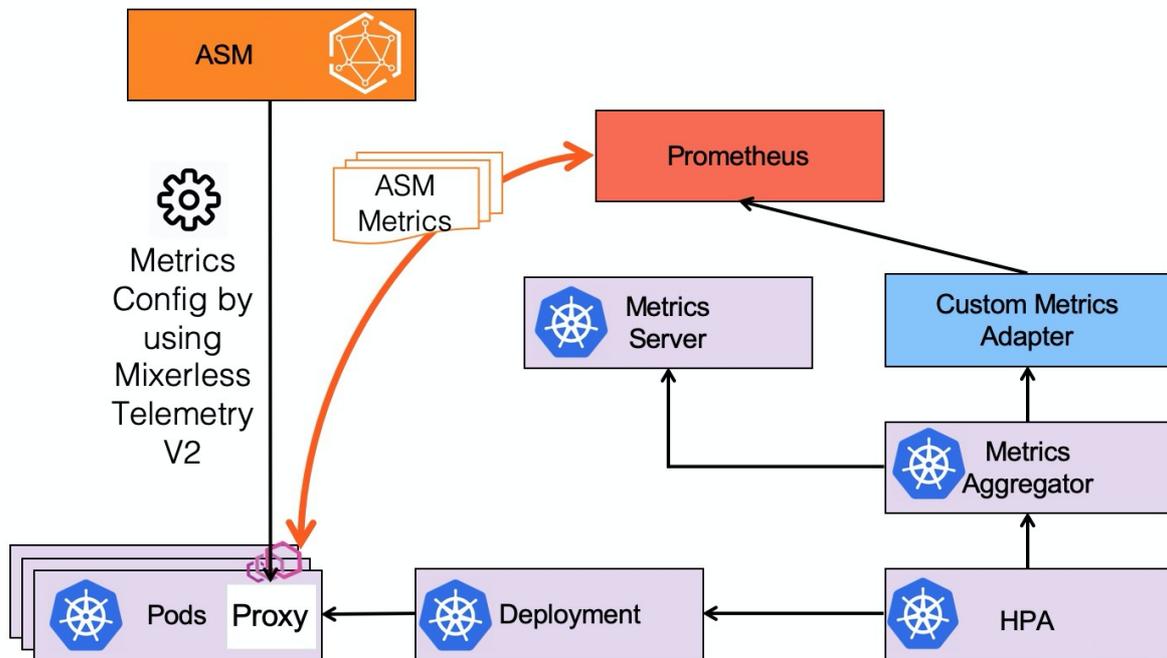
Context

ASM generates a series of metrics for the services that it manages. For more information, visit [Istio Standard Metrics](#).

Auto scaling is an approach that is used to automatically scale up or down workloads based on the resource usage. In Kubernetes, two autoscalers are used to implement auto scaling.

- Cluster Autoscaler (CA): CAs are used to increase or decrease the number of nodes in a cluster.
- Horizontal Pod Autoscaler (HPA): HPAs are used to increase or decrease the number of pods that are used to deploy applications.

The aggregation layer of Kubernetes allows third-party applications to extend the Kubernetes API by registering themselves as API add-ons. These add-ons can be used to implement the custom metrics API and allow HPAs to query any metrics. HPAs periodically query core metrics such as CPU utilization and memory usage by using the resource metrics API. In addition, HPAs use the custom metrics API to query application-specific metrics, such as the observability metrics that are provided by ASM.



Step 1: Enable Prometheus monitoring for the ASM instance

- 1.
- 2.
- 3.
4. On the details page of the ASM instance, choose in the left-side navigation pane. On the Basic Information page, click .

 **Note** Make sure that the Istio version of the ASM instance is 1.6.8.4 or later.

5. In the **Settings Update** panel, select **Enable Prometheus**, select **Enable Self-managed Prometheus**, enter the endpoint of the Prometheus instance, and then click **OK**.

After you enable Prometheus monitoring for the ASM instance, ASM automatically configures the Envoy filters that are required for Prometheus.

Step 2: Deploy the adapter for the custom metrics API

1. Download the installation package of the adapter. For more information, visit [kube-metrics-adapter](#). Then, install and deploy the adapter for the custom metrics API in the ACK cluster.

```
## Use Helm 3.
helm -n kube-system install asm-custom-metrics ./kube-metrics-adapter --set prometheus.url=http://prometheus.istio-system.svc:9090
```

2. After the installation is completed, run the following commands to check whether kube-metrics-adapter is enabled.

- o Check whether the autoscaling/v2beta API group exists.

```
kubectl api-versions |grep "autoscaling/v2beta"
```

Expected output:

```
autoscaling/v2beta
```

- o Check the status of the pod of kube-metrics-adapter.

```
kubectl get po -n kube-system |grep metrics-adapter
```

Expected output:

```
asm-custom-metrics-kube-metrics-adapter-85c6d5d865-2cm57      1/1      Running    0
19s
```

- o Query the custom metrics that are provided by kube-metrics-adapter.

```
kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1" | jq .
```

Expected output:

```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "external.metrics.k8s.io/v1beta1",
  "resources": []
}
```

Step 3: Deploy a sample application

1. Create a namespace named test. For more information, see [Manage namespaces](#).
2. Enable automatic sidecar injection. For more information, see [Install a sidecar proxy](#).
3. Deploy a sample application.
 - i. Create a file named *podinfo.yaml*.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: podinfo
  namespace: test
  labels:
    app: podinfo
spec:
  minReadySeconds: 5
  strategy:
    rollingUpdate:
      maxUnavailable: 0
    type: RollingUpdate
  selector:
    matchLabels:
      app: podinfo
```

```
    app: podinfo
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
      labels:
        app: podinfo
    spec:
      containers:
      - name: podinfo
        image: stefanprodan/podinfo:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9898
          name: http
          protocol: TCP
        command:
        - ./podinfo
        - --port=9898
        - --level=info
        livenessProbe:
          exec:
            command:
            - podcli
            - check
            - http
            - localhost:9898/healthz
          initialDelaySeconds: 5
          timeoutSeconds: 5
        readinessProbe:
          exec:
            command:
            - podcli
            - check
            - http
            - localhost:9898/readyz
          initialDelaySeconds: 5
          timeoutSeconds: 5
      resources:
        limits:
          cpu: 2000m
          memory: 512Mi
        requests:
          cpu: 100m
          memory: 64Mi
  ---
  apiVersion: v1
  kind: Service
  metadata:
    name: podinfo
    namespace: test
  labels:
    app: podinfo
  spec:
    type: ClusterIP
```

```
type: Service
ports:
  - name: http
    port: 9898
    targetPort: 9898
    protocol: TCP
selector:
  app: podinfo
```

ii. Deploy the podinfo application.

```
kubectl apply -n test -f podinfo.yaml
```

4. To trigger auto scaling, you must deploy a load testing service in the test namespace for triggering requests.

i. Create a file named *loadtester.yaml*.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: loadtester
  namespace: test
  labels:
    app: loadtester
spec:
  selector:
    matchLabels:
      app: loadtester
  template:
    metadata:
      labels:
        app: loadtester
      annotations:
        prometheus.io/scrape: "true"
    spec:
      containers:
        - name: loadtester
          image: weaveworks/flagger-loadtester:0.18.0
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8080
          command:
            - ./loadtester
            - -port=8080
            - -log-level=info
            - -timeout=1h
          livenessProbe:
            exec:
              command:
                - wget
                - --quiet
                - --tries=1
                - --timeout=4
                - --spider
```

```

      - http://localhost:8080/healthz
        timeoutSeconds: 5
      readinessProbe:
        exec:
          command:
            - wget
            - --quiet
            - --tries=1
            - --timeout=4
            - --spider
            - http://localhost:8080/healthz
          timeoutSeconds: 5
      resources:
        limits:
          memory: "512Mi"
          cpu: "1000m"
        requests:
          memory: "32Mi"
          cpu: "10m"
      securityContext:
        readOnlyRootFilesystem: true
        runAsUser: 10001
    ---
  apiVersion: v1
  kind: Service
  metadata:
    name: loadtester
    namespace: test
    labels:
      app: loadtester
  spec:
    type: ClusterIP
    selector:
      app: loadtester
    ports:
      - name: http
        port: 80
        protocol: TCP
        targetPort: http

```

ii. Deploy the load testing service.

```
kubectl apply -n test -f loadtester.yaml
```

5. Check whether the sample application and the load testing service are deployed.

i. Check the pod status.

```
kubectl get pod -n test
```

Expected output:

NAME	READY	STATUS	RESTARTS	AGE
loadtester-64df4846b9-nxhvv	2/2	Running	0	2m8s
podinfo-6d845cc8fc-26xbq	2/2	Running	0	11m

- ii. Log on to the container for load testing and run the hey command to generate loads.

```
export loadtester=$(kubectl -n test get pod -l "app=loadtester" -o jsonpath='{.items[0].metadata.name}')
kubectl -n test exec -it ${loadtester} -c loadtester -- hey -z 5s -c 10 -q 2 http://podinfo.test:9898
```

A load is generated, which indicates that the sample application and the load testing service are deployed.

Step 4: Configure an HPA by using ASM metrics

Define an HPA to scale the workloads of the Podinfo application based on the number of requests that the Podinfo application receives per second. When more than 10 requests are received per second on average, the HPA increases the number of replicas.

1. Create a file named *hpa.yaml* and copy the following code to the file:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo
  namespace: test
  annotations:
    metric-config.external.prometheus-query.prometheus/processed-requests-per-second: |
      sum(
        rate(
          istio_requests_total{
            destination_workload="podinfo",
            destination_workload_namespace="test",
            reporter="destination"
          }[1m]
        )
      )
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  metrics:
    - type: External
      external:
        metric:
          name: prometheus-query
          selector:
            matchLabels:
              query-name: processed-requests-per-second
        target:
          type: AverageValue
          averageValue: "10"
```

2. Deploy the HPA.

```
kubectl apply -f hpa.yaml
```

3. Check whether the HPA is deployed.

Query the custom metrics that are provided by kube-metrics-adapter.

```
kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1" | jq .
```

Expected output:

```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "external.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "prometheus-query",
      "singularName": "",
      "namespaced": true,
      "kind": "ExternalMetricValueList",
      "verbs": [
        "get"
      ]
    }
  ]
}
```

The output contains the resource list of custom ASM metrics, which indicates that the HPA is deployed.

Verify auto scaling

1. Log on to the container for load testing and run the hey command to generate loads.

```
kubectl -n test exec -it ${loadtester} -c loadtester -- sh
~ $ hey -z 5m -c 10 -q 5 http://podinfo.test:9898
```

2. View the effect of auto scaling.

Note Metrics are synchronized every 30 seconds by default. The container can be scaled only once in every 3 to 5 minutes. This way, the HPA can reserve time for automatic scaling before the conflict strategy is executed.

```
watch kubectl -n test get hpa/podinfo
```

Expected output:

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
podinfo	Deployment/podinfo	8308m/10 (avg)	1	10	6	124m

The HPA starts to scale up workloads in 1 minute until the number of requests per second decreases under the specified threshold. After the load testing is completed, the number of requests per second decreases to zero. Then, the HPA starts to decrease the number of pods. A few minutes later, the number of replicas decreases from the value in the preceding output to one.

2. Traffic Management

2.1. Use ASM to deploy an application in blue-green release mode and phased release mode

This topic describes how to use virtual services and destination rules of Alibaba Cloud Service Mesh (ASM) to deploy an application in blue-green release mode and phased release mode.

Prerequisites

- At least one ASM instance is created. For more information, see [Create an ASM instance](#).
- At least one Alibaba Cloud Container Service for Kubernetes (ACK) cluster is added to the ASM instance. For more information, see [Add a cluster to an ASM instance](#).
- The Bookinfo application is deployed in the ACK cluster that is added to the ASM instance. For more information, see [Deploy an application in an ASM instance](#).
- An ingress gateway is deployed for the ACK cluster that is added to the ASM instance. For more information, see [Deploy an ingress gateway service](#).

Create a destination rule

Create a destination rule for the Bookinfo application that is deployed in your ASM instance. For more information, see [Manage destination rules](#). The following code shows the configuration of a sample destination rule:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: productpage
spec:
  host: productpage
  subsets:
  - name: v1
    labels:
      version: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
```

```
name: v3
  labels:
    version: v3
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ratings
spec:
  host: ratings
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v2-mysql
    labels:
      version: v2-mysql
  - name: v2-mysql-vm
    labels:
      version: v2-mysql-vm
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: details
spec:
  host: details
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Create a virtual service

Create a virtual service for the Bookinfo application that is deployed in your ASM instance. For more information, see [Manage virtual services](#). The following code shows the configuration of a sample virtual service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: productpage
spec:
  hosts:
  - productpage
  http:
  - route:
    - destination:
        host: productpage
        subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: details
spec:
  hosts:
  - details
  http:
  - route:
    - destination:
        host: details
        subset: v1
```

Deploy version 2 in blue-green release mode

After the preceding destination rule and virtual service are created, version 2 of the reviews microservice of the Bookinfo application is running. However, no traffic is routed to version 2. To route traffic to version 2, you must deploy version 2 in blue-green release mode.

Create a virtual service to deploy version 2 of Bookinfo in blue-green release mode. For more information, see [Manage virtual services](#). The following code shows the configuration of a sample virtual service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
```

After the preceding virtual service is created, refresh the page of Bookinfo. The reviews microservice displays ratings as black stars.

Deploy version 3 in phased release mode to handle traffic by weight

You can run both version 2 and version 3 online and route traffic to the two versions by weight, such as 50% to 50%.

Create a virtual service to deploy version 3 of Bookinfo in phased release mode. For more information, see [Manage virtual services](#). The following code shows the configuration of a sample virtual service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
        weight: 50
    - destination:
        host: reviews
        subset: v3
        weight: 50
```

After the preceding virtual service is created, refresh the page of Bookinfo. The reviews microservice displays ratings by using version 2 or version 3 at random. The reviews microservice of version 3 displays ratings as red stars.

Deploy version 3 in phased release mode based on the request content

Phased release based on the traffic weight cannot meet the requirements of all scenarios. You can also deploy an application in phased release mode based on the user identity. This way, the application displays different pages for different users.

Create a virtual service to deploy the Bookinfo application in phased release mode. For more information, see [Manage virtual services](#). The following code shows the configuration of a sample virtual service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
      end-user:
        exact: jason
    route:
    - destination:
        host: reviews
        subset: v3
    - route:
        - destination:
            host: reviews
            subset: v2
```

After the preceding virtual service is created, refresh the page of Bookinfo. Bookinfo always displays ratings as black stars. You can click **Sign in** in the upper-right corner to log on to Bookinfo with the username jason. The logon does not require a password. After you log on, you can find that Bookinfo displays ratings as red stars.

 **Note** When you log on to Bookinfo and access its backend microservices, your requests contain the HTTP header `end-user=XXX`, which indicates the user identity. If you log on to Bookinfo with the username jason, the rule in the YAML file is matched and your requests are directed to the reviews microservice of version 3.

2.2. Use ASM and Wasm to implement end-to-end A/B testing in a non-intrusive manner

Alibaba Cloud Service Mesh (ASM) allows you to manage the traffic of microservices in a non-intrusive manner. However, to implement end-to-end A/B testing on a microservice in ASM without changes on the code of the microservice, you must also use WebAssembly (Wasm). This topic shows you how to use ASM and Wasm to implement end-to-end A/B testing in a non-intrusive manner.

Prerequisites

- An ASM instance is created. For more information, see [Create an ASM instance](#).

 **Note** Make sure that the version of the ASM instance is 1.8 or later.

- A Container Service for Kubernetes (ACK) cluster is added to the ASM instance. For more information, see [Add a cluster to an ASM instance](#).
- An ingress gateway is deployed in the ACK cluster that is added to the ASM instance. For more information, see [Deploy an ingress gateway service](#).
- An image repository is created in Container Registry. The address of the image repository and the information that is used to log on to the image repository are obtained. For more information, see [Use a Container Registry Enterprise Edition instance to push and pull images](#).

Context

Wasm is an effective and portable binary instruction format. You can use Wasm to extend the data plane of an ASM instance with new features. For more information about non-intrusive end-to-end A/B testing and Wasm development, see [Wasm-based non-intrusive end-to-end A/B testing](#).

 **Note** The image repository in this topic is for reference only. Use an image script to build and push images to your self-managed image repository. For more information about the image script, visit [hello-servicemesh-grpc](#).

Step 1: Enable Wasm-based ASM instance extension

1. Create a `runtime-config.json` file that contains the following code:

```
{
  "type": "envoy_proxy",
  "abiVersions": [
    "v0-541b2c1155fffb15ccde92b8324f3e38f7339ba6",
    "v0-097b7f2e4cc1fb490cc1943d0d633655ac3c522f",
    "v0-4689a30309abf31aee9ae36e73d34b1bb182685f",
    "v0.2.1"
  ],
  "config": {
    "rootIds": [
      "propaganda_filter_root"
    ]
  }
}
```

2. Run the following command to push a Wasm filter to an image repository in Container Registry:

```
oras push ${WASM_REGISTRY}/propagate_header:0.0.1 \
  --manifest-config \
  --runtime-config.json:application/vnd.module.wasm.config.v1+json \
  ${WASM_IMAGE}:application/vnd.module.wasm.content.layer.v1+wasm
```

- o `WASM_REGISTRY` : the address of the image repository.
- o `WASM_IMAGE` : the file name of the Wasm filter under the current path.
- o `runtime-config.json` : the runtime configuration file under the current path.

3. Enable Wasm-based ASM instance extension.

- i. Run the following command to check the version of Alibaba Cloud CLI:

The version of Alibaba Cloud CLI must be 3.0.73 or later.

```
aliyun version
```

- ii. Run the following command to enable Wasm-based ASM instance extension:

```
aliyun servicemesh UpdateMeshFeature --ServiceMeshId=xxxxxx --WebAssemblyFilterEnabled=true
```

4. Run the following command to check whether Wasm-based ASM instance extension is enabled:

```
aliyun servicemesh DescribeServiceMeshDetail \
  --ServiceMeshId $MESH_ID |
jq '.ServiceMesh.Spec.MeshConfig.WebAssemblyFilterDeployment'
```

The following output is expected:

```
{
  "Enabled": true
}
```

5. Run the following command to check the status of the `asmwasm-cache` DaemonSets:

After Wasm-based ASM instance extension is enabled, a DaemonSet that is named `asmwasm-cache` is created for each node of the ACK cluster.

```
kubectl get daemonset -n istio-system
```

The following output is expected:

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR
AGE						
asmwasm-cache	4	4	4	4	4	kubernetes.io/os=linux
34						

Step 2: Deploy resources for implementing A/B testing

1. Create a *hello.yaml* file that contains the following code in the *kube* directory:

The *hello.yaml* file defines the Hello1, Hello2, and Hello3 applications. Each application has two versions, which are version 1 and version 2.

 **Note** You can also obtain a YAML file that defines the Hello application from GitHub. For more information, visit [Kube](#).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello1-deploy-v1
  labels:
    app: hello1-deploy-v1
    service: hello1-deploy
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello1-deploy-v1
      service: hello1-deploy
      version: v1
  template:
    metadata:
      labels:
        app: hello1-deploy-v1
        service: hello1-deploy
        version: v1
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v1-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v1:1.0.0
          env:
            - name: HTTP_HELLO_BACKEND
              value: "hello2-svc"
          ports:
            - containerPort: 8001
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
name: hello1-deploy-v2
labels:
  app: hello1-deploy-v2
  service: hello1-deploy
  version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello1-deploy-v2
      service: hello1-deploy
      version: v2
  template:
    metadata:
      labels:
        app: hello1-deploy-v2
        service: hello1-deploy
        version: v2
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v2-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v2:1.0.0
          env:
            - name: HTTP_HELLO_BACKEND
              value: "hello2-svc"
          ports:
            - containerPort: 8001
              apiVersion: v1
---
kind: Service
metadata:
  name: hello1-svc
  labels:
    app: hello1-svc
spec:
  ports:
    - port: 8001
      name: http
  selector:
    service: hello1-deploy
    apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello2-deploy-v1
  labels:
    app: hello2-deploy-v1
    service: hello2-deploy
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello2-deploy-v1
      service: hello2-deploy
      version: v1
```

```
template:
  metadata:
    labels:
      app: hello2-deploy-v1
      service: hello2-deploy
      version: v1
  spec:
    serviceAccountName: http-hello-sa
    containers:
      - name: hello-v1-deploy
        image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v1:1.0.0
        env:
          - name: HTTP_HELLO_BACKEND
            value: "hello3-svc"
        ports:
          - containerPort: 8001
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello2-deploy-v2
  labels:
    app: hello2-deploy-v2
    service: hello2-deploy
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello2-deploy-v2
      service: hello2-deploy
      version: v2
  template:
    metadata:
      labels:
        app: hello2-deploy-v2
        service: hello2-deploy
        version: v2
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v2-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v2:1.0.0
          env:
            - name: HTTP_HELLO_BACKEND
              value: "hello3-svc"
          ports:
            - containerPort: 8001
---
kind: Service
metadata:
  name: hello2-svc
  labels:
    app: hello2-svc
```

```
spec:
  ports:
    - port: 8001
      name: http
  selector:
    service: hello2-deployapiVersion: apps/v1
---
kind: Deployment
metadata:
  name: hello3-deploy-v1
  labels:
    app: hello3-deploy-v1
    service: hello3-deploy
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello3-deploy-v1
      service: hello3-deploy
      version: v1
  template:
    metadata:
      labels:
        app: hello3-deploy-v1
        service: hello3-deploy
        version: v1
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v1-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v1:1.0.0
          ports:
            - containerPort: 8001
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello3-deploy-v2
  labels:
    app: hello3-deploy-v2
    service: hello3-deploy
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello3-deploy-v2
      service: hello3-deploy
      version: v2
  template:
    metadata:
      labels:
        app: hello3-deploy-v2
```

```

    service: hello3-deploy
    version: v2
  spec:
    serviceAccountName: http-hello-sa
    containers:
      - name: hello-v2-deploy
        image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v2:1.0.0
        ports:
          - containerPort: 8001
            apiVersion: v1
---
kind: Service
metadata:
  name: hello3-svc
  labels:
    app: hello3-svc
spec:
  ports:
    - port: 8001
      name: http
  selector:
    service: hello3-deploy
    apiVersion: v1
---
kind: ServiceAccount
metadata:
  name: http-hello-sa
  labels:
    account: http-hello-deploy

```

2. Create a *mesh.yaml* file that contains the following code in the *mesh* directory:

 **Note** You can also obtain a YAML file that defines ingress gateways, destination rules, and virtual services from GitHub. For more information, visit [Mesh](#).

The *mesh.yaml* file defines an ingress gateway, three destination rules, and three virtual services.

The following subsets are defined in the destination rules:

- o hello1v1: the version 1 of the Hello1 application. hello1v2: the version 2 of the Hello1 application.
- o hello2v1: the version 1 of the Hello2 application. hello2v2: the version 2 of the Hello2 application.
- o hello3v1: the version 1 of the Hello3 application. hello3v2: the version 2 of the Hello3 application.

The following routing rules are configured in the virtual services:

- o Only requests whose headers contain route-v:v2 can be routed to hello1v2. Otherwise, requests are routed to hello1v1.
- o Only requests whose headers contain route-v:hello2v2 can be routed to hello2v2. Otherwise, requests are routed to hello2v1.
- o Only requests whose headers contain route-v:hello3v2 can be routed to hello3v2. Otherwise, requests are routed to hello3v1.

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: DestinationRule
metadata:
  name: hello1-dr
spec:
  host: hello1-svc
  subsets:
    - name: hello1v1
      labels:
        version: v1
    - name: hello1v2
      labels:
        version: v2
---
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: hello-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 8001
        name: http
        protocol: HTTP
      hosts:
        - "*"
---
# https://istio.io/latest/docs/reference/config/networking/virtual-service/
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello1-vs
spec:
  hosts:
    - "*"
  gateways:
    - hello-gateway
  # - mesh
  http:
    - name: hello1-v1-route
      match:
        - headers:
            route-v:
              exact: v2
          route:
            - destination:
                host: hello1-svc
                subset: hello1v2
        - route:
            - destination:
                host: hello1-svc
                subset: hello1v1
---
```

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: hello2-dr
spec:
  host: hello2-svc
  subsets:
    - name: hello2v1
      labels:
        version: v1
    - name: hello2v2
      labels:
        version: v2
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello2-vs
spec:
  hosts:
    - hello2-svc
  http:
    - name: hello2-v2-route
      match:
        - headers:
            route-v:
              exact: hello2v2
          route:
            - destination:
                host: hello2-svc
                subset: hello2v2
        - route:
            - destination:
                host: hello2-svc
                subset: hello2v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: hello3-dr
spec:
  host: hello3-svc
  subsets:
    - name: hello3v1
      labels:
        version: v1
    - name: hello3v1
      labels:
        version: v2
    - name: hello3v2
      labels:
        version: v2
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```

kind: VirtualService
metadata:
  name: hello3-vs
spec:
  hosts:
  - hello3-svc
  http:
  - match:
    - headers:
        route-v:
          exact: hello3v2
      route:
    - destination:
        host: hello3-svc
        subset: hello3v2
  - route:
    - destination:
        host: hello3-svc
        subset: hello3v1

```

3. Run the following command to deploy the Hello application, ingress gateway, virtual services, and destination rules:

```

alias k="kubectl --kubeconfig $USER_CONFIG"
alias m="kubectl --kubeconfig $MESH_CONFIG"
k -n "$NS" apply -f kube/kube.yaml
m -n "$NS" apply -f mesh/mesh.yaml

```

Step 3: Deploy a custom ASMFilterDeployment resource

1. Create a secret for the ACK cluster to access the image repository.

For more information about the secrets of ACK clusters, see [Secret](#).

- i. Create a *myconfig.json* file that contains the following code:

```

{
  "auths":{
    "*****.cn-hangzhou.cr.aliyuncs.com":{
      "username":"*****username*****",
      "password":"*****password*****"
    }
  }
}

```

- `*****.cn-hangzhou.cr.aliyuncs.com` : the address of the image repository.
- `username` : the username of the image repository.
- `password` : the password of the image repository.

- ii. Run the following command to create a secret:

Note The secret must be named `asmwasm-cache` and reside in the `istio-system` namespace.

```
kubectl create secret generic asmwasm-cache -n istio-system --from-file=.dockerconfigjson=myconfig.json --type=kubernetes.io/dockerconfigjson
```

2. Deploy the `ASMFilterDeployment` resource.

- i. Create a `hello1-afd.yaml` file that contains the following code:

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMFilterDeployment
metadata:
  name: hello1-propagate-header
spec:
  workload:
    kind: Deployment
    labels:
      app: hello1-deploy-v2
      version: v2
  filter:
    patchContext: 'SIDECAR_OUTBOUND'
    parameters: '{"head_tag_name": "route-v", "head_tag_value": "hello2v2"}'
    image: 'wasm-repo-registry.cn-beijing.cr.aliyuncs.com/asm_wasm/propagate_header:0.0.1'
    rootID: 'propaganda_filter_root'
    id: 'hello1-propagate-header'
```

- Parameters in `workload` :
 - a. `kind` : the type of the workload.
 - b. `labels` : the filter conditions.
- Parameters in `filter` :
 - a. `patchContext` : the context that takes effect.
 - b. `parameters` : the parameters that are required for running the Wasm filter.
 - c. `image` : the address of the image repository to which the Wasm filter is pushed.
 - d. `rootID` : the root ID of the Wasm filter.
 - e. `id` : the unique ID of the Wasm filter.

- ii. Create a *hello2-afd.yaml* file that contains the following code:

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMFilterDeployment
metadata:
  name: hello2-propagate-header
spec:
  workload:
    kind: Deployment
    labels:
      app: hello2-deploy-v2
      version: v2
  filter:
    patchContext: 'SIDECAR_OUTBOUND'
    parameters: '{"head_tag_name": "route-v", "head_tag_value": "hello3v2"}'
    image: 'wasm-repo-registry.cn-beijing.cr.aliyuncs.com/asm_wasm/propagate_header:0.0.1'
    rootID: 'propaganda_filter_root'
    id: 'hello2-propagate-header'
```

- Parameters in `workload` :
 - a. `kind` : the type of the workload.
 - b. `labels` : the filter conditions.
- Parameters in `filter` :
 - a. `patchContext` : the context that takes effect.
 - b. `parameters` : the parameters that are required for running the Wasm filter.
 - c. `image` : the address of the image repository to which the Wasm filter is pushed.
 - d. `rootID` : the root ID of the Wasm filter.
 - e. `id` : the unique ID of the Wasm filter.

- iii. Run the following command to deploy the ASMFilterDeployment resource:

```
alias m="kubectl --kubeconfig $MESH_CONFIG"
m apply -f hello1-afd.yaml -n "$NS"
m apply -f hello2-afd.yaml -n "$NS"
```

3. Run the following command to check the deployment of the ASMFilterDeployment resource:

After the ASMFilterDeployment resource is deployed, ASM automatically generates an Envoy filter.

```
alias m="kubectl --kubeconfig $MESH_CONFIG"
m get envoyfilter -n "$NS"
m get ASMFilterDeployment -n "$NS"
```

The following output is expected:

```

NAME                AGE
hello1-propagate-header 1s
hello2-propagate-header 0s
NAME                STATUS      REASON      AGE
hello1-propagate-header Available    Available    1s
hello2-propagate-header Available    Available    1s

```

Implement A/B testing

Run the following command to implement A/B testing:

```

alias k="kubectl --kubeconfig $USER_CONFIG"
ingressGatewayIp=$(k -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
for j in {1..3}; do
  curl -H "route-v:v2" "http://$ingressGatewayIp:8001/hello/eric"
  echo
done

```

The following output is expected:

```

Bonjour eric@hello1:172.17.68.239<Bonjour eric@hello2:172.17.68.209<Bonjour eric@hello3:172.17.68.208
Bonjour eric@hello1:172.17.68.239<Bonjour eric@hello2:172.17.68.209<Bonjour eric@hello3:172.17.68.208
Bonjour eric@hello1:172.17.68.239<Bonjour eric@hello2:172.17.68.209<Bonjour eric@hello3:172.17.68.208

```

The output indicates that if the headers of the request contain route-v:v2, the request can be routed to hello1v2, hello2v2, and hello3v2.

Troubleshooting

If the expected output is not returned, you can run the following script code to check the logs of workloads.

- Check Envoy access logs

```

alias k="kubectl --kubeconfig $USER_CONFIG"
hello1_v2_pod=$(k get pod -l app=hello1-deploy-v2 -n "$NS" -o jsonpath={.items..metadata.name})
# Change the level of Envoy access logs to info.
k -n "$NS" exec "$hello1_v2_pod" -c istio-proxy -- curl -XPOST -s "http://localhost:15000/logging?level=info"
# Display Envoy access logs.
k -n "$NS" logs -f deployment/hello1-deploy-v2 -c istio-proxy

```

- Check the logs of the Hello application

```

alias k="kubectl --kubeconfig $USER_CONFIG"
k -n "$NS" logs -f deployment/hello2-deploy-v1 -c hello-v1-deploy

```

2.3. Use ASM and KubeVela to implement a canary release

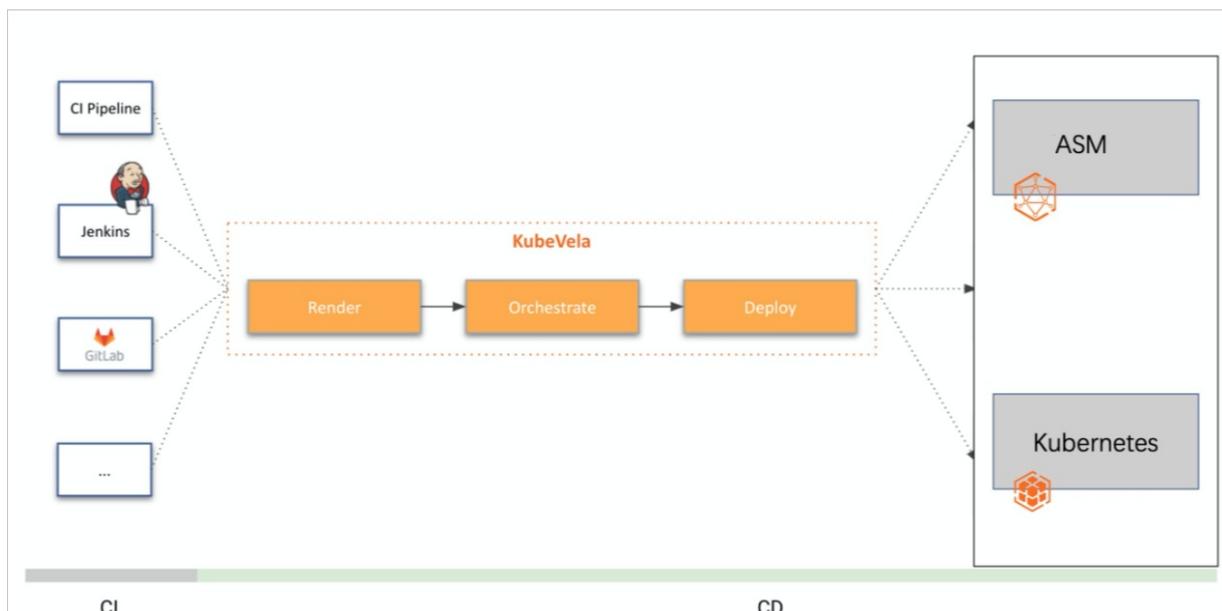
KubeVela is a modern and out-of-the-box platform used to deliver and manage applications. You can use Alibaba Cloud Service Mesh (ASM) and KubeVela to implement canary releases for applications. In canary releases, applications can be updated in a gradual manner. This topic describes how to use ASM and KubeVela to implement a canary release.

Prerequisites

- An ASM instance whose version is v1.9.7.93-g7910a454-aliyun or later is created. For more information, see [Create an ASM instance](#).
-
-
- The ACK cluster is connected by using kubectl. For more information, see [Connect to ACK clusters by using kubectl](#).
- KubeVela CLI is installed. For more information, see [Installation](#).
- The Kubernetes API of clusters on the data plane is allowed to access Istio resources. For more information, see [Use the Kubernetes API of clusters on the data plane to access Istio resources](#).

Context

KubeVela is a modern and out-of-the-box platform that makes it easier to deliver and manage applications across hybrid environments. In addition, KubeVela is highly extensible and allows you to deal with rapid business changes by updating applications with ease. The Open Application Model (OAM) of KubeVela is designed and implemented with extreme extensibility. OAM provides programmable delivery workflows and is application-oriented and independent of infrastructure. For more information, see [Progressive Rollout with Istio](#).



Usage notes

Before you start, you must download and decompress the `asm_kubevela` package on your computer. All required files are stored in the `asm_kubevela` folder.

The `asm_kubevela` folder contains the following files: `application.yaml`, `application_rollback.yaml`, `application_rollout-v2.yaml`, `canary-rollout-wf-def.yaml`, `rollback-wf-def.yaml`, and `traffic-trait-def.yaml`. Where:

- In Step 3, the `canary-rollout-wf-def.yaml`, `rollback-wf-def.yaml`, and `traffic-trait-def.yaml` files are used.
- In Step 4, the `application.yaml` file is used.
- In Step 5, the `application_rollout-v2.yaml` file is used.
- In Step 6, the `application_rollback.yaml` file is used.

Step 1: Install KubeVela

- 1.
- 2.
3. On the **App Catalog** page, search for `ack-kubevela`. Then, click `ack-kubevela`.
4. On the details page, click **Deploy** in the upper-right corner. In the **Deploy** panel, select a cluster, set relevant parameters, and then click **OK**.

Step 2: Enable automatic sidecar injection

- 1.
- 2.
- 3.
- 4.
5. On the **Global Namespace** page, find the default namespace and click **Enable Automatic Sidecar Injection** in the **Automatic Sidecar Injection** column.
6. In the **Submit** message, click **OK**.

Step 3: Deploy the configuration files of KubeVela

To integrate the traffic management rules of KubeVela with those of ASM, deploy the configuration files of KubeVela.

 **Note** Before you perform this operation, make sure that the Kubernetes API of clusters on the data plane is allowed to access Istio resources. Otherwise, an error is reported. For more information, see [Use the Kubernetes API of clusters on the data plane to access Istio resources](#).

Navigate to the `asm_kubevela` folder in the command prompt window. Then, run the following commands to deploy the configuration files of KubeVela:

```
kubectl apply -f rollback-wf-def.yaml
```

```
kubectl apply -f canary-rollout-wf-def.yaml
```

```
kubectl apply -f traffic-trait-def.yaml
```

Step 4: Deploy an application and a gateway

1. Navigate to the `asm_kubevela` folder in the command prompt window. Then, run the following command to deploy the Bookinfo application:

```
kubectl apply -f application.yaml
```

In the `application.yaml` file, the `type` parameter in the `traits` parameter of the `reviews` application is set to `canary-traffic`. This indicates that a canary release is configured.

2. Deploy a gateway and a virtual service in the ASM console.
 - i.
 - ii.
 - iii.
 - iv. Deploy a gateway.
 - a. On the details page of the ASM instance, choose **Traffic Management > Gateway** in the left-side navigation pane. On the Gateway page, click **Create from YAML**.
 - b. On the **Create** page, select `default` from the Namespace drop-down list, copy the following content to the code editor, and then click **Create**.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"

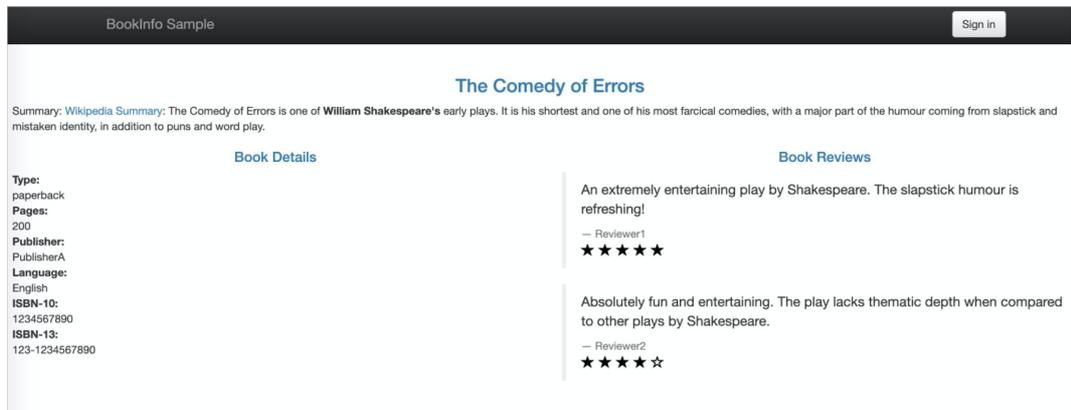
```

- v. Deploy a virtual service.
 - a. On the details page of the ASM instance, choose **Traffic Management** > **VirtualService** in the left-side navigation pane. On the VirtualService page, click **Create from YAML**.
 - b. On the **Create** page, select default from the Namespace drop-down list, copy the following content to the code editor, and then click **Create**.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
  - "*"
  gateways:
  - bookinfo-gateway
  http:
  - match:
    - uri:
        exact: /productpage
    - uri:
        prefix: /static
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        prefix: /api/v1/products
  route:
  - destination:
      host: productpage
      port:
        number: 9080
```

- vi. Access the Bookinfo application.
 - a.
 - b.
 - c.
 - d. On the cluster details page, choose **Network > Services** in the left-side navigation pane.
 - e. In the upper part of the Services page, select `istio-system` from the Namespace drop-down list. Find `istio-ingressgateway` and view the external endpoint whose port number is 80 in the External Endpoint column. Then, enter *IP address of the ingress gateway whose port number is 80/product page* in the address bar of your browser to access the Bookinfo application.

Refresh the page multiple times. You can see that black stars are displayed on the page.



Step 5: Perform a canary release for an application

1. Navigate to the `asm_kubevela` folder in the command prompt window. Then, run the following command to update the reviews application and adjust the traffic routed to the application:

```
kubectl apply -f application_rollout-v2.yaml
```

- o The `application_rollout-v2.yaml` file is used to update the reviews image from V2 to V3. In addition, the file specifies that two instances are updated one by one in two phases.

```
...
- name: reviews
  type: webservice
  properties:
    image: docker.io/istio/examples-bookinfo-reviews-v3:1.16.2
    port: 9080
    volumes:
      - name: wlp-output
        type: emptyDir
        mountPath: /opt/ibm/wlp/output
      - name: tmp
        type: emptyDir
        mountPath: /tmp
  traits:
    - type: expose
      properties:
        port:
          - 9080
    - type: rollout
      properties:
        targetSize: 2
        rolloutBatches:
          - replicas: 1
          - replicas: 1
    - type: canary-traffic
      properties:
        port: 9080
...
```

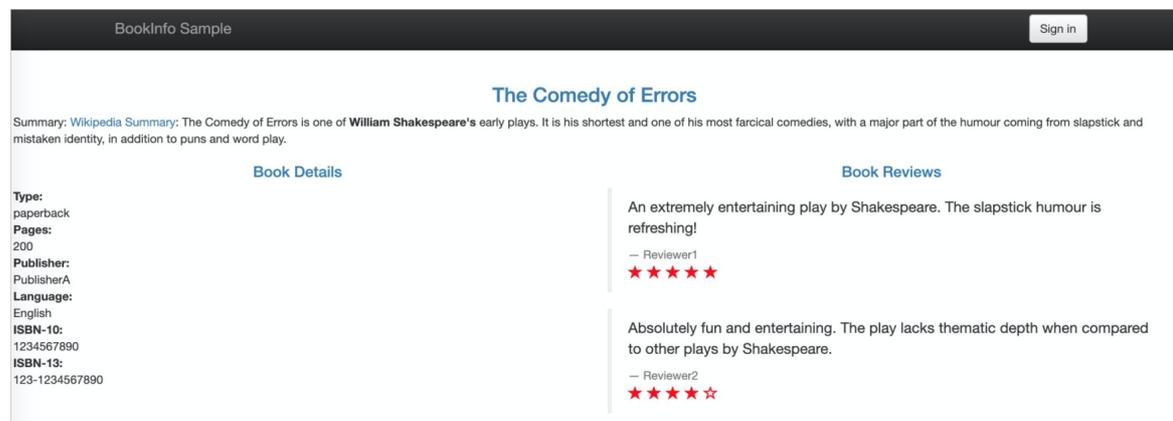
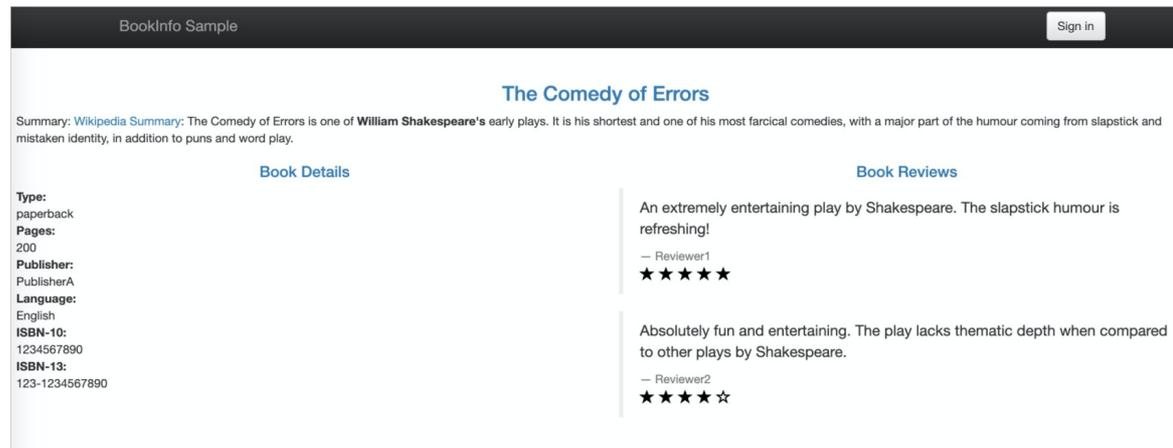
- targetSize: the number of phases for updating instances.
- rolloutBatches: the number of instances to be updated in each phase.
- The application_rollout-v2.yaml file specifies the following workflows:
 - a. The batchPartition parameter is set to 0. This specifies that only one of the two pods of the reviews application is updated. The traffic.weightedTargets parameter is set to specify that 10% of the traffic is routed to the new reviews application, whereas 90% of the traffic is routed to the earlier version.
 - b. The type parameter is set to *suspend*. This specifies that the application release is suspended after the first workflow is complete.
 - c. The batchPartition parameter is set to 1. This specifies that both pods of the reviews applications are updated to V3. The traffic.weightedTargets parameter is set to specify that all traffic is routed to the new reviews application.

```
...
workflow:
  steps:
    - name: rollout-1st-batch
      type: canary-rollout
      properties:
        # just upgrade first batch of component
        batchPartition: 0
        traffic:
          weightedTargets:
            - revision: reviews-v1
              weight: 90 # 90% shift to new version
            - revision: reviews-v2
              weight: 10 # 10% shift to new version
        # give user time to verify part of traffic shifting to newRevision
    - name: manual-approval
      type: suspend
    - name: rollout-rest
      type: canary-rollout
      properties:
        # upgrade all batches of component
        batchPartition: 1
        traffic:
          weightedTargets:
            - revision: reviews-v2
              weight: 100 # 100% shift to new version
...

```

2. Enter *IP address of the ingress gateway whose port number is 80/product page* in the address bar of your browser to access the Bookinfo application.

Refresh the page multiple times. You can see red stars for 10% of the times and black stars for 90% of the times.

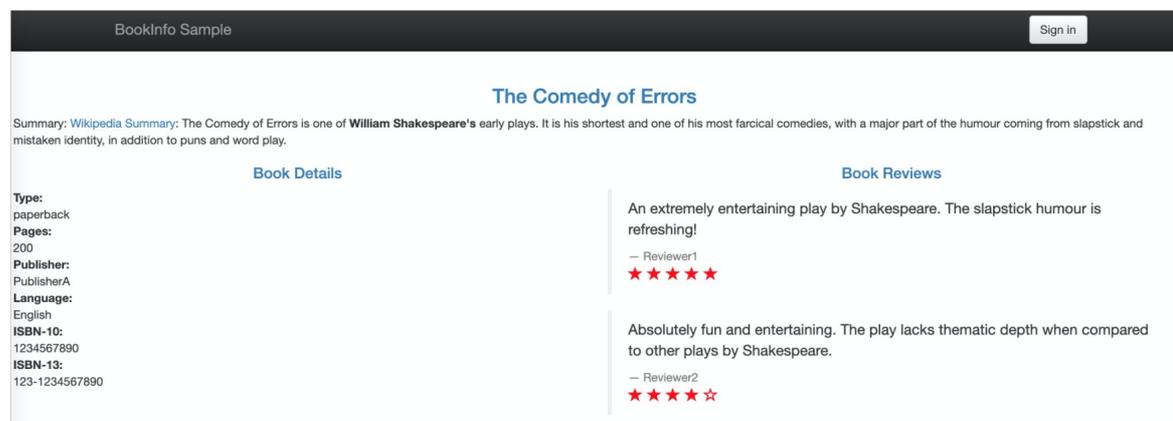


- Run the following command to continue the application release to update all images of the reviews application to V3:

```
vela workflow resume book-info
```

- Enter *IP address of the ingress gateway whose port number is 80/product page* in the address bar of your browser to access the Bookinfo application.

Refresh the page multiple times. The page displays only red stars. This indicates that the reviews application is updated to V3.



Step 6: (Optional) Roll back the application

If you find that the new application does not meet your expectations, you can stop the application release and roll back the application to the earlier version.

1. Run the following command to roll back the application:

```
kubectl apply -f rollback.yaml
```

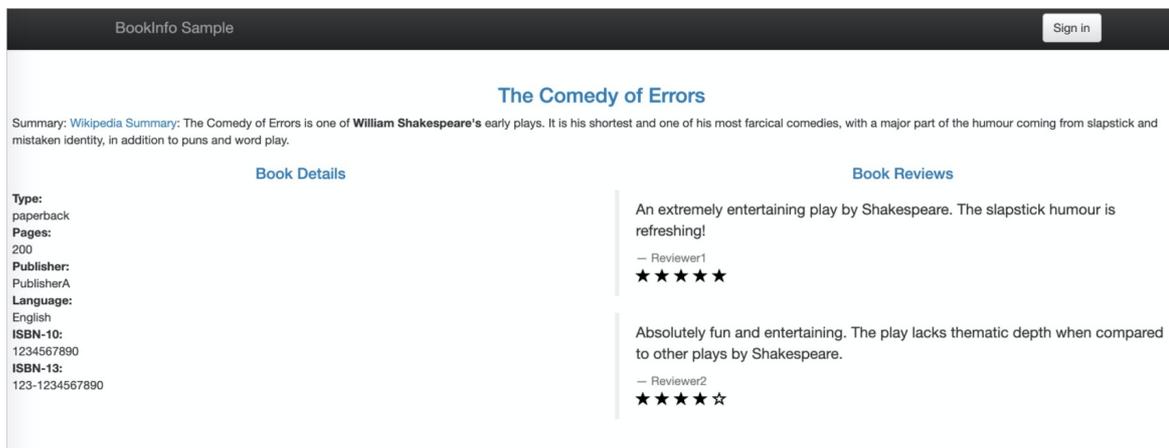
In the `rollback.yaml` file, the `type` parameter is set to `canary-rollback`. The following operations are automatically performed:

- Update the `targetRevisionName` parameter of the application to the earlier version. Roll back all instances of the new application to the earlier version and keep all earlier instances that are not updated.
- Update the `route` parameter of the virtual service to route all traffic to the earlier version.
- Update the `subset` parameter of the destination rule to the earlier version.

```
...
workflow:
  steps:
    - name: rollback
      type: canary-rollback
```

2. Enter *IP address of the ingress gateway whose port number is 80/product page* in the address bar of your browser to access the Bookinfo application.

Refresh the page multiple times. The page displays only black stars. This indicates that the reviews application is rolled back to V2.



2.4. Use an ASM instance of a commercial edition to implement an end-to-end canary release

If you need to implement an end-to-end canary release among multiple services, you can configure the TrafficLabel custom resource definition (CRD) to identify traffic characteristics and divide the ingress traffic of a gateway into regular traffic and canary traffic. The canary traffic characteristics are passed among the services that are used to process user requests. This way, an end-to-end canary release is implemented. This topic uses a sample demo to describe how to use the TrafficLabel CRD to implement an end-to-end canary release for microservices.

Prerequisites

- An Alibaba Cloud Service Mesh (ASM) instance of a commercial edition is created. The Istio version of the ASM instance is 1.10.5.40 or later. For more information, see [Create an ASM instance](#).
-
-
- Tracing Analysis is enabled for the ASM instance. When you create the ASM instance, select **Enable Tracing Analysis** in the Observability section. For more information, see [Create an ASM instance](#).
- An ASM gateway is created. For more information, see [Deploy an ingress gateway service](#).
- Application monitoring is enabled. For more information, see [Monitor application performance](#).

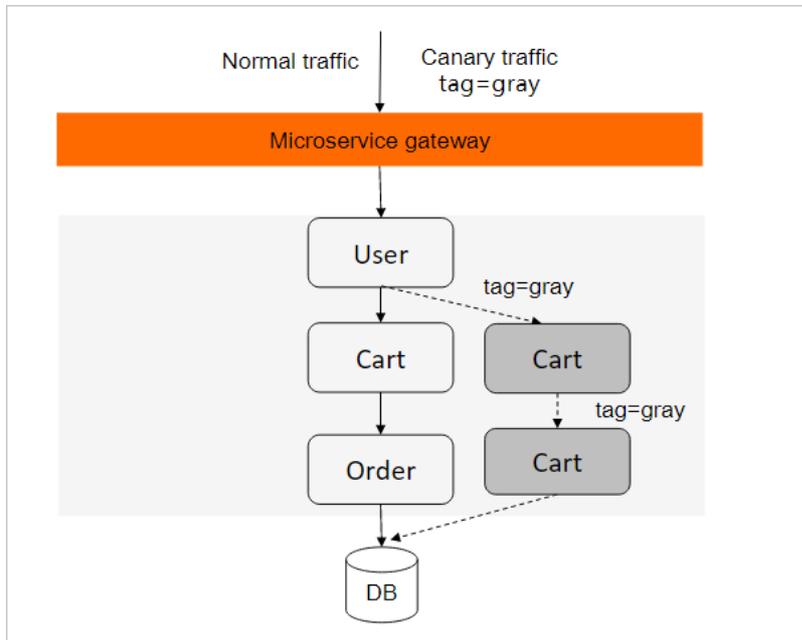
 **Note** In this example, the demo application is connected to Application Real-Time Monitoring Service (ARMS) by using the ARMS agent for Java. For more information about how to connect to ARMS by using agents for other programming languages, see [Overview](#).

- A Container Service for Kubernetes (ACK) cluster is connected by using kubectl. For more information, see [Connect to ACK clusters by using kubectl](#).
- The ASM instance is connected by using kubectl. For more information, see [Use kubectl to connect to an ASM instance](#).

Context

Canary releases can be implemented in various ways. For example, you can use ASM and KubeVela to implement a progressive canary release, and use ASM to implement a blue-green release and a canary release for an application. For more information, see [Use ASM and KubeVela to implement a canary release](#) and [Use ASM to deploy an application in blue-green release mode and phased release mode](#). The preceding two types of canary releases focus on the release of a single service by using the label-based routing and weight-based traffic distribution of a VirtualService provided by Istio.

In specific scenarios, the canary release only between two services cannot meet the requirements. For example, the Cart and Order services both have canary release versions, as shown in the following figure.



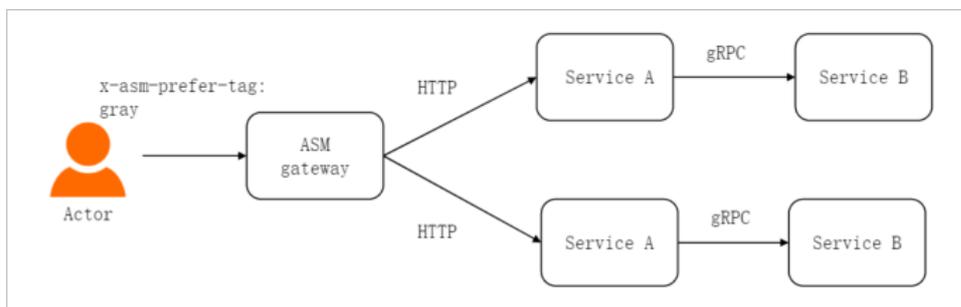
When you verify the canary release in this scenario, you can find that: The ingress traffic includes regular traffic and canary traffic, and the User service needs to identify the traffic characteristics of user requests. The canary traffic is routed to the canary release version of the Cart service. In this scenario, the system no longer simply distributes traffic to different backend versions at a specific traffic ratio. Instead, the canary traffic characteristics are passed among all the services that are used to process user requests.

The end-to-end canary release in ASM is implemented based on the traffic labeling and label-based routing features. For more information, see [Traffic labeling and label-based routing](#).

Demo

You can [download](#) the orchestration files and related configuration files of the demo.

The following figure shows the architecture of the demo.



The Deployment orchestration file *demo.yaml* contains the following code:

```
apiVersion: v1
kind: Service
metadata:
  name: spring-boot-istio-client
spec:
```

```

    type: ClusterIP
    ports:
    - name: http
      port: 80
      targetPort: 19090
    selector:
      app: spring-boot-istio-client
  ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-istio-client
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-boot-istio-client
      version: base
  template:
    metadata:
      annotations:
        armsPilotAutoEnable: 'on'
        armsPilotCreateAppName: spring-boot-istio-client
      labels:
        app: spring-boot-istio-client
        version: base
    spec:
      containers:
      - name: spring-boot-istio-client
        image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-client:Abase
        imagePullPolicy: Always
        tty: true
        ports:
        - name: http
          protocol: TCP
          containerPort: 19090
  ---
apiVersion: v1
kind: Service
metadata:
  name: spring-boot-istio-server
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 18080
    targetPort: 18080
  - name: grpc
    port: 18888
    targetPort: 18888
  selector:
    app: spring-boot-istio-server
  ---

```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-istio-server
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-boot-istio-server
      version: base
  template:
    metadata:
      annotations:
        armsPilotAutoEnable: 'on'
        armsPilotCreateAppName: spring-boot-istio-server
      labels:
        app: spring-boot-istio-server
        version: base
    spec:
      containers:
        - name: spring-boot-istio-server
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-server:Base
          imagePullPolicy: Always
          tty: true
          ports:
            - name: http
              protocol: TCP
              containerPort: 18080
            - name: grpc
              protocol: TCP
              containerPort: 18888
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-istio-client-gray
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-boot-istio-client
      version: gray
  template:
    metadata:
      annotations:
        armsPilotAutoEnable: 'on'
        armsPilotCreateAppName: spring-boot-istio-client
      labels:
        app: spring-boot-istio-client
        version: gray
    spec:
      containers:
        - name: spring-boot-istio-client
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-client:gray
```

```

    image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-server-gray
  ent:Agray
    imagePullPolicy: Always
    tty: true
    ports:
      - name: http
        protocol: TCP
        containerPort: 19090
  ---
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: spring-boot-istio-server-gray
  spec:
    replicas: 2
    selector:
      matchLabels:
        app: spring-boot-istio-server
        version: gray
    template:
      metadata:
        annotations:
          armsPilotAutoEnable: 'on'
          armsPilotCreateAppName: spring-boot-istio-server
        labels:
          app: spring-boot-istio-server
          version: gray
      spec:
        containers:
          - name: spring-boot-istio-server
            image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-server-gray
  r:Bgray
    imagePullPolicy: Always
    tty: true
    ports:
      - name: http
        protocol: TCP
        containerPort: 18080
      - name: grpc
        protocol: TCP
        containerPort: 18888

```

The demo services are all Java applications that use the Spring Boot framework. In addition, ARMS application monitoring is enabled to monitor the services. For more information, see [Monitor application performance](#).

The `template metadata` of each Deployment in the `demo.yaml` file contains configurations similar to the following content:

```

template:
  metadata:
    annotations:
      armsPilotAutoEnable: 'on'
      armsPilotCreateAppName: spring-boot-istio-server

```

Step 1: Deploy the demo microservices in the ACK cluster

Run the following command to deploy the demo:

```
kubectl apply -f demo.yaml
```

Step 2: Configure simple routing

1. Use the following code to create the *istio-config.yaml* file:

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: simple-springboot-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - "*"
      port:
        name: http
        number: 80
        protocol: HTTP
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-client-vs
spec:
  gateways:
    - simple-springboot-gateway
  hosts:
    - "*"
  http:
    - match:
        - uri:
            prefix: "/hello"
      route:
        - destination:
            host: spring-boot-istio-client
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-server-vs
spec:
  hosts:
    - spring-boot-istio-server
  http:
    - route:
        - destination:
            host: spring-boot-istio-server
---
```

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: springboot-istio-client-dr
spec:
  host: spring-boot-istio-client
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
    - labels:
        version: base
      name: version-base
    - labels:
        version: gray
      name: version-gray
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: springboot-istio-server-dr
spec:
  host: spring-boot-istio-server
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
    - labels:
        version: base
      name: version-base
    - labels:
        version: gray
      name: version-gray
```

2. Run the following command to configure routing:

```
kubectl --kubeconfig <The kubeconfig file of the ASM instance> apply -f istio-config.yaml
```

3. Check whether a service can be accessed.

- i. Obtain the public IP address of the ASM gateway in the and run the following command:

```
export ASM_GATEWAY_IP=xxx
```

- ii. Run the following command to check whether a service can be accessed:

```
while true; do curl -H'x-asm-prefer-tag: gray' http://${ASM_GATEWAY_IP}/hello ; echo;sleep 1;done
```

Expected output:

```
--> HTTP A-Base --> gRPC B-Gray.  
--> HTTP A-Gray --> gRPC B-Base.  
--> HTTP A-Base --> gRPC B-Gray.  
--> HTTP A-Gray --> gRPC B-Base.  
--> HTTP A-Base --> gRPC B-Base.
```

The traffic that flows from the gateway to Service A and then to Service B is an example of traffic routing from the ASM gateway to the base and canary release versions of services in a load balancing manner. In this case, the `x-asm-prefer-tag` header that you set in the curl command takes effect only if the TrafficLabel CRD and corresponding label-based routing rule are configured. By default, the `istio-config.yaml` file is used to configure simple routing, and subsets are not specified in the routing configuration under VirtualService.

Step 3: Configure traffic labels

1. Use the following code to create the `traffic_label_default.yaml` file:

```
---
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: example1
  namespace: default
spec:
  rules:
  - labels:
    - name: userdefinelabel1
      valueFrom:
      - $getContext(x-b3-traceid)
      - $localLabel
    attachTo:
    - opentracing
    # The protocols that take effect. If you do not set the protocols parameter, no protocol takes effect. If you set the protocols parameter to an asterisk (*), all protocols take effect.
    protocols: "*"
    hosts: # The services that take effect.
    - "*"
---
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: ingressgateway
  namespace: istio-system
spec:
  hosts:
  - '*'
  rules:
  - attachTo:
    - opentracing
    labels:
    - name: userdefinelabel1
      valueFrom:
      - $getContext(x-b3-traceid)
      - $localLabel
    protocols: '*'
  workloadSelector:
    labels:
    app: istio-ingressgateway
```

2. Run the following command to use the kubeconfig file of the ASM instance for deployment:

```
kubectl --kubeconfig <The kubeconfig file of the ASM instance> apply -f traffic_label_default.yaml
```

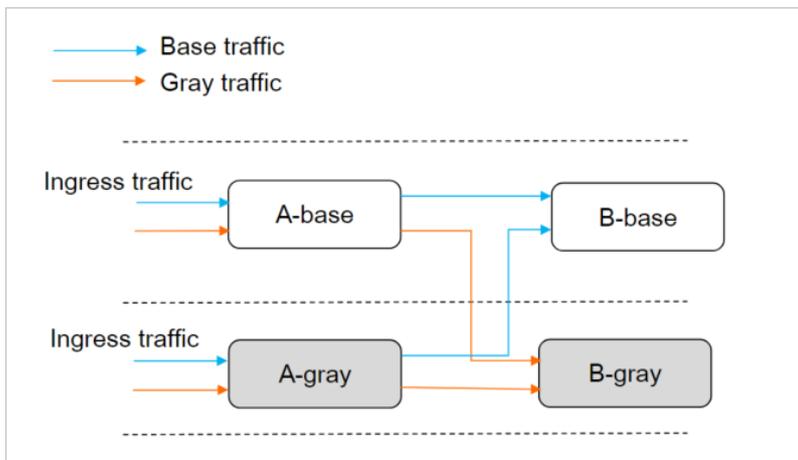
The TrafficLabel CRD applies to all services in the default namespace, including Service A and Service B that are deployed by using the *demo.yaml* file.

Note In this example, the demo is connected to ARMS that uses `traceId` of the Zipkin tracer type. Therefore, the `getContext` parameter is set to `x-b3-traceid`.

Step 4: Verify the TrafficLabel-based routing

1. Check whether the traffic routing from Service A to Service B meets the requirements. To be specific, check whether the canary traffic of Service A is forwarded to the canary release version of Service B, and whether the base traffic of Service A is forwarded to the base version of Service B.

Configure the TrafficLabel-based routing file `b-vs-tf.yaml` for Service B and make the file take effect for Service A. The following figure shows the corresponding traffic routing model.



- i. Use the following code to create the `b-vs-tf.yaml` file:

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-server-vs
spec:
  hosts:
  - spring-boot-istio-server
  http:
  - route:
    - destination:
        host: spring-boot-istio-server
        subset: $userdefinelabel1
```

- ii. Run the following command for the `b-vs-tf.yaml` file to take effect for Service A:

```
kubectl -f <The kubeconfig file of the ASM instance> apply -f b-vs-tf.yaml
```

- iii. Run the following command to check whether the canary traffic of Service A is forwarded to the canary release version of Service B:

```
while true; do curl -H'x-asm-prefer-tag: version-gray' http://{ASM_GATEWAY_IP}/hello ; echo;sleep 1;done
```

Expected output:

```
--> HTTP A-Gray --> gRPC B-Gray.  
--> HTTP A-Base --> gRPC B-Gray.  
--> HTTP A-Gray --> gRPC B-Gray.  
--> HTTP A-Gray --> gRPC B-Gray.  
--> HTTP A-Gray --> gRPC B-Gray.  
--> HTTP A-Base --> gRPC B-Gray.  
--> HTTP A-Base --> gRPC B-Gray.  
--> HTTP A-Base --> gRPC B-Gray.  
--> HTTP A-Gray --> gRPC B-Gray.  
--> HTTP A-Base --> gRPC B-Gray.  
--> HTTP A-Base --> gRPC B-Gray.
```

- iv. Run the following command to check whether the base traffic of Service A is forwarded to the base version of Service B:

```
while true; do curl -H'x-asm-prefer-tag: version-base' http://{ASM_GATEWAY_IP}/hello ; echo;sleep 1;done
```

Expected output:

```
--> HTTP A-Base --> gRPC B-Base.  
--> HTTP A-Base --> gRPC B-Base.  
--> HTTP A-Base --> gRPC B-Base.  
--> HTTP A-Gray --> gRPC B-Base.  
--> HTTP A-Gray --> gRPC B-Base.  
--> HTTP A-Gray --> gRPC B-Base.  
--> HTTP A-Base --> gRPC B-Base.  
--> HTTP A-Gray --> gRPC B-Base.  
--> HTTP A-Gray --> gRPC B-Base.  
--> HTTP A-Base --> gRPC B-Base.  
--> HTTP A-Gray --> gRPC B-Base.
```

 **Note** If the ingress traffic that passes Service A is not forwarded to the specified version, you must configure the TrafficLabel-based routing file for Service A.

2. Check whether the traffic routing from the ASM gateway to Service A meets the requirements. To be specific, check whether the canary traffic of ingress requests is forwarded to the canary release version of Service A, and whether the base traffic of ingress requests is first forwarded to the base version of Service A and then to that of Service B.

Configure the TrafficLabel-based routing file *a-vs-tf.yaml* for Service A and make the file take effect for the ASM gateway.

 **Note** ASM gateways also support TrafficLabel-based routing.

- i. Use the following code to create the *a-vs-tf.yaml* file:

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-client-vs
spec:
  gateways:
  - simple-springboot-gateway
  hosts:
  - "*"
  http:
  - match:
    - uri:
        prefix: "/hello"
    route:
    - destination:
        host: spring-boot-istio-client
        subset: $userdefinelabel1
```

- ii. Run the following command for the *a-vs-tf.yaml* file to take effect for the ASM gateway:

```
kubectl -f <The kubeconfig file of the ASM instance> apply -f a-vs-tf.yaml
```

- iii. Run the following command to check whether the canary traffic of ingress requests is forwarded to the canary release version of Service A:

```
while true; do curl -H'x-asm-prefer-tag: version-gray' http://${ASM_GATEWAY_IP}/hello ; echo;sleep 1;done
```

Expected output :

```
--> HTTP A-Gray --> gRPC B-Gray.
```

- iv. Run the following command to check whether the base traffic of ingress requests is first forwarded to the base version of Service A and then to that of Service B:

```
while true; do curl -H'x-asm-prefer-tag: version-base' http://${ASM_GATEWAY_IP}/hello ; echo;sleep 1;done
```

Expected output:

```
--> HTTP A-Base --> gRPC B-Base.  
--> HTTP A-Base --> gRPC B-Base.
```

3. Check whether the weight-based traffic distribution that corresponds to the TrafficLabel-based routing meets the requirements.

- i. Use the following code to create the *a-vs-tf-10-90.yaml* file:

```
---  
apiVersion: networking.istio.io/v1beta1  
kind: VirtualService  
metadata:  
  name: springboot-istio-client-vs  
spec:  
  gateways:  
  - simple-springboot-gateway  
  hosts:  
  - "*"br/>  http:  
  - match:  
    - uri:  
      prefix: "/hello"  
    route:  
      - destination:  
        host: spring-boot-istio-client  
        subset: $userdefinelabel1  
        weight: 10  
      - destination:  
        host: spring-boot-istio-client  
        subset: version-base  
        weight: 90
```

 **Note** Only 10% of the canary or base traffic is forwarded to the corresponding subset that you specify. The remaining traffic is forwarded to the version-base subset.


```
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: example1
  namespace: default
spec:
  rules:
  - labels:
    - name: userdefinelabel1
      valueFrom:
      - $localLabel
    attachTo:
    - opentracing
    # The protocols that take effect. If you do not set the protocols parameter, no protocol takes effect. If you set the protocols parameter to an asterisk (*), all protocols take effect.
    protocols: "*"
    hosts: # The services that take effect.
    - "*"

```

Canary traffic configuration at the ASM gateway

If you need to clarify the canary traffic in ingress traffic by using `<gatewayIP>/hello`, you must use the `x-asm-prefer-tag` header to specify a traffic label, as shown in the preceding *a-vs-tf.yaml* file. In the preceding examples, the canary traffic is manually labeled by running the `curl -H 'x-asm-prefer-tag: xxx'` command.

In actual business scenarios, a client application or user may use a browser for access without setting the `x-asm-prefer-tag` header. In such scenarios, you can use the custom header feature of an ASM gateway and the Lua plug-in to map the canary configuration to the `x-asm-prefer-tag` header for standardized processing.

For example, you can use an `Envoy filter` to specify the traffic that is generated by the users who use iPhone 13 as canary traffic. Sample code:

```

apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  labels:
    provider: "asm"
    asm-system: "true"
  name: gateway-lua-filter-add-x-asm-prefer-tag-header
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      istio: ingressgateway
  configPatches:
  - applyTo: HTTP_FILTER
    match:
      proxy:
        proxyVersion: "^1.*"
      context: GATEWAY
      listener:
        filterChain:
          filter:
            name: "envoy.filters.network.http_connection_manager"
            subFilter:
              name: "envoy.filters.http.router"
    patch:
      operation: INSERT_BEFORE
      value:
        name: envoy.lua
        typed_config:
          "@type": "type.googleapis.com/envoy.extensions.filters.http.lua.v3.Lua"
          inlineCode: |
            function envoy_on_request(request_handle)
              local user_agent = request_handle:headers():get("user-agent")
              request_handle:logInfo("user_agent:".user_agent)
              if string.match(user_agent, "^.*iPhone13.*") then
                request_handle:headers():add("x-asm-prefer-tag", "version-gray")
              else
                request_handle:headers():add("x-asm-prefer-tag", "version-base")
              end
            end
          end
        function envoy_on_response(response_handle)
          end
    end

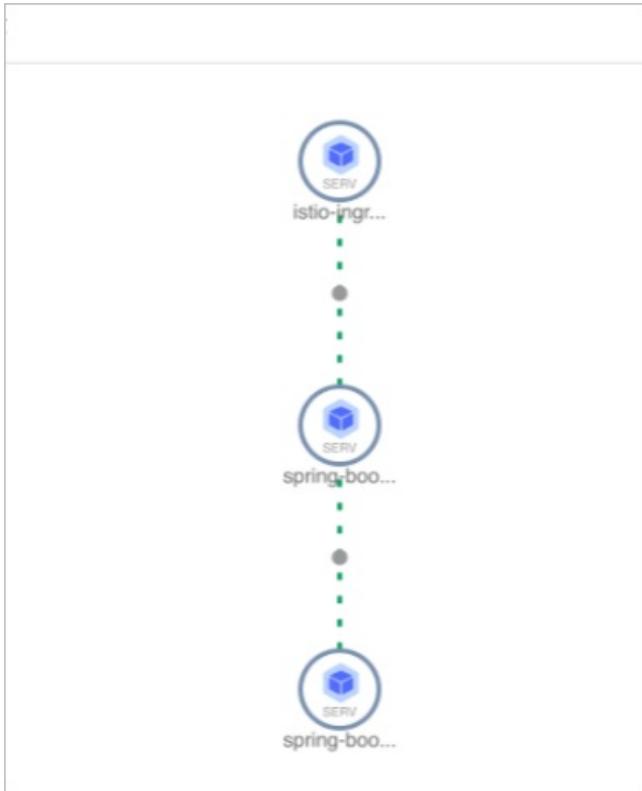
```

FAQ

Why does an end-to-end canary release not take effect?

An end-to-end canary release takes effect for an application only if the tracing feature of the application takes effect. The Spring Cloud services in this topic are connected to ARMS by using a non-intrusive manner to implement the tracing feature. If the test result does not meet your expectations, perform the following steps to check whether application monitoring is enabled:

Log on to the [Tracing Analysis console](#). In the left-side navigation pane, click **Global Topology**. On the **Global Topology** page, you can view the following trace: ingressgateway > springcloud-istio-client > springcloud-istio-server. This indicates that application monitoring is enabled.



If you enable application monitoring after you deploy the demo, you must redeploy the demo after you enable application monitoring. For more information about how to enable application monitoring, see [Monitor application performance](#).

Related information

- [Monitor application performance](#)
- [Traffic labeling and label-based routing](#)

3. Security

3.1. Implement CORS in ASM

When a client from one domain accesses a service in a different domain or a service that resides in the same domain but uses a different port from the client, the client initiates a cross-origin request. If the service disallows cross-origin resource access, the client cannot access the service. In this case, you can implement cross-origin resource sharing (CORS) to allow web application servers to access cross-origin resources. This topic describes how to configure a CORS policy in a virtual service of Alibaba Cloud Service Mesh (ASM) to implement CORS.

CORS overview

For security reasons, browsers restrict cross-origin HTTP requests that are initiated from scripts. For example, XMLHttpRequest and the Fetch API follow the same-origin policy. This means that a web application that uses these APIs can request only resources from the same origin in which the application is loaded unless the response from other origins includes valid CORS headers.

CORS is a mechanism based on HTTP headers and allows a server to identify domains, schemes, or ports other than its own from which a browser permits loading resources.

The CORS mechanism supports two types of requests: simple requests and preflight requests.

- Simple request mode:

A browser sends a cross-origin request. The Origin header is specified in the request, which indicates that the request is a cross-origin request. After the destination server receives the cross-origin request, the server determines whether to allow the request based on configured CORS rules. In response, the server returns the Access-Control-Allow-Origin and Access-Control-Allow-Methods headers to indicate whether the request is allowed.

- Preflight request mode:

A browser sends a preflight request, which is an HTTP OPTIONS request. The request is used to check whether the destination server allows cross-origin requests from the current domain. If the destination server allows cross-origin requests from the current domain, the browser sends an actual cross-origin request.

The OPTIONS request contains the following headers: Origin, Access-Control-Request-Method, and Access-Control-Request-Headers. After the destination server receives the OPTIONS request, the server specifies the Access-Control-Allow-Origin, Access-Control-Allow-Method, Access-Control-Allow-Headers, and Access-Control-Max-Age headers in the response to indicate whether the request is allowed. If the preflight request is allowed, the browser sends an actual cross-origin request.

If a request meets the following three requirements, the CORS mechanism processes the request as a simple request. Otherwise, the CORS mechanism processes the request as a preflight request.

- The request uses one of the following methods:

GET, HEAD, and POST

- The Content-Type header in the request is set to one of the following values:

text/plain, application/x-www-form-urlencoded, and multipart/form-data

- The request uses one of the following CORS-safelisted headers that are defined by the Fetch standard:

Accept, Accept-Language, Content-Language, and Content-Type. Note: The value of the Content-Type header must be set to one of the values that are listed in the second requirement.

Configure a CORS policy in a virtual service

Browsers automatically implement CORS communication. To allow cross-origin requests that are initiated to a service and implement CORS communication, you must set the `corsPolicy` field in the virtual service that is defined for the service.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings-route
spec:
  hosts:
  - ratings.prod.svc.cluster.local
  http:
  - route:
    - destination:
        host: ratings.prod.svc.cluster.local
        subset: v1
    corsPolicy:
      allowOrigins:
      - exact: https://example.com
      # - regex: *      # You can use regular expressions to specify the addresses of the origins.
      allowMethods:
      - POST
      - GET
      allowCredentials: false
      allowHeaders:
      - X-Foo-Bar
      maxAge: "24h"
```

Parameter	Description
<code>allowOrigins</code>	The addresses of the origins that are allowed to access the service. Regular expressions are supported. For requests without credentials, the server can set this parameter to a wildcard (*) so that all origins are allowed to access the service.
<code>allowMethods</code>	The HTTP methods that can be used to initiate cross-origin requests.
<code>allowHeaders</code>	The headers that can be contained to initiate cross-origin requests. The headers are used to precheck the responses to requests.
<code>exposeHeaders</code>	The whitelist of headers that the server allows browsers to access.
<code>maxAge</code>	The maximum amount of time that browsers can cache the response to a preflight request.

Parameter	Description
allowCredentials	Specifies whether credentials are required to initiate cross-origin requests. Only valid credentials can be used to initiate cross-origin requests.

3.2. Enable Multi-Buffer for TLS acceleration

Alibaba Cloud Service Mesh (ASM) Commercial Edition (Professional Edition) combines with Intel Multi-Buffer to accelerate Transport Layer Security (TLS) processing in Envoy. This topic describes how to enable Multi-Buffer for TLS acceleration.

Prerequisites

- An ASM Commercial Edition (Professional Edition) instance of version 1.10 or later is created. For more information, see [Create an ASM instance](#).
- A Container Service for Kubernetes (ACK) cluster is created, and the instance families of nodes in the cluster support the Multi-Buffer CPU model, Intel Ice Lake. For more information, see [Create an ACK managed cluster](#).

The following table describes the instance families that support Intel Ice Lake.

 **Note** For more information about instance types, see [Instance family](#).

Instance family	Description
g7	g7se, storage-enhanced general-purpose instance family
	g7, general-purpose instance family
	g7t, security-enhanced general-purpose instance family
c7	c7, compute-optimized instance family
	c7re, RDMA-enhanced instance family
	c7se, storage-enhanced compute-optimized instance family
	c7t, security-enhanced compute-optimized instance family
r7	r7p, memory-optimized instance family
	r7se, storage-enhanced memory-optimized instance family
	r7, memory-optimized instance family

Instance family	Description
	r7t, security-enhanced memory-optimized instance family
Others	re7p, high-memory instance family
	vgn7i-vws, vGPU-accelerated instance family
	gn7i, GPU-accelerated compute-optimized instance family
	ebmgn7i, GPU-accelerated compute optimized ECS Bare Metal Instance family
	sccc7, compute-optimized SCC instance family
	sccg7, general-purpose SCC instance family

-

Context

With the development of network security technologies, TLS has become the cornerstone of network communication. A TLS session is generally divided into the handshake phase and the data transmission phase. The most important task in the handshake phase is to use asymmetric encryption to negotiate a session key. In the data transmission phase, the session key is used to perform symmetric encryption on the data before data transmission.

In microservice scenarios, Envoy needs to process a large number of TLS requests, whether Envoy serves as an ingress gateway or as a proxy for microservices. Especially during the handshake phase, asymmetric encryption and decryption consume a large amount of CPU resources. This may become a bottleneck in large-scale microservice scenarios. ASM combines with Intel Multi-Buffer to accelerate TLS processing in Envoy to alleviate the bottleneck.

Multi-Buffer uses Intel CPU AVX-512 to process multiple independent buffers at the same time. In other words, multiple encryption and decryption operations can be simultaneously executed in one execution cycle, which accelerates encryption and decryption. Multi-Buffer does not need additional hardware. The CPU package must contain the AVX-512 instruction set. Alibaba Cloud has included the latest AVX-512 instruction set in the Ice Lake processor.

Procedure

You can use one of the following methods to enable the Multi-Buffer feature:

- If no ASM instances exist, select **Enable MultiBuffer-based TLS encryption and decryption performance optimization** when you create an ASM instance. For more information, see [Create an ASM instance](#).
- If an ASM instance exists, select **Enable MultiBuffer-based TLS encryption and decryption performance optimization** on the **Basic Information** page of the ASM instance. The following procedure describes how to enable Multi-Buffer if you already have an ASM instance.
 - 1.
 - 2.

- 3.
- 4.
5. In the **Settings Update** panel, select **Enable MultiBuffer-based TLS encryption and decryption performance optimization**, and then click **OK**.

If you use the general-purpose instance family g7 as the instance family of the Kubernetes nodes, the query per second (QPS) performance improves by 75% after Multi-Buffer is enabled. If you use the ECS Bare Metal Instance, a more significant performance improvement can be obtained.

FAQ

What happens if Multi-Buffer is enabled on the control plane, but the nodes in the data-plane Kubernetes cluster do not support Intel Ice Lake?

Alert logs are generated from Envoy, and Multi-Buffer does not take effect.

```

2021-11-09T15:24:03.269127Z    info    sds service generate, Multibuffer enable: true
2021-11-09T15:24:03.269158Z    info    cache returned workload trust anchor from cache          ttl=23h59m59.730845791s
2021-11-09T15:24:03.269177Z    info    proxyConfig: config_path:"/etc/istio/proxy" binary_path:"/usr/local/bin/envoy" service_cluster:"istio-ingressgateway" drain_duration:<seconds:45 > parent_shutdown_duration:<seconds:60 > discovery_address:"istiod.istio-system.svc:15012" proxy_admin_port:15000 control_plane_auth_policy:MUTUAL_TLS stat_name_length:189 concurrency:<> tracing:<zipkin:<address:"zipkin.istio-system:9411" > > proxy_metadata:<key:"DNS_AGENT" value:"" > status_port:15020 termination_drain_duration:<seconds:5 > multibuffer:<enabled:true poll_delay:<nanos:20000000 > >
2021-11-09T15:24:03.269185Z    info    sds service generate, Multibuffer enable: true
2021-11-09T15:24:03.269211Z    info    cache returned workload certificate from cache          ttl=23h59m59.730792927s
2021-11-09T15:24:03.269223Z    info    pollDelay config: 20ms
2021-11-09T15:24:03.269456Z    info    sds SDS: PUSH resource=ROOTCA
2021-11-09T15:24:03.269589Z    info    sds SDS: PUSH resource=default
2021-11-09T15:24:03.270330Z    warning envoy config gRPC config for type.googleapis.com/envoy.extensions.transport_sockets.tls.v3.Secret rejected: Multi-buffer CPU instructions not available.
2021-11-09T15:24:03.271696Z    warn    ads ADS:SDS: ACK ERROR router-172.18.96.137-istio-ingressgateway1-d7447cb55-khr8s.istio-system-istio-system.svc.cluster.local-2 Internal:Multi-buffer CPU instructions not available.
2021-11-09T15:24:04.309379Z    info    Initialization took 1.267025329s
2021-11-09T15:24:04.309416Z    info    Envoy proxy is ready
2021-11-09T15:24:04.458149Z    warning envoy config gRPC config for type.googleapis.com/envoy.config.cluster.v3.Cluster rejected: Error adding/updating cluster(s) outbound|15021||istio-ingressgateway1.istio-system.svc.cluster.local: Multi-buffer CPU instructions not available., outbound|80||istio-ingressgateway1.istio-system.svc.cluster.local: Multi-buffer CPU instructions not available., outbound|443||istio-ingressgateway1.istio-system.svc.cluster.local: Multi-buffer CPU instructions not available.

```

ASM Commercial Edition (Professional Edition) 1.10 and later can automatically determine whether TLS acceleration takes effect when TLS acceleration is enabled. If the node to which the business or gateway pod is scheduled does not support Intel Ice Lake, ASM does not deliver the corresponding acceleration configuration to the node. In this case, TLS acceleration does not take effect.

If a Kubernetes cluster does not support Multi-Buffer, how can the cluster use Multi-Buffer?

1. Add a node that supports Intel Ice Lake to the Kubernetes cluster. For more information, see [Add existing ECS instances to an ACK cluster](#).
2. Add the `multibuffer-support:true` label to the newly added node. For more information, see [Manage node labels](#).
3. Add the following content to the YAML configuration of the ASM gateway. For more information, see [Modify an ingress gateway service](#).

You can increase node affinity to ensure that gateway instances are scheduled to the newly added node that supports Multi-Buffer.

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: multibuffer-support
                operator: In
                values:
                  - true
```

4. Enable Multi-Buffer by following the preceding procedure.

After you enable Multi-Buffer, the new node can use Multi-Buffer to accelerate TLS processing.

4. Use gRPC in ASM

4.1. Design principle of the gRPC practice

Alibaba Cloud Service Mesh (ASM) allows you to manage applications that use the gRPC protocol. For example, you can develop applications and add applications to containers and ASM instances. This topic describes the design principle of the gRPC practice that ASM provides.

Communication models of gRPC

Design principle

- The practice involves the four communication models of gRPC.
- The method names and parameter names that are used in the practice do not indicate any business features. This way, you can focus on the technology.

Communication models and implementation methods

Communication model	Implementation method
Unary RPC	talk
Server streaming RPC	talkOneAnswerMore
Client streaming RPC	talkMoreAnswerOne
Bidirectional streaming RPC	talkBidirectional

Protocol Buffers definition

```

service LandingService {
  //Unary RPC
  rpc talk (TalkRequest) returns (TalkResponse) {
  }
  //Server streaming RPC
  rpc talkOneAnswerMore (TalkRequest) returns (stream TalkResponse) {
  }
  //Client streaming RPC with random & sleep
  rpc talkMoreAnswerOne (stream TalkRequest) returns (TalkResponse) {
  }
  //Bidirectional streaming RPC
  rpc talkBidirectional (stream TalkRequest) returns (stream TalkResponse) {
  }
}

```

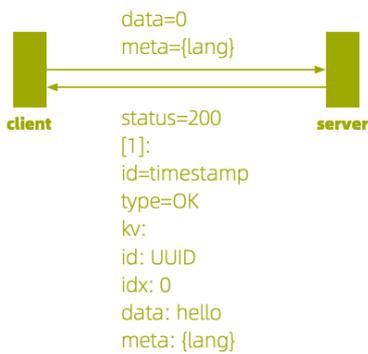
Methods

- The mainline logic of the practice is simple. For example, the data parameter in the following figure indicates a subscript of the hello array. When the gRPC server receives a request that contains the data parameter, the gRPC server returns the corresponding value in the hello array to the gRPC client.
- The practice simplifies requests and responses. The practice uses only one method to encapsulate

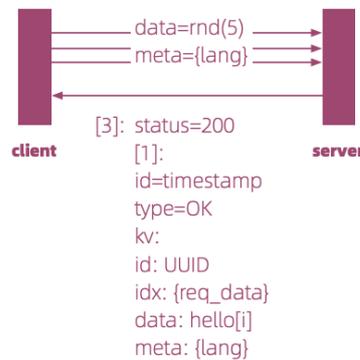
each request and response, no matter whether the request or response contains one or more messages.

- All requests are strings. If a request contains multiple messages, separate the messages with commas (,).
- All responses are arrays. If a response contains only one message, the returned array contains only one value.
- The gRPC client and server communicate with each other by using the programming language. The traffic shaping configuration is displayed in the language that is specified by the `lang` variable.

Talk Unary RPC



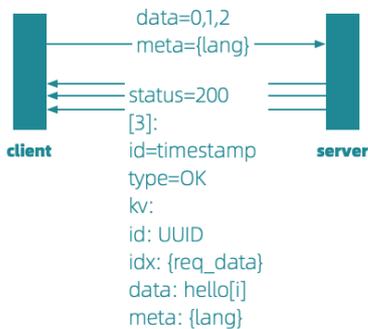
TalkMoreAnswerOne Client streaming RPC



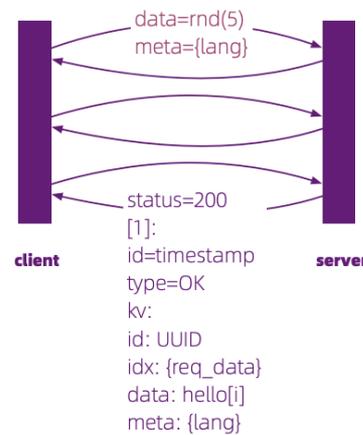
hello[]

0	Hello
1	Bonjour
2	Hola
3	こんにちは
4	Ciao
5	안녕하세요
6	你好

TalkOneAnswerMore Server streaming RPC



TalkBidirectional Bidirectional streaming RPC



Protocols

Design principle

- The practice uses simple request parameters to facilitate debugging. At the same time, the request parameters contain sufficient information.
- The response parameters in the practice support all the data types that are required for demonstration.

Request protocol

The request parameters in the practice include data and meta. All the request parameters are strings. The data parameter specifies the value of the subscript that you want to add to the hello array. The meta parameter specifies the programming language.

```
message TalkRequest {
  //language index
  string data = 1;
  //clientside language
  string meta = 2;
}
```

Response protocol

- The response parameters in the practice contain only the status parameter and the TalkResult parameter. The value of the status parameter is an integer, which indicates a status code. The value of the TalkResult parameter is an array.
- The array of the `TalkResult` parameter supports values of multiple data types, including the big integer, enumeration, and key-value pair types. The generic type of key-value pairs is string.

```
message TalkResponse {
  int32 status = 1;
  repeated TalkResult results = 2;
}
message TalkResult {
  //timestamp
  int64 id = 1;
  //enum
  ResultType type = 2;
  // result uuid
  // language index
  // data hello
  // meta serverside language (It's not good here,
  //      but ok since I feel like to keep the response)
  map<string, string> kv = 3;
}
enum ResultType {
  OK = 0;
  FAIL = 1;
}
```

Functions

Function	Description
Environment variable	The practice provides the GRC_SERVER variable for the gRPC client. In local development and debugging, the value of this variable is localhost. You must specify domain names of the gRPC services of the pod that you want to access. When the pod of the gRPC client starts, the value of the GRC_SERVER variable changes to the domain name of an active gRPC service. This way, the gRPC client can call the gRPC service.

Function	Description
Random number	For the client streaming remote procedure call (RPC) and bidirectional streaming RPC models, the gRPC client needs to call the random number function to generate a random integer value, which must be one of the subscripts of the hello array.
Timestamp	TalkResult.id is a unique identifier of the int64 type. The value is a timestamp that is generated by the timestamp function.
UUID	TalkResult.kv[id] is a unique identifier of the string type. The value is a UUID that is generated by the UUID function.
Sleep	For the models excluding unary RPC, you can call the sleep function to set the interval between requests. This way, you can better observe the sequence of messages between the gRPC client and server.

4.2. Implement the communication models of gRPC

This topic describes how to use Java, Go, Node.js, and Python to implement gRPC communication models. The models include unary remote procedure call (RPC), server streaming RPC, client streaming RPC, and client streaming RPC.

Sample project

For information about the sample project of gRPC, see [hello-servicemesh-grpc](#). The directories in this topic are directories of [hello-servicemesh-grpc](#).

Step 1: Convert code

1. Run the following command to install gRPC and Protocol Buffers. In this example, gRPC and Protocol Buffers are installed in the macOS operating system.

```
brew install grpc protobuf
```

2. Convert the Protocol Buffers definition to code in the programming languages that you use. In the topic, Java, Go, Node.js, and Python are used:

 **Note** In the sample project, the code directory of each language contains the proto directory that stores the *landing.proto* file. The *landing.proto* file is a symbolic link to the *proto/landing.proto* file in the root directory of the sample project. This way, you can update the Protocol Buffers definition in a unified manner.

- o Java: Maven is a build automation tool for Java. Maven provides the `protobuf-maven-plugin` plug-in to automatically convert code. You can run the `mvn package` command to use `protoc-gen-grpc-java` to generate gRPC template code. For more information, see [hello-grpc-java/pom.xml](#).
- o Go: Run the `go get github.com/golang/protobuf/protoc-gen-go` command to install `protoc-gen-go`. Then, run the `protoc` command to generate gRPC code. For more information, see [hello-grpc-go/proto2go.sh](#).

- Node.js: Run the `npm install -g grpc-tools` command to install `grpc_tools_node_protoc`. Then, run the `protoc` command to generate gRPC code. For more information, see *hello-grpc-nodejs/proto2js.sh*.
- Python: Run the `pip install grpcio-tools` command to install `grpcio-tools`. Then, run the `protoc` command to generate gRPC code. For more information, see *hello-grpc-python/proto2py.sh*.

Step 2: Set communication models

1. Set the hello array.

- Java:

```
private final List<String> HELLO_LIST = Arrays.asList("Hello", "Bonjour", "Hola", "こんにちは", "Ciao", "안녕하세요");
kv.put("data", HELLO_LIST.get(index));
```

- Go:

```
var helloList = []string{"Hello", "Bonjour", "Hola", "こんにちは", "Ciao", "안녕하세요"}
kv["data"] = helloList[index]
```

- Node.js:

```
let hellos = ["Hello", "Bonjour", "Hola", "こんにちは", "Ciao", "안녕하세요"]
kv.set("data", hellos[index])
```

- Python:

```
hellos = ["Hello", "Bonjour", "Hola", "こんにちは", "Ciao", "안녕하세요"]
result.kv["data"] = hellos[index]
```

2. Set the communication models.

- Set the unary RPC model.

- Java:

```
// Use the blocking stub to send a request to the server.
public TalkResponse talk(TalkRequest talkRequest) {
    return blockingStub.talk(talkRequest);
}
// After the server processes the request, the onNext and onCompleted events of the
// StreamObserver instance are triggered.
public void talk(TalkRequest request, StreamObserver<TalkResponse> responseObserver) {
    ...
    responseObserver.onNext(response);
    responseObserver.onCompleted();
}
```

- Go:

```
func talk(client pb.LandingServiceClient, request *pb.TalkRequest) {
    r, err := client.Talk(context.Background(), request)
}

func (s *ProtoServer) Talk(ctx context.Context, request *pb.TalkRequest) (*pb.TalkResponse, error) {
    return &pb.TalkResponse{
        Status: 200,
        Results: []*pb.TalkResult{s.buildResult(request.Data)},
    }, nil
}
```

- Node.js:

```
function talk(client, request) {
    client.talk(request, function (err, response) {
        ...
    })
}

function talk(call, callback) {
    const talkResult = buildResult(call.request.getData())
    ...
    callback(null, response)
}
```

- Python:

```
def talk(stub):
    response = stub.talk(request)
def talk(self, request, context):
    result = build_result(request.data)
    ...
    return response
```

- Set the server streaming RPC model.

- Java:

```
public List<TalkResponse> talkOneAnswerMore(TalkRequest request) {
    Iterator<TalkResponse> talkResponses = blockingStub.talkOneAnswerMore(request);
    talkResponses.forEachRemaining(talkResponseList::add);
    return talkResponseList;
}

public void talkOneAnswerMore(TalkRequest request, StreamObserver<TalkResponse> responseObserver) {
    String[] datas = request.getData().split(",");
    for (String data : datas) {...}
    talkResponses.forEach(responseObserver::onNext);
    responseObserver.onCompleted();
}
```

■ Go:

```
func talkOneAnswerMore(client pb.LandingServiceClient, request *pb.TalkRequest) {
    stream, err := client.TalkOneAnswerMore(context.Background(), request)
    for {
        r, err := stream.Recv()
        if err == io.EOF {
            break
        }
        ...
    }
}
func (s *ProtoServer) TalkOneAnswerMore(request *pb.TalkRequest, stream pb.Landing.
.Server) error {
    datas := strings.Split(request.Data, ",")
    for _, d := range datas {
        stream.Send(&pb.TalkResponse{...})
    }
}
```

■ Node.js:

```
function talkOneAnswerMore(client, request) {
    let call = client.talkOneAnswerMore(request)
    call.on('data', function (response) {
        ...
    })
}
function talkOneAnswerMore(call) {
    let datas = call.request.getData().split(",")
    for (const data in datas) {
        ...
        call.write(response)
    }
    call.end()
}
```

■ Python:

```
def talk_one_answer_more(stub):
    responses = stub.talkOneAnswerMore(request)
    for response in responses:
        logger.info(response)
def talkOneAnswerMore(self, request, context):
    datas = request.data.split(",")
    for data in datas:
        yield response
```

- Set the client streaming RPC model.

■ Java:

```
public void talkMoreAnswerOne(List<TalkRequest> requests) throws InterruptedException {
    final CountdownLatch finishLatch = new CountdownLatch(1);
    StreamObserver<TalkResponse> responseObserver = new StreamObserver<TalkResponse>() {
        @Override
        public void onNext(TalkResponse talkResponse) {
            log.info("Response=\n{}", talkResponse);
        }
        @Override
        public void onCompleted() {
            finishLatch.countDown();
        }
    };
    final StreamObserver<TalkRequest> requestObserver = asyncStub.talkMoreAnswerOne(
        responseObserver);
    try {
        requests.forEach(request -> {
            if (finishLatch.getCount() > 0) {
                requestObserver.onNext(request);
            }
        });
        requestObserver.onCompleted();
    }
}

public StreamObserver<TalkRequest> talkMoreAnswerOne(StreamObserver<TalkResponse> responseObserver) {
    return new StreamObserver<TalkRequest>() {
        @Override
        public void onNext(TalkRequest request) {
            talkRequests.add(request);
        }
        @Override
        public void onCompleted() {
            responseObserver.onNext(buildResponse(talkRequests));
            responseObserver.onCompleted();
        }
    };
}
```

■ Go:

```
func talkMoreAnswerOne(client pb.LandingServiceClient, requests []*pb.TalkRequest)
{
    stream, err := client.TalkMoreAnswerOne(context.Background())
    for _, request := range requests {
        stream.Send(request)
    }
    r, err := stream.CloseAndRecv()
}

func (s *ProtoServer) TalkMoreAnswerOne(stream pb.LandingService_TalkMoreAnswerOneS
erver) error {
    for {
        in, err := stream.Recv()
        if err == io.EOF {
            talkResponse := &pb.TalkResponse{
                Status: 200,
                Results: rs,
            }
            stream.SendAndClose(talkResponse)
            return nil
        }
        rs = append(rs, s.buildResult(in.Data))
    }
}
```

■ Node.js:

```
function talkMoreAnswerOne(client, requests) {
    let call = client.talkMoreAnswerOne(function (err, response) {
        ...
    })
    requests.forEach(request => {
        call.write(request)
    })
    call.end()
}

function talkMoreAnswerOne(call, callback) {
    let talkResults = []
    call.on('data', function (request) {
        talkResults.push(buildResult(request.getData()))
    })
    call.on('end', function () {
        let response = new messages.TalkResponse()
        response.setStatus(200)
        response.setResultsList(talkResults)
        callback(null, response)
    })
}
```

- Python:

```
def talk_more_answer_one(stub):
    response_summary = stub.talkMoreAnswerOne(request_iterator)
def generate_request():
    for _ in range(0, 3):
        yield request
def talkMoreAnswerOne(self, request_iterator, context):
    for request in request_iterator:
        response.results.append(build_result(request.data))
    return response
```

- Set the bidirectional streaming RPC model.

- Java:

```
public void talkBidirectional(List<TalkRequest> requests) throws InterruptedExcepti
on {
    final CountdownLatch finishLatch = new CountdownLatch(1);
    StreamObserver<TalkResponse> responseObserver = new StreamObserver<TalkResponse
>() {
        @Override
        public void onNext(TalkResponse talkResponse) {
            log.info("Response=\n{}", talkResponse);
        }
        @Override
        public void onCompleted() {
            finishLatch.countDown();
        }
    };
    final StreamObserver<TalkRequest> requestObserver = asyncStub.talkBidirectional
(responseObserver);
    try {
        requests.forEach(request -> {
            if (finishLatch.getCount() > 0) {
                requestObserver.onNext(request);
            }
            ...
            requestObserver.onCompleted();
        }
    }
    public StreamObserver<TalkRequest> talkBidirectional(StreamObserver<TalkResponse> r
esponseObserver) {
        return new StreamObserver<TalkRequest>() {
            @Override
            public void onNext(TalkRequest request) {
                responseObserver.onNext(TalkResponse.newBuilder()
                    .setStatus(200)
                    .addResults(buildResult(request.getData())).build());
            }
            @Override
            public void onCompleted() {
                responseObserver.onCompleted();
            }
        };
    }
}
```

■ Go:

```
func talkBidirectional(client pb.LandingServiceClient, requests []*pb.TalkRequest)
{
    stream, err := client.TalkBidirectional(context.Background())
    waitc := make(chan struct{})
    go func() {
        for {
            r, err := stream.Recv()
            if err == io.EOF {
                // read done.
                close(waitc)
                return
            }
        }
    }()
    for _, request := range requests {
        stream.Send(request)
    }
    stream.CloseSend()
    <-waitc
}

func (s *ProtoServer) TalkBidirectional(stream pb.LandingService_TalkBidirectionalS
erver) error {
    for {
        in, err := stream.Recv()
        if err == io.EOF {
            return nil
        }
        stream.Send(talkResponse)
    }
}
```

■ Node.js:

```
function talkBidirectional(client, requests) {
    let call = client.talkBidirectional()
    call.on('data', function (response) {
        ...
    })
    requests.forEach(request => {
        call.write(request)
    })
    call.end()
}

function talkBidirectional(call) {
    call.on('data', function (request) {
        call.write(response)
    })
    call.on('end', function () {
        call.end()
    })
}
```

- Python:

```
def talk_bidirectional(stub):
    responses = stub.talkBidirectional(request_iterator)
    for response in responses:
        logger.info(response)
def talkBidirectional(self, request_iterator, context):
    for request in request_iterator:
        yield response
```

Step 3: Implement functions

1. Implement the environment variable function.

- Java:

```
private static String getGrcServer() {
    String server = System.getenv("GRPC_SERVER");
    if (server == null) {
        return "localhost";
    }
    return server;
}
```

- Go:

```
func grpcServer() string {
    server := os.Getenv("GRPC_SERVER")
    if len(server) == 0 {
        return "localhost"
    } else {
        return server
    }
}
```

- Node.js:

```
function grpcServer() {
    let server = process.env.GRPC_SERVER;
    if (typeof server !== 'undefined' && server !== null) {
        return server
    } else {
        return "localhost"
    }
}
```

- Python:

```
def grpc_server():
    server = os.getenv("GRPC_SERVER")
    if server:
        return server
    else:
        return "localhost"
```

2. Implement the random number function.

◦ Java:

```
public static String getRandomId() {  
    return String.valueOf(random.nextInt(5));  
}
```

◦ Go:

```
func randomId(max int) string {  
    return strconv.Itoa(rand.Intn(max))  
}
```

◦ Node.js:

```
function randomId(max) {  
    return Math.floor(Math.random() * Math.floor(max)).toString()  
}
```

◦ Python:

```
def random_id(end):  
    return str(random.randint(0, end))
```

3. Implement the timestamp function.

◦ Java:

```
TalkResult.newBuilder().setId(System.nanoTime())
```

◦ Go:

```
result.Id = time.Now().UnixNano()
```

◦ Node.js:

```
result.setId(Math.round(Date.now() / 1000))
```

◦ Python:

```
result.id = int((time.time()))
```

4. Implement the UUID function.

◦ Java:

```
kv.put("id", UUID.randomUUID().toString());
```

◦ Go:

```
import (  
    "github.com/google/uuid"  
)  
kv["id"] = uuid.New().String()
```

◦ Node.js:

```
kv.set("id", uuid.v1())
```

◦ Python:

```
result.kv["id"] = str(uuid.uuid1())
```

5. Implement the sleep function.

- o Java:

```
 TimeUnit.SECONDS.sleep(1);
```

- o Go:

```
 time.Sleep(2 * time.Millisecond)
```

- o Node.js:

```
 let sleep = require('sleep')
 sleep.msleep(2)
```

- o Python:

```
 time.sleep(random.uniform(0.5, 1.5))
```

Verify the results

Feature

Run the following commands to start the gRPC server on a terminal and the gRPC client on another terminal. After you start the gRPC client and server, the gRPC client sends requests to the API operations of the four communication models.

- Java:

```
 mvn exec:java -Dexec.mainClass="org.feuyeux.grpc.server.ProtoServer"
```

```
 mvn exec:java -Dexec.mainClass="org.feuyeux.grpc.client.ProtoClient"
```

- Go:

```
 go run server.go
```

```
 go run client/proto_client.go
```

- Node.js:

```
 node proto_server.js
```

```
 node proto_client.js
```

- Python:

```
 python server/protoServer.py
```

```
 python client/protoClient.py
```

If no communication error occurs, the gRPC client and server are started.

Cross communication

Cross communication ensures that the gRPC client and server communicate with each other in the same manner, no matter what language is used by the gRPC client and server. This way, the response of a request does not vary with the language version.

1. Start the gRPC server, for example, the Java gRPC server:

```
mvn exec:java -Dexec.mainClass="org.feuyeux.grpc.server.ProtoServer"
```

2. Run the following commands to start the gRPC clients in Java, Go, Node.js, and Python:

```
mvn exec:java -Dexec.mainClass="org.feuyeux.grpc.client.ProtoClient"
```

```
go run client/proto_client.go
```

```
node proto_client.js
```

```
python client/protoClient.py
```

If no communication error occurs, cross communication is successful.

What to do next

After you verify that the gRPC client and server can communicate as expected, you can build images for the client and server.

Step 1: Build a project

Use four programming languages to build projects for the gRPC client and server.

- Java

Create JAR packages for the gRPC client and server. Then, copy the packages to the Docker directory.

```
mvn clean install -DskipTests -f server_pom
cp target/hello-grpc-java.jar ../docker/
mvn clean install -DskipTests -f client_pom
cp target/hello-grpc-java.jar ../docker/
```

- Go

The binary files that are compiled by using Go contain the configuration about the operating systems and need to be deployed in Linux. Therefore, add the following content to the binary file: Then, copy the binary files to the Docker directories.

```
env GOOS=linux GOARCH=amd64 go build -o proto_server server.go
mv proto_server ../docker/
env GOOS=linux GOARCH=amd64 go build -o proto_client client/proto_client.go
mv proto_client ../docker/
```

- NodeJS

The Node.js project must be created in a Docker image to support all kinds of C++ dependencies that are required for the runtime. Therefore, copy the file to the Docker directory.

```
cp ../hello-grpc-nodejs/proto_server.js node
cp ../hello-grpc-nodejs/package.json node
cp -R ../hello-grpc-nodejs/common node
cp -R ../proto node
cp ../hello-grpc-nodejs/*_client.js node
```

- Python

Copy the Python file to the Docker directory without compilation.

```
cp -R ../hello-grpc-python/server py
cp ../hello-grpc-python/start_server.sh py
cp -R ../proto py
cp ../hello-grpc-python/proto2py.sh py
cp -R ../hello-grpc-python/client py
cp ../hello-grpc-python/start_client.sh py
```

Step 2: Build images for the gRPC server and client

After you build the project, all the files that are required by Dockerfile are saved in the Docker directory. This section describes the major information about the Dockerfile.

- Select alpine as the basic image because its size is the smallest. In the example, the basic image of Python is python v2.7. You can change the image version as needed.
- Node.js requires the installation of C++ and the compiler Make. The Npm package needs to be installed with grpc-tools.

This example shows how to build the image of the Node.js server.

1. Create the *grpc-server-node.dockerfile* file.

```
FROM node:14.11-alpine
RUN apk add --update \
    python \
    make \
    g++ \
    && rm -rf /var/cache/apk/*
RUN npm config set registry http://registry.npmirror.com && npm install -g node-pre-gyp
COPY node/package.json .
RUN npm install --unsafe-perm
COPY node .
ENTRYPOINT ["node","proto_server.js"]
```

2. Build an image.

```
docker build -f grpc-server-node.dockerfile -t registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_node:1.0.0 .
```

A total of eight images are built.

3. Run the Push command to distribute the images to Container Registry.

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_java:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_client_java:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_go:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_client_go:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_node:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_client_node:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_python:1.0.0
```

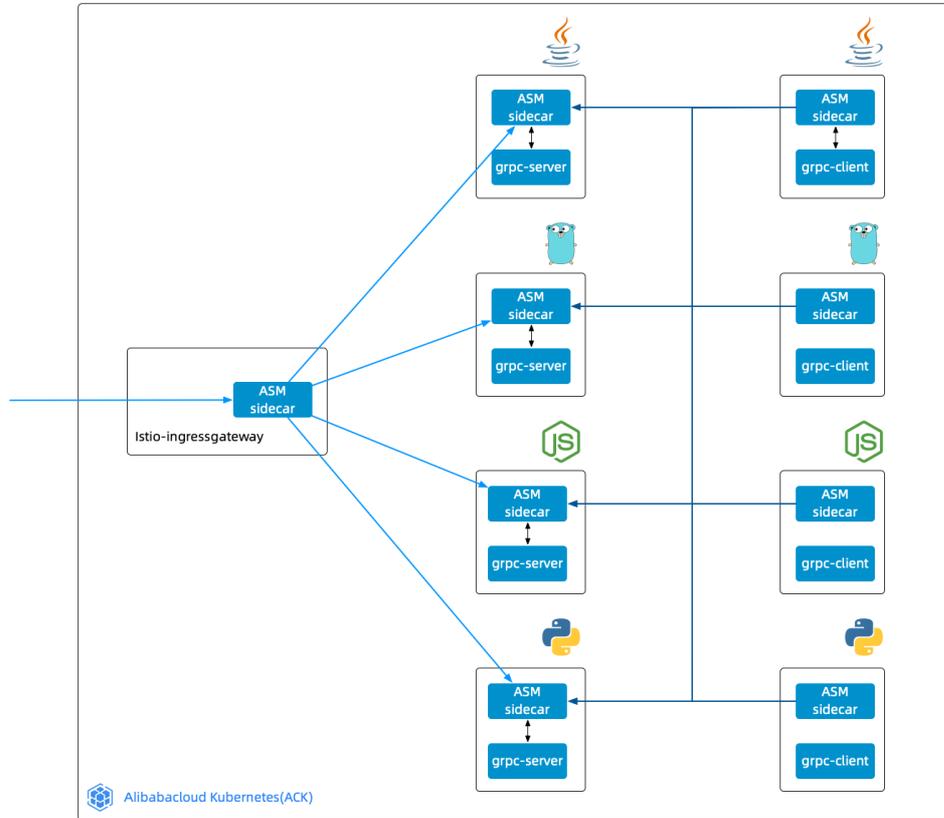
```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_client_python:1.0.0
```

4.3. Implement load balancing among gRPC servers

After gRPC clients send requests to access the `grpc-server-svc.grpc-best.svc.cluster.local` service that is specified by the `GRPC_SERVER` variable, Alibaba Cloud Service Mesh (ASM) can route the requests to gRPC servers in round robin mode. This topic describes how to deploy a gRPC service in a Container Service for Kubernetes (ACK) cluster to implement load balancing among gRPC servers. This topic also describes how to verify the load balancing of the gRPC service.

Context

In this topic, four gRPC clients and four gRPC servers in Java, Go, Node.js, and Python are used. For example, the gRPC clients call the `grpc-server-svc.grpc-best.svc.cluster.local` service that is specified by the `GRPC_SERVER` variable. When ASM receives the internal requests, ASM routes the requests to the four gRPC servers in round robin mode. In addition, you can configure an ingress gateway to route external requests to the four gRPC servers based on a load balancing policy.



Sample project

For information about the sample project of gRPC, download [hello-servicemesh-grpc](#). The directories in this topic are the directories of [hello-servicemesh-grpc](#).

Note The image repository in this topic is for reference only. Use an image script to build and push images to your self-managed image repository. For more information about the image script, see [hello-servicemesh-grpc](#).

Step 1: Create a gRPC service on the gRPC servers

In this example, a gRPC service named `grpc-server-svc` is created on all gRPC servers.

Note The value of the `spec.ports.name` parameter must start with `grpc`.

1. Create a YAML file named `grpc-server-svc`.
e

```
apiVersion: v1
kind: Service
metadata:
  namespace: grpc-best
  name: grpc-server-svc
  labels:
    app: grpc-server-svc
spec:
  ports:
    - port: 9996
      name: grpc-port
  selector:
    app: grpc-server-deploy
```

2. Run the following command to create the gRPC service:

```
kubectl apply -f grpc-server-svc.yaml
```

Step 2: Create a Deployment on each gRPC server

In this step, you must create a Deployment on each of the four gRPC servers. The following example shows you how to use the *grpc-server-node.yaml* file of a Node.js-based gRPC server to create a Deployment on the gRPC server. For more information about all the Deployments for gRPC servers in other languages, visit the *kube/deployment* page on GitHub.

 **Note** You must set the `app` label to `grpc-server-deploy` for the four Deployments on the gRPC servers to match the selector of the gRPC service that you create in Step 1. Each of the Deployments on the four gRPC servers in different languages must have a unique version label.

1. Create a YAML file named *grpc-server-node*.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: grpc-best
  name: grpc-server-node
  labels:
    app: grpc-server-deploy
    version: v3
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grpc-server-deploy
      version: v3
  template:
    metadata:
      labels:
        app: grpc-server-deploy
        version: v3
    spec:
      containers:
        - name: grpc-server-deploy
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/asm-grpc-server-node:1.0.0
          imagePullPolicy: Always
          ports:
            - containerPort: 9996
              name: grpc-port
```

2. Run the following command to create the Deployment:

```
kubectl apply -f grpc-server-node.yaml
```

Step 3: Create a Deployment on each gRPC client

The Deployments for the gRPC clients and gRPC servers are different in the following aspects:

- The gRPC servers continuously run after they are started. The gRPC clients stop running when the requests are complete. Therefore, an endless loop is required to keep client-side containers from stopping.
- You must set the `GRPC_SERVER` variable on the gRPC clients. When the pod of a gRPC client is started, the value of the `GRPC_SERVER` variable is passed to the gRPC client.

In this step, you must create a Deployment on each of the four gRPC clients. The following example shows you how to use the *grpc-client-go.yaml* file of a Go-based gRPC client to create a Deployment on the gRPC client.

1. Create a YAML file named *grpc-client-go*.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: grpc-best
  name: grpc-client-go
  labels:
    app: grpc-client-go
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grpc-client-go
  template:
    metadata:
      labels:
        app: grpc-client-go
    spec:
      containers:
        - name: grpc-client-go
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/asm-grpc-client-go:1.0.0
          command: ["/bin/sleep", "3650d"]
          env:
            - name: GRPC_SERVER
              value: "grpc-server-svc.grpc-best.svc.cluster.local"
          imagePullPolicy: Always
```

2. Run the following command to create the Deployment:

```
kubectl apply -f grpc-client-go.yaml
```

The `command: ["/bin/sleep", "3650d"]` line keeps the container running in sleep mode after the pod of the Go-based gRPC client is started. The `GRPC_SERVER` variable in `env` is set to `grpc-server-svc.grpc-best.svc.cluster.local`.

Step 4: Deploy the gRPC service and the Deployments

1. Run the following commands to create a namespace named `grpc-best` in the ACK cluster:

```
alias k="kubectl --kubeconfig $USER_CONFIG"
k create ns grpc-best
```

2. Run the following command to enable automatic sidecar injection for the namespace:

```
k label ns grpc-best istio-injection=enabled
```

3. Run the following commands to deploy the gRPC service and the eight Deployments:

```
kubectl apply -f grpc-svc.yaml
kubectl apply -f deployment/grpc-server-java.yaml
kubectl apply -f deployment/grpc-server-python.yaml
kubectl apply -f deployment/grpc-server-go.yaml
kubectl apply -f deployment/grpc-server-node.yaml
kubectl apply -f deployment/grpc-client-java.yaml
kubectl apply -f deployment/grpc-client-python.yaml
kubectl apply -f deployment/grpc-client-go.yaml
kubectl apply -f deployment/grpc-client-node.yaml
```

Verify the result

Use pods to verify the load balancing of the gRPC service

You can check load balancing among gRPC servers by sending requests to the gRPC service on the gRPC servers from the pods of the gRPC clients.

1. Run the following commands to obtain the names of the pods of the four gRPC clients:

```
client_java_pod=$(k get pod -l app=grpc-client-java -n grpc-best -o jsonpath={.items..metadata.name})
```

```
client_go_pod=$(k get pod -l app=grpc-client-go -n grpc-best -o jsonpath={.items..metadata.name})
```

```
client_node_pod=$(k get pod -l app=grpc-client-node -n grpc-best -o jsonpath={.items..metadata.name})
```

```
client_python_pod=$(k get pod -l app=grpc-client-python -n grpc-best -o jsonpath={.items..metadata.name})
```

2. Run the following commands to send requests from the pods of the gRPC clients to the gRPC service on the four gRPC servers:

```
k exec "$client_java_pod" -c grpc-client-java -n grpc-best -- java -jar /grpc-client.jar
```

```
k exec "$client_go_pod" -c grpc-client-go -n grpc-best -- ./grpc-client
```

```
k exec "$client_node_pod" -c grpc-client-node -n grpc-best -- node proto_client.js
```

```
k exec "$client_python_pod" -c grpc-client-python -n grpc-best -- sh /grpc-client/start_client.sh
```

3. Use a FOR loop to verify the load balancing among the gRPC servers. In this example, the Node.js-based gRPC client is used.

```
for ((i = 1; i <= 100; i++)); do
  kubectl exec "$client_node_pod" -c grpc-client-node -n grpc-best -- node kube_client.js > kube_result
done
sort kube_result grep -v "^[[:space:]]*$" | uniq -c | sort -nrk1
```

Expected output:

```

26 Talk:PYTHON
25 Talk:NODEJS
25 Talk:GOLANG
24 Talk:JAVA

```

The output indicates that the four gRPC servers on which the gRPC service is deployed receive an approximate number of requests. The load balancing result indicates that ASM can route external requests to the four gRPC servers on which the gRPC service is deployed based on a load balancing policy.

Use an ingress gateway to verify the load balancing of the gRPC service

You can verify load balancing among gRPC servers by using the Istio ingress gateway.

- 1.
- 2.
- 3.
- 4.
5. On the Create page, select a namespace as required, copy the following content to the code editor, and then click **Create**.

```

apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  namespace: grpc-best
  name: grpc-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 9996
        name: grpc
        protocol: GRPC
      hosts:
        - "*"

```

6. Run the following command to obtain the IP address of the Istio ingress gateway:

```

INGRESS_IP=$(k -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

```

7. Use a FOR loop to verify the load balancing among the gRPC servers.

```

docker run -d --name grpc_client_node -e GRPC_SERVER="${INGRESS_IP}" registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/asm-grpc-client-node:1.0.0 /bin/sleep 3650d
client_node_container=$(docker ps -q)
docker exec -e GRPC_SERVER="${INGRESS_IP}" -it "$client_node_container" node kube_client.js
for ((i = 1; i <= 100; i++)); do
docker exec -e GRPC_SERVER="${INGRESS_IP}" -it "$client_node_container" node kube_client.js >> kube_result
done
sort kube_result grep -v "^[[:space:]]*$" | uniq -c | sort -nrk1

```

Expected output :

```
26 Talk:PYTHON
25 Talk:NODEJS
25 Talk:GOLANG
24 Talk:JAVA
```

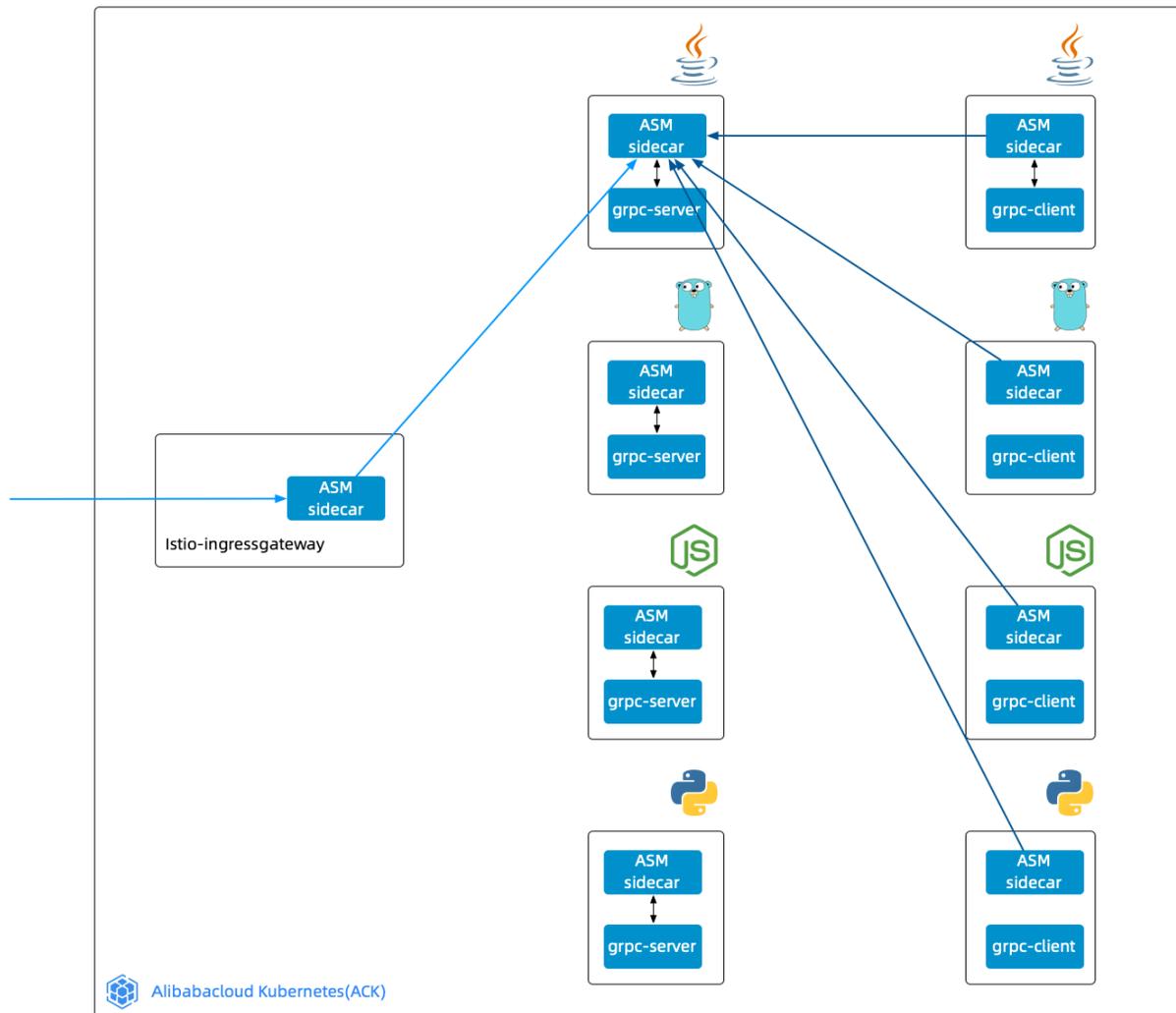
The output indicates that the four gRPC servers on which the gRPC service is deployed receive an approximate number of requests. The load balancing result indicates that ASM can route external requests to the four gRPC servers on which the gRPC service is deployed based on a load balancing policy.

4.4. Shape traffic to gRPC servers

This topic describes how to shape traffic to gRPC servers based on the gRPC version and gRPC API in the Alibaba Cloud Service Mesh (ASM) console.

Shape traffic to gRPC servers based on the gRPC version

A gRPC service is deployed on each of the Java, Go, Node.js, and Python gRPC servers. The following example shows how to route requests from gRPC clients to the gRPC service that is deployed on the Java gRPC server.



- 1.
- 2.
- 3.
4. In the **Control Plane** section, click the **DestinationRule** tab and then **Create**.
5. In the **Create** panel, select the required namespace from the Namespaces drop-down list. Copy the following content to the code editor. Then, click **OK**.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  namespace: grpc-best
  name: grpc-server-dr
spec:
  host: grpc-server-svc
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
    - name: v3
      labels:
        version: v3
    - name: v4
      labels:
        version: v4
```

6. In the **Control Plane** section, click the **VirtualService** tab and then **Create**.
7. In the **Create** panel, select the required namespace from the Namespaces drop-down list. Copy the following content to the code editor. Then, click **OK**.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  namespace: grpc-best
  name: grpc-server-vs
spec:
  hosts:
    - "*"
  gateways:
    - grpc-gateway
  http:
    - match:
        - port: 9996
      route:
        - destination:
            host: grpc-server-svc
            subset: v1
          weight: 100
```

Run the following command to check whether all the requests are routed to the Java gRPC service:

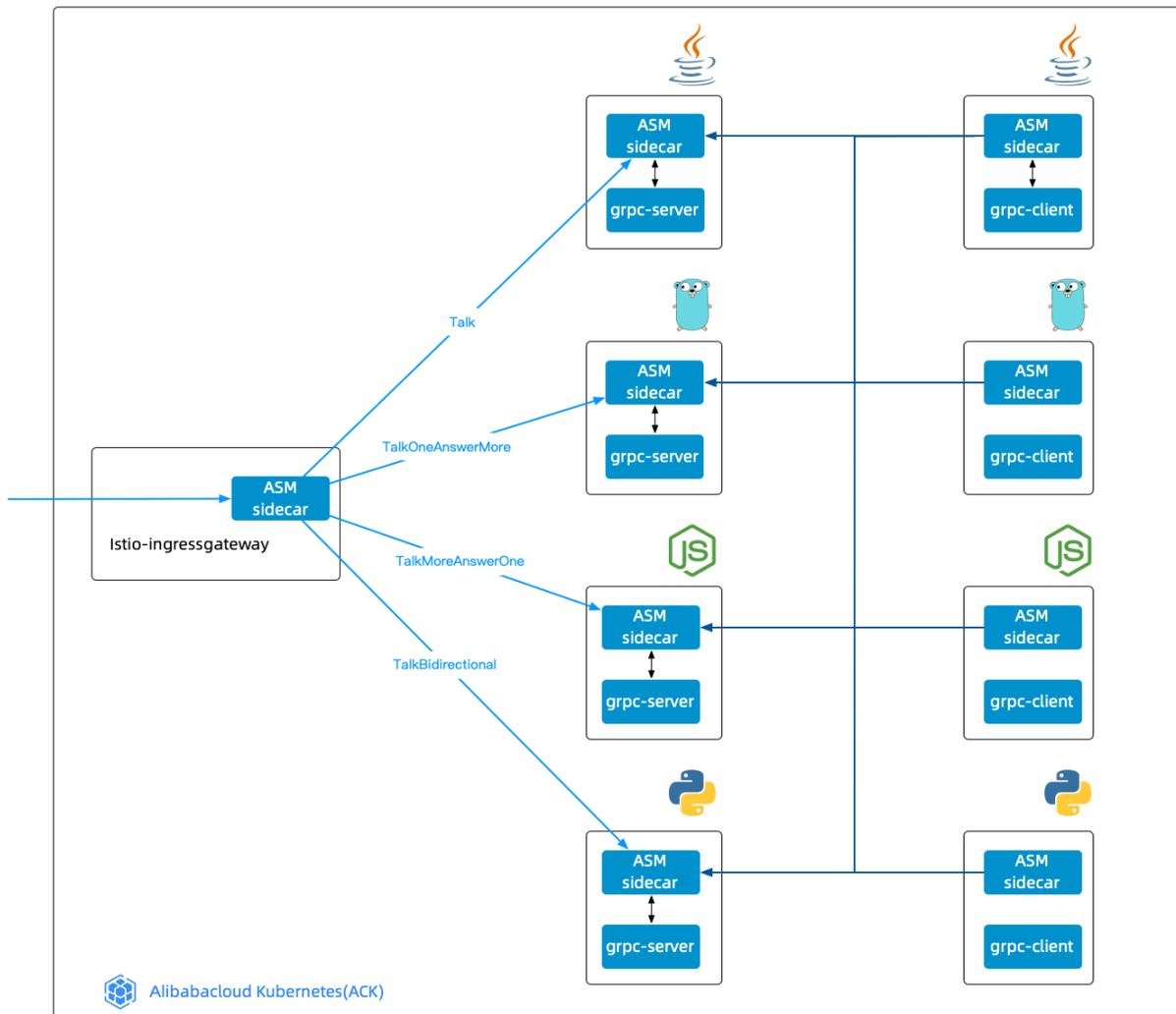
```
for i in {1..100}; do
  docker exec -e GRPC_SERVER="${INGRESS_IP}" -it "$client_node_container" node mesh_client.js >> mesh_result
done
sort mesh_result | grep -v "^[:,space:]*$" | uniq -c | sort -nrk1
```

Expected output:

```
100 TalkOneAnswerMore:JAVA
100 TalkMoreAnswerOne:JAVA
100 TalkBidirectional:JAVA
100 Talk:JAVA
```

Shape traffic to gRPC servers by using the gRPC API operations

You can use the gRPC API operations to shape traffic to gRPC servers in a fine-grained way. The gRPC API operations can be built for the communication models. For more information, see [Implement the communication models of gRPC](#). Four gRPC API operations and four gRPC services in the following programming languages are available: Java, Go, Node.js, and Python. The following example shows how to set a routing rule to route the requests of a gRPC API operation to the gRPC server that uses the same language as the operation.



- 1.
- 2.
- 3.
4. In the **Control Plane** section, click the **VirtualService** tab and then **Create**.
5. In the **Create** panel, select the required namespace from the **Namespaces** drop-down list. Copy the following content to the code editor. Then, click **OK**.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  namespace: grpc-best
  name: grpc-server-vs
spec:
  hosts:
  - "*"
  gateways:
  - grpc-gateway
  http:
  - match:
    - port: 9996
    - uri:
      exact: /org.feuyeux.grpc.LandingService/talk
    route:
    - destination:
        host: grpc-server-svc
        subset: v1
        weight: 100
  - match:
    - port: 9996
    - uri:
      exact: /org.feuyeux.grpc.LandingService/talkOneAnswerMore
    route:
    - destination:
        host: grpc-server-svc
        subset: v2
        weight: 100
  - match:
    - port: 9996
    - uri:
      exact: /org.feuyeux.grpc.LandingService/talkMoreAnswerOne
    route:
    - destination:
        host: grpc-server-svc
        subset: v3
        weight: 100
  - match:
    - port: 9996
    - uri:
      exact: /org.feuyeux.grpc.LandingService/talkBidirectional
    route:
    - destination:
        host: grpc-server-svc
        subset: v4
        weight: 100
```

Run the following command to check whether the requests of each gRPC API operation are directed to the gRPC server that uses the same language as the operation:

```
for i in {1..100}; do
  docker exec -e GRPC_SERVER="${INGRESS_IP}" -it "$client_node_container" node mesh_client.js >> mesh_result
done
sort mesh_result | grep -v "^[:space:]*$" | uniq -c | sort -nrk1
```

Expected output:

```
100 TalkOneAnswerMore:GOLANG
100 TalkMoreAnswerOne:NODEJS
100 TalkBidirectional:PYTHON
100 Talk:JAVA
```

4.5. Redirect traffic for gRPC-based applications

Alibaba Cloud Service Mesh (ASM) allows you to set matching conditions for the keys and values of headers. This way, ASM can dynamically redirect traffic based on request headers. This topic describes how to redirect the traffic of applications in ASM based on headers.

Obtain headers on the gRPC server and client

Obtain headers on the gRPC server

- Basic methods

- Use Java to implement the basic method to obtain headers on the gRPC server.

Implement the `interceptCall(ServerCall<ReqT, RespT> call, final Metadata m, ServerCallHandler<ReqT, RespT> h)` method of the `ServerInterceptor` operation. Then, run the `String v = m.get(k)` command to obtain headers on the server. The type of the input parameter of the `get()` method is `Metadata.Key<String>`.

- Use Go to implement the basic method to obtain headers on the gRPC server.

Implement the `metadata.FromIncomingContext(ctx) (md MD, ok bool)` method. The format of MD is `map[string][]string`.

- Use Node.js to implement the basic method to obtain headers on the gRPC server.

Implement the `call.metadata.getMap()` method. The type of the returned value is `[key: string]: MetadataValue`. The type of `MetadataValue` is `string/Buffer`.

- Use Python to implement the basic method to obtain headers on the gRPC server.

Implement the `context.invocation_metadata()` method. The returned value is a two-tuple array in the format of `('k', 'v')`. The key-value pair can be obtained from `m[key], m.value`.

- Unary RPC

- Use Java to implement the unary remote procedure call (RPC) method to obtain headers on the server.

The headers are intercepted.

- Use Go to implement the unary RPC method to obtain headers on the server.
Call `metadata.FromIncomingContext(ctx)` in the method. The value of the `ctx` parameter is obtained from the input parameter of the `Talk` method.
- Use Node.js to implement the unary RPC method to obtain headers on the server.
Call `call.metadata.getMap()` in the method.
- Use Python to implement the unary RPC method to obtain headers on the server.
Call `context.invocation_metadata()` in the method.
- Server streaming RPC
 - Use Java to implement the server streaming RPC method to obtain headers on the server.
The headers are intercepted.
 - Use Go to implement the server streaming RPC method to obtain headers on the server.
Call `metadata.FromIncomingContext(ctx)` in the method. You can call the `stream.Context()` method to obtain the value of the `ctx` parameter from the input parameter `stream` of the `TalkOneAnswerMore` method.
 - Use Node.js to implement the server streaming RPC method to obtain headers on the server.
Call `call.metadata.getMap()` in the method.
 - Use Python to implement the server streaming RPC method to obtain headers on the server.
Call `context.invocation_metadata()` in the method.
- Client streaming RPC
 - Use Java to implement the client streaming RPC method to obtain headers on the server.
The headers are intercepted.
 - Use Go to implement the client streaming RPC method to obtain headers on the server.
Call `metadata.FromIncomingContext(ctx)` in the method. You can call the `stream.Context()` method to obtain the value of the `ctx` parameter from the input parameter `stream` of the `TalkMoreAnswerOne` method.
 - Use Node.js to implement the client streaming RPC method to obtain headers on the server.
Call `call.metadata.getMap()` in the method.
 - Use Python to implement the client streaming RPC method to obtain headers on the server.
Call `context.invocation_metadata()` in the method.
- Bidirectional streaming RPC
 - Use Java to implement the bidirectional streaming RPC method to obtain headers on the server.
The headers are intercepted.
 - Use Go to implement the bidirectional streaming RPC method to obtain headers on the server.
Call `metadata.FromIncomingContext(ctx)` in the method. You can call the `stream.Context()` method to obtain the value of the `ctx` parameter from the input parameter `stream` of the `TalkBidirectional` method.

- Use Node.js to implement the bidirectional streaming RPC method to obtain headers on the server.
Call `call.metadata.getMap()` in the method.
- Use Python to implement the bidirectional streaming RPC method to obtain headers on the server.
Call `context.invocation_metadata()` in the method.

Send headers from the client

• Basic methods

- Use Java to implement the basic method to send headers from the client.

Implement the `interceptCall(MethodDescriptor<ReqT, RespT> m, CallOptions o, Channel c)` method of the `ClientInterceptor` operation. Implement the `start((Listener<RespT> l, Metadata h))` method of the `ClientCall<ReqT, RespT>` type. Then, run `h.put(k, v)` to send headers on the client. The type of the input parameter `k` of `put` is `Metadata.Key<String>`, and that of the input parameter `v` is `String`.

- Use Go to implement the basic method to send headers from the client.

```
metadata.AppendToOutgoingContext(ctx, kv ...) context.Context
```

- Use Node.js to implement the basic method to send headers from the client.

```
metadata=call.metadata.getMap()metadata.add(key, headers[key])
```

- Use Python to implement the basic method to send headers from the client.

Set the variable in the `metadata_dict = {}` command in the following format: `metadata_dict[c.key] = c.value`. Convert the type of data in the `metadata_dict` array to `list tuple` by using `list(metadata_dict.items())`.

• Unary RPC

- Use Java to implement the unary RPC method to send headers from the client.
The headers are intercepted.

- Use Go to implement the unary RPC method to send headers on the client.

Call `metadata.AppendToOutgoingContext(ctx, kv)` in the method.

- Use Node.js to implement the unary RPC method to send headers from the client.
Call the basic method.

- Use Python to implement the unary RPC method to send headers from the client.
Call the basic method.

• Server streaming RPC

- Use Java to implement the server streaming RPC method to send headers from the client.
The headers are intercepted.

- Use Go to implement the server streaming RPC method to send headers from the client.

Call `metadata.AppendToOutgoingContext(ctx, kv)` in the method.

- Use Node.js to implement the server streaming RPC method to send headers from the client.
Call the basic method.

- Use Python to implement the server streaming RPC method to send headers from the client.
Call the basic method.
- Client streaming RPC
 - Use Java to implement the client streaming RPC method to send headers from the client.
The headers are intercepted.
 - Use Go to implement the client streaming RPC method to send headers from the client.
Call `metadata.AppendToOutgoingContext(ctx, kv)` in the method.
 - Use Node.js to implement the client streaming RPC method to send headers from the client.
Call the basic method.
 - Use Python to implement the client streaming RPC method to send headers from the client.
Call the basic method.
- Bidirectional streaming RPC
 - Use Java to implement the bidirectional streaming RPC method to send headers from the client.
The headers are intercepted.
 - Use Go to implement the bidirectional streaming RPC method to send headers from the client.
Call `metadata.AppendToOutgoingContext(ctx, kv)` in the method.
 - Use Node.js to implement the bidirectional streaming RPC method to send headers from the client.
Call the basic method.
 - Use Python to implement the bidirectional streaming RPC method to send headers from the client.
Call the basic method.

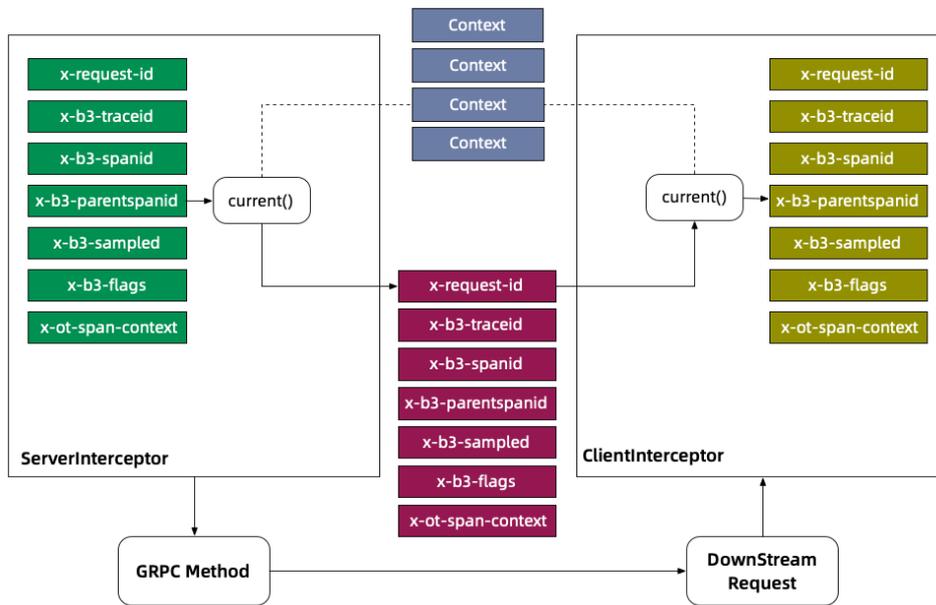
Propaganda Headers

In Tracing Analysis, upstream link metadata must be passed through to the downstream to obtain the complete information of a trace. Therefore, the tracing-related header information that is obtained on the server must be passed through to the client that sends the request to the downstream.

The operations of the communication models that are implemented by using Go, Node.js, and Python can receive headers. Therefore, the following three actions can be implemented in order by using the operations of the four communication models: First, the server reads the headers. Then, the server passes the headers. Last, the client sends the headers.

The operations of the communication models that are implemented by using Java cannot be used to propagate headers in an ordered process. This is because Java reads and writes headers by using two interceptors. Only the read interceptor obtains the unique ID of the tracing. In addition, gRPC services may receive and send requests at the same time. As a result, the two interceptors cannot be connected by using caching, which is the most intuitive method to show traces.

Java uses Metadata-Context Propagation to trace headers.



When the server interceptor reads headers, the `headers` are written into `Context` by using `ctx.withValue(key, metadata)`. The type of the key parameter is `Context.Key<String>`. Then, the client interceptor reads the `headers` from `Context` by using `key.get()`. By default, the `get` method uses `Context.current()`. This ensures that the same context is used when headers are read and written.

When headers can be propagated, you can trace the request and response messages between the gRPC client and server.

Deploy and verify the topology of an ASM instance

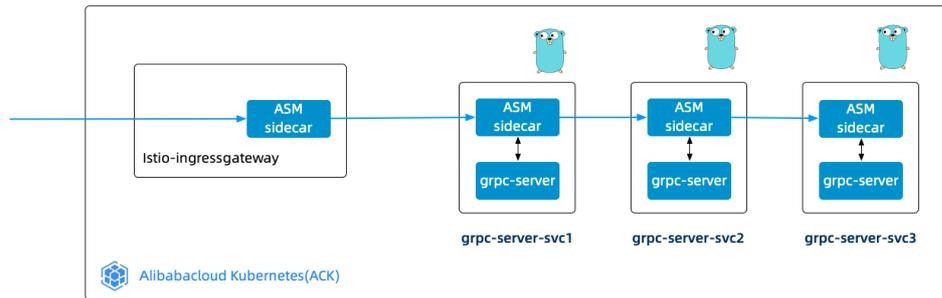
Before you can redirect traffic, you must deploy and verify the topology of the ASM instance in which your application resides. Make sure that the topology of the ASM instance works as expected.

The tracing folder of the sample project contains deployment scripts in Java, Go, Node.js, and Python. In this example, the Go deployment script is used to deploy and verify the topology of the ASM instance.

```
cd go
# Deploy the topology of the ASM instance.
sh apply.sh
# Verify the topology of the ASM instance.
sh test.sh
```

If no exceptions occur, the topology of the ASM instance works as expected.

The following figure shows the deployed topology of the ASM instance.



Redirect traffic

You can create a virtual service in ASM to set matching conditions for the keys and values of headers. This way, ASM can dynamically redirect traffic based on request headers. Furthermore, You can shape the traffic of your application in a fine-grained way based on the gRPC version and gRPC API operations. For more information, see [Shape traffic to gRPC servers](#). The following example shows you how to create a virtual service to direct all the requests of which the headers contain `server-version=go` to the Go-based gRPC server.

- 1.
- 2.
- 3.
- 4.
5. On the Create page, select a namespace as required, copy the following content to the code editor, and then click **Create**.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  namespace: grpc-best
  name: grpc-server-vs
spec:
  hosts:
  - "*"
  gateways:
  - grpc-gateway
  http:
  - match:
    - headers:
      server-version:
        exact: go
    route:
    - destination:
      host: grpc-server-svc
      subset: v2
      weight: 100
```

5. Use Flagger in ASM

5.1. Use Mixerless Telemetry to observe ASM instances

The Mixerless Telemetry technology of Alibaba Cloud Service Mesh (ASM) allows you to obtain telemetry data from containers in a non-intrusive manner. Telemetry data is collected by Prometheus Service or self-managed Prometheus as monitoring metrics. You can use the telemetry data to observe ASM instances. This topic describes how to use ASM to obtain application monitoring metrics that are collected by self-managed Prometheus to observe ASM instances.

Prerequisites

-
-
-

Step 1: Install Prometheus

1. Download and decompress the installation package of Istio. To download the installation package of Istio, go to the [Download Istio](#) page.
2. Use kubectl to connect to the ACK cluster. For more information, see [Connect to ACK clusters by using kubectl](#).
3. Run the following command to install Prometheus:

```
kubectl --kubeconfig <Path of the kubeconfig file> apply -f <Path to which the installation package of Istio is decompressed>/samples/addons/prometheus.yaml
```

Step 2: Create a service entry

- 1.
- 2.
- 3.
- 4.
5. In the **Settings Update** panel, select **Enable Prometheus** and then **Enable Self-managed Prometheus**. In the field that appears, enter a Prometheus endpoint. In this example, the default endpoint `http://prometheus:9090` is used. Then, click **OK**.

 **Note** In this example, self-managed Prometheus is used. If you use Prometheus Service, see [Monitor service meshes based on ARMS Prometheus](#).

In the left-side navigation pane of the details page, choose **Cluster & Workload Management** > . On the page that appears, you can view the created service entry.

Step 3: Configure Prometheus

1. Configure the monitoring metrics of Istio.

i

- i.
 - ii.
 - iii.
 - iv.
 - v. In the upper part of the **ConfigMap** page, select `istio-system` from the Namespace drop-down list. Find the item that is named `prometheus` and click **Edit** in the **Actions** column.
 - vi. In the **Edit** panel, enter configuration information in the **Value** field and click **OK**. To obtain the configuration information, visit [GitHub](#).
2. Delete the pod of Prometheus to make Prometheus configurations take effect.
 - i.
 - ii.
 - iii.
 - iv.
 - v. On the **Pods** page, find the pod that is named Prometheus and click **Delete** in the **Actions** column.
 - vi. In the **Delete Pod** message, click **Confirm**.
 3. Run the following command to view `job_name` in the Prometheus configurations:

```
kubectl --kubeconfig <Path of the kubeconfig file> get cm prometheus -n istio-system -o jsonpath={.data.prometheus\\.yaml} | grep job_name
```

Expected output:

```
- job_name: 'istio-mesh'  
- job_name: 'envoy-stats'  
- job_name: 'istio-policy'  
- job_name: 'istio-telemetry'  
- job_name: 'pilot'  
- job_name: 'sidecar-injector'  
- job_name: prometheus  
  job_name: kubernetes-apiservers  
  job_name: kubernetes-nodes  
  job_name: kubernetes-nodes-cadvisor  
- job_name: kubernetes-service-endpoints  
- job_name: kubernetes-service-endpoints-slow  
  job_name: prometheus-pushgateway  
- job_name: kubernetes-services  
- job_name: kubernetes-pods  
- job_name: kubernetes-pods-slow
```

Step 4: Generate monitoring data

1. Deploy the `podinfo` application in the ACK cluster.
 - i. Download the required YAML files of the `podinfo` application. For more information, visit [GitHub](#).

- ii. Run the following commands to deploy the podinfo application in the ACK cluster:

```
kubectl --kubeconfig <Path of the kubeconfig file> apply -f <Path of the podinfo application>/kustomize/deployment.yaml -n test
kubectl --kubeconfig <Path of the kubeconfig file> apply -f <Path of the podinfo application>/kustomize/service.yaml -n test
```

2. Run the following command to request the podinfo application to generate monitoring data:

```
podinfo_pod=$(k get po -n test -l app=podinfo -o jsonpath={.items..metadata.name})
for i in {1..10}; do
  kubectl --kubeconfig "$USER_CONFIG" exec $podinfo_pod -c podinfo -n test -- curl -s podinfo:9898/version
  echo
done
```

3. Check whether monitoring data is generated in the Envoy container.

- i. Run the following command to request Envoy to check whether the monitoring data of the `istio_requests_total` metric is generated:

```
kubectl --kubeconfig <Path of the kubeconfig file> exec $podinfo_pod -n test -c istio-proxy -- curl -s localhost:15090/stats/prometheus | grep istio_requests_total
```

Expected output:

```
::: istio_requests_total :::
# TYPE istio_requests_total counter
istio_requests_total{response_code="200",reporter="destination",source_workload="podinfo",source_workload_namespace="test",source_principal="spiffe://cluster.local/ns/test/sa/default",source_app="podinfo",source_version="unknown",source_cluster="c199d81d4e3104a5d90254b2a210914c8",destination_workload="podinfo",destination_workload_namespace="test",destination_principal="spiffe://cluster.local/ns/test/sa/default",destination_app="podinfo",destination_version="unknown",destination_service="podinfo.test.svc.cluster.local",destination_service_name="podinfo",destination_service_namespace="test",destination_cluster="c199d81d4e3104a5d90254b2a210914c8",request_protocol="http",response_flags="-",grpc_response_status="",connection_security_policy="mutual_tls",source_canonical_service="podinfo",destination_canonical_service="podinfo",source_canonical_revision="latest",destination_canonical_revision="latest"} 10
istio_requests_total{response_code="200",reporter="source",source_workload="podinfo",source_workload_namespace="test",source_principal="spiffe://cluster.local/ns/test/sa/default",source_app="podinfo",source_version="unknown",source_cluster="c199d81d4e3104a5d90254b2a210914c8",destination_workload="podinfo",destination_workload_namespace="test",destination_principal="spiffe://cluster.local/ns/test/sa/default",destination_app="podinfo",destination_version="unknown",destination_service="podinfo.test.svc.cluster.local",destination_service_name="podinfo",destination_service_namespace="test",destination_cluster="c199d81d4e3104a5d90254b2a210914c8",request_protocol="http",response_flags="-",grpc_response_status="",connection_security_policy="unknown",source_canonical_service="podinfo",destination_canonical_service="podinfo",source_canonical_revision="latest",destination_canonical_revision="latest"} 10
```

- ii. Run the following command to request Envoy to check whether the monitoring data of the `istio_request_duration` metric is generated:

```
kubectl --kubeconfig <Path of the kubeconfig file> exec $podinfo_pod -n test -c istio-proxy -- curl -s localhost:15090/stats/prometheus | grep istio_request_duration
```

Expected output:

```
::: istio_request_duration :::
# TYPE istio_request_duration_milliseconds histogram
istio_request_duration_milliseconds_bucket{response_code="200",reporter="destination",source_workload="podinfo",source_workload_namespace="test",source_principal="spiffe://cluster.local/ns/test/sa/default",source_app="podinfo",source_version="unknown",source_cluster="c199d81d4e3104a5d90254b2a210914c8",destination_workload="podinfo",destination_workload_namespace="test",destination_principal="spiffe://cluster.local/ns/test/sa/default",destination_app="podinfo",destination_version="unknown",destination_service="podinfo.test.svc.cluster.local",destination_service_name="podinfo",destination_service_namespace="test",destination_cluster="c199d81d4e3104a5d90254b2a210914c8",request_protocol="http",response_flags="-",grpc_response_status="",connection_security_policy="mutual_tls",source_canonical_service="podinfo",destination_canonical_service="podinfo",source_canonical_revision="latest",destination_canonical_revision="latest",le="0.5"} 10
istio_request_duration_milliseconds_bucket{response_code="200",reporter="destination",source_workload="podinfo",source_workload_namespace="test",source_principal="spiffe://cluster.local/ns/test/sa/default",source_app="podinfo",source_version="unknown",source_cluster="c199d81d4e3104a5d90254b2a210914c8",destination_workload="podinfo",destination_workload_namespace="test",destination_principal="spiffe://cluster.local/ns/test/sa/default",destination_app="podinfo",destination_version="unknown",destination_service="podinfo.test.svc.cluster.local",destination_service_name="podinfo",destination_service_namespace="test",destination_cluster="c199d81d4e3104a5d90254b2a210914c8",request_protocol="http",response_flags="-",grpc_response_status="",connection_security_policy="mutual_tls",source_canonical_service="podinfo",destination_canonical_service="podinfo",source_canonical_revision="latest",destination_canonical_revision="latest",le="1"} 10
...
```

Verify the result

1. Expose Prometheus by using a Server Load Balancer (SLB) instance. For more information, see [Manage Services](#).
- 2.
- 3.
- 4.
- 5.
6. On the **Services** page, find the service that is named Prometheus and click the IP address in the **External Endpoint** column.
7. On the Prometheus page, enter `istio_requests_total` in the search box and click **Execute**.

The following figure shows that application monitoring metrics are collected by Prometheus.

The screenshot shows the Prometheus web interface. At the top, there are navigation links: Prometheus, Alerts, Graph, Status, Help, and Classic UI. Below that, there are checkboxes for 'Enable query history' and 'Use local time'. A search bar contains the query 'istio_requests_total' and an 'Execute' button. Below the search bar, there are tabs for 'Table' and 'Graph'. The 'Table' tab is active, showing a table with two rows of data. Each row contains a long string of labels and values for the 'istio_requests_total' metric. The labels include fields like 'connection_security_policy', 'destination_app', 'destination_canonical_revision', 'destination_canonical_service', 'destination_cluster', 'destination_principal', 'destination_service', 'destination_service_namespace', 'destination_version', 'destination_workload', 'destination_workload_namespace', 'instance', 'job', 'namespace', 'pod_name', 'reporter', 'request_protocol', 'response_code', 'response_flags', 'source_app', 'source_canonical_revision', 'source_canonical_service', 'source_cluster', 'source_principal', 'source_version', and 'source_workload_namespace'. The values for these labels are specific to the environment and the request being monitored.

5.2. Use Mixerless Telemetry to scale the pods of an application

The Mixerless Telemetry technology of Alibaba Cloud Service Mesh (ASM) allows you to obtain telemetry data on containers in a non-intrusive manner. You can use Prometheus to collect the monitoring metrics of an application, such as the number of requests, the average latency of requests, and the P99 latency of requests. Then, a Horizontal Pod Autoscaler (HPA) automatically scales the pods of the application based on the collected metrics. This topic describes how to use Mixerless Telemetry to scale the pods of an application.

Prerequisites

Application monitoring metrics are collected by Prometheus. For more information, see [Use Mixerless Telemetry to observe ASM instances](#).

Step 1: Deploy a metrics adapter and a Flagger load tester

1. Use `kubectl` to connect to a Container Service for Kubernetes (ACK) cluster. For more information, see [Connect to ACK clusters by using kubectl](#).
2. Run the following command to deploy a metrics adapter:

Note To obtain the complete script of a metrics adapter, visit [GitHub](#).

```
helm --kubeconfig <Path of the kubeconfig file> -n kube-system install asm-custom-metrics \
  $KUBE_METRICS_ADAPTER_SRC/deploy/charts/kube-metrics-adapter \
  --set prometheus.url=http://prometheus.istio-system.svc:9090
```

3. Verify whether the metrics adapter is deployed as expected.

- i. Run the following command to view the pod of the metrics adapter:

```
kubectl --kubeconfig <Path of the kubeconfig file> get po -n kube-system | grep metrics-adapter
```

Expected output:

```
asm-custom-metrics-kube-metrics-adapter-6fb4949988-ht8pv 1/1 Running 0
30s
```

- ii. Run the following command to view the custom resource definitions (CRDs) of autoscaling/v2beta:

```
kubectl --kubeconfig <Path of the kubeconfig file> api-versions | grep "autoscaling/v2beta"
```

Expected output:

```
autoscaling/v2beta1
autoscaling/v2beta2
```

- iii. Run the following command to view the metrics adapter:

```
kubectl --kubeconfig <Path of the kubeconfig file> get --raw "/apis/external.metrics.k8s.io/v1beta1" | jq .
```

Expected output:

```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "external.metrics.k8s.io/v1beta1",
  "resources": []
}
```

4. Deploy a Flagger load tester.

- i. Download the required YAML files of the Flagger load tester. For more information, visit [GitHub](#).
- ii. Run the following commands to deploy the Flagger load tester:

```
kubectl --kubeconfig <Path of the kubeconfig file> apply -f <Path of the Flagger load tester>/kustomize/tester/deployment.yaml -n test
kubectl --kubeconfig <Path of the kubeconfig file> apply -f <Path of the Flagger load tester>/kustomize/tester/service.yaml -n test
```

Step 2: Create different HPAs based on your business requirements

1. Create an HPA to scale the pods of an application based on the value of the `istio_requests_total` parameter. The `istio_requests_total` parameter indicates the number of requests that are sent to the application.

- i. Use the following content to create the `requests_total_hpa.yaml` file:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo-total
  namespace: test
  annotations:
    metric-config.external.prometheus-query.prometheus/processed-requests-per-second: |
      sum(rate(istio_requests_total{destination_workload_namespace="test", reporter="destination"}[1m]))
spec:
  maxReplicas: 5
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  metrics:
  - type: External
    external:
      metric:
        name: prometheus-query
        selector:
          matchLabels:
            query-name: processed-requests-per-second
      target:
        type: AverageValue
        averageValue: "10"
```

- `annotations`: Add annotations to configure the HPA to scale the pods of the application based on the value of the `istio_requests_total` parameter.
- `target`: In this example, set the `averageValue` parameter to 10. If the average number of requests that are sent to the application is greater than or equal to 10, the HPA automatically scales out the pods of the application.

- ii. Run the following command to deploy the HPA:

```
kubectl --kubeconfig <Path of the kubeconfig file> apply -f resources_hpa/requests_total_hpa.yaml
```

iii. Verify whether the HPA is deployed as expected.

```
kubectl --kubeconfig <Path of the kubeconfig file> get --raw "/apis/external.metrics.k8s.io/v1beta1" | jq .
```

Expected output:

```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "external.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "prometheus-query",
      "singularName": "",
      "namespaced": true,
      "kind": "ExternalMetricValueList",
      "verbs": [
        "get"
      ]
    }
  ]
}
```

2. Create an HPA to scale the pods of an application based on the value of the `istio_request_duration_milliseconds_sum` parameter. The `istio_request_duration_milliseconds_sum` parameter indicates the average latency of requests that are sent to the application. Use the following content to create the `podinfo-latency-avg.yaml` file:

Repeat Substep b in Step 1 to deploy the HPA.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo-latency-avg
  namespace: test
  annotations:
    metric-config.external.prometheus-query.prometheus/latency-average: |
      sum(rate(istio_request_duration_milliseconds_sum{destination_workload_namespace="
test",reporter="destination"}[1m]))
      /sum(rate(istio_request_duration_milliseconds_count{destination_workload_namespac
e="test",reporter="destination"}[1m]))
spec:
  maxReplicas: 5
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  metrics:
  - type: External
    external:
      metric:
        name: prometheus-query
        selector:
          matchLabels:
            query-name: latency-average
      target:
        type: AverageValue
        averageValue: "0.005"
```

- annotations: Add annotations to configure the HPA to scale the pods of the application based on the value of the `istio_request_duration_milliseconds_sum` parameter.
 - target: In this example, set the `averageValue` parameter to 0.005. If the average latency of requests that are sent to the application is greater than or equal to 0.005s, the HPA automatically scales out the pods of the application.
3. Create an HPA to scale the pods of an application based on the value of the `istio_request_duration_milliseconds_bucket` parameter. The `istio_request_duration_milliseconds_bucket` parameter indicates the P95 latency of requests that are sent to the application. Use the following content to create the `podinfo-p95.yaml` file:
- Repeat Substep b in Step 1 to deploy the HPA.

```

apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo-p95
  namespace: test
  annotations:
    metric-config.external.prometheus-query.prometheus/p95-latency: |
      histogram_quantile(0.95, sum(irate(istio_request_duration_milliseconds_bucket{destination_workload_namespace="test", destination_canonical_service="podinfo"}[5m])) by (le)
)
spec:
  maxReplicas: 5
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  metrics:
    - type: External
      external:
        metric:
          name: prometheus-query
          selector:
            matchLabels:
              query-name: p95-latency
        target:
          type: AverageValue
          averageValue: "4"

```

- annotations: Add annotations to configure the HPA to scale the pods of the application based on the value of the `istio_request_duration_milliseconds_bucket` parameter.
- target: In this example, set the `averageValue` parameter to 4. If the average P95 latency of requests that are sent to the application is greater than or equal to 4 ms, the HPA automatically scales out the pods of the application.

Verify whether the pods of an application can be scaled as expected

In this example, verify the HPA that is deployed to scale the pods of an application based on the number of requests sent to the application. Verify whether the HPA works as expected if the number of requests that are sent to the application is greater than or equal to 10.

1. Run the following command to initiate requests for 5 minutes. Set the number of requests per second to 10 and the number of concurrent requests that are processed at a time to 2.

```

alias k="kubectl --kubeconfig $USER_CONFIG"
loadtester=$(k -n test get pod -l "app=flagger-loadtester" -o jsonpath='{.items..metadata.name}')
k -n test exec -it ${loadtester} -c loadtester -- hey -z 5m -c 2 -q 10 http://podinfo:9898

```

- `-z` : the duration within which requests are initiated.
- `-c` : the number of concurrent requests that are processed at a time.
- `-q` : the number of requests per second.

- Run the following command to check whether the pods are scaled out as expected:

```
watch kubectl --kubeconfig $USER_CONFIG -n test get hpa/podinfo-total
```

Expected output:

```
Every 2.0s: kubectl --kubeconfig /Users/han/shop_config/ack_zjk -n test get hpa/podinfo
East6C16G: Tue Jan 26 18:01:30 2021
NAME          REFERENCE          TARGETS          MINPODS  MAXPODS  REPLICAS  AGE
podinfo       Deployment/podinfo  10056m/10 (avg)  1         5         2          4m45s
```

A value of 2 appears in the `REPLICAS` column, which indicates that the current number of pods of the application is 2.

- Run the following command to initiate requests for 5 minutes. Set the number of requests per second to 15 and the number of concurrent requests that are processed at a time to 2.

```
alias k="kubectl --kubeconfig $USER_CONFIG"
loadtester=$(k -n test get pod -l "app=flagger-loadtester" -o jsonpath='{.items..metadata.name}')
k -n test exec -it ${loadtester} -c loadtester -- hey -z 5m -c 2 -q 15 http://podinfo:9898
```

- Run the following command to check whether the pods are scaled out as expected:

```
watch kubectl --kubeconfig $USER_CONFIG -n test get hpa/podinfo-total
```

Expected output:

```
Every 2.0s: kubectl --kubeconfig /Users/han/shop_config/ack_zjk -n test get hpa/podinfo
East6C16G: Tue Jan 26 18:01:30 2021
NAME          REFERENCE          TARGETS          MINPODS  MAXPODS  REPLICAS  AGE
podinfo       Deployment/podinfo  10056m/10 (avg)  1         5         3          4m45s
```

A value of 3 appears in the `REPLICAS` column, which indicates that the current number of pods of the application is 3. The result shows that the pods of the application are scaled out when the number of requests that are sent to the application increases. If you decrease the number of requests that are sent to the application to a specific level, a value of 1 appears in the `REPLICAS` column. The result shows that the pods of the application are scaled in when the number of requests that are sent to the application decreases.

5.3. Use Mixerless Telemetry to implement a canary release

The Mixerless Telemetry technology of Alibaba Cloud Service Mesh (ASM) allows you to obtain telemetry data on containers in a non-intrusive manner. Telemetry data is collected by Prometheus as monitoring metrics. Flagger is a tool that automates the release process of applications. You can use Flagger to monitor the metrics that are collected by Prometheus to manage traffic in canary releases. This topic describes how to use Mixerless Telemetry to implement a canary release.

Prerequisites

Application monitoring metrics are collected by Prometheus. For more information, see [Use Mixerless](#)

Telemetry to observe ASM instances.

Procedure for implementing a canary release

1. Connect ASM to Prometheus to collect application monitoring metrics.
2. Deploy Flagger and an Istio gateway.
3. Deploy a Flagger load tester to detect traffic routing for the pods of an application in the canary release.
4. Deploy an application. In this example, the podinfo application V3.1.0 is deployed.
5. Deploy a Horizontal Pod Autoscaler (HPA) to scale out the pods of the podinfo application if the CPU utilization of the podinfo application reaches 99%.
6. Implement a canary resource to specify that the traffic routed to the podinfo application is progressively increased by a fixed percentage of 10% if the P99 latency keeps being greater than or equal to 500 ms for 30s.
7. Flagger copies the podinfo application and generates the podinfo-primary application. The podinfo application is used as the deployment of the canary release version. The podinfo-primary application is used as the deployment of the production version.
8. Update the podinfo application to V3.1.1.
9. Flagger monitors the metrics that are collected by Prometheus to manage traffic in the canary release. Flagger progressively increases the traffic routed to the podinfo application V3.1.1 by a fixed percentage of 10% if the P99 latency keeps being greater than or equal to 500 ms for 30s. In addition, the HPA scales out the pods of the podinfo application and scales in the pods of the podinfo-primary application based on the status of the canary release.

Procedure

1. Use `kubectl` to connect to a Container Service for Kubernetes (ACK) cluster. For more information, see [Connect to ACK clusters by using kubectl](#).
2. Run the following commands to deploy Flagger:

```
alias k="kubectl --kubeconfig $USER_CONFIG"
alias h="helm --kubeconfig $USER_CONFIG"
cp $MESH_CONFIG kubeconfig
k -n istio-system create secret generic istio-kubeconfig --from-file kubeconfig
k -n istio-system label secret istio-kubeconfig istio/multiCluster=true
h repo add flagger https://flagger.app
h repo update
k apply -f $FLAAGER_SRC/artifacts/flagger/crd.yaml
h upgrade -i flagger flagger/flagger --namespace=istio-system \
  --set crd.create=false \
  --set meshProvider=istio \
  --set metricsServer=http://prometheus:9090 \
  --set istio.kubeconfig.secretName=istio-kubeconfig \
  --set istio.kubeconfig.key=kubeconfig
```

3. Use `kubectl` to connect to an ASM instance. For more information, see [Use kubectl to connect to an ASM instance](#).
4. Deploy an Istio gateway.

- i. Use the following content to create the *public-gateway.yaml* file:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: public-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"

```

- ii. Run the following command to deploy the Istio gateway:

```
kubectl --kubeconfig <Path of the kubeconfig file of the ASM instance> apply -f resources_canary/public-gateway.yaml

```

5. Run the following command to deploy a Flagger load tester in the ACK cluster:

```
kubectl --kubeconfig <Path of the kubeconfig file of the ACK cluster> apply -k "https://github.com/fluxcd/flagger//kustomize/tester?ref=main"

```

6. Run the following command to deploy the podinfo application and an HPA in the ACK cluster:

```
kubectl --kubeconfig <Path of the kubeconfig file of the ACK cluster> apply -k "https://github.com/fluxcd/flagger//kustomize/podinfo?ref=main"

```

7. Deploy a canary resource in the ACK cluster.

 **Note** For more information about a canary resource, see [How it works](#).

- i. Use the following content to create the *podinfo-canary.yaml* file:

```
apiVersion: flagger.app/v1beta1
kind: Canary
metadata:
  name: podinfo
  namespace: test
spec:
  # deployment reference
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  # the maximum time in seconds for the canary deployment
  # to make progress before it is rollback (default 600s)
  progressDeadlineSeconds: 60
  # HPA reference (optional)
  autoscalerRef:
    apiVersion: autoscaling/v2beta2

```

```

specVersion: v2beta2
kind: HorizontalPodAutoscaler
name: podinfo
service:
  # service port number
  port: 9898
  # container port number or name (optional)
  targetPort: 9898
  # Istio gateways (optional)
  gateways:
  - public-gateway.istio-system.svc.cluster.local
  # Istio virtual service host names (optional)
  hosts:
  - '*'
  # Istio traffic policy (optional)
  trafficPolicy:
    tls:
      # use ISTIO_MUTUAL when mTLS is enabled
      mode: DISABLE
  # Istio retry policy (optional)
  retries:
    attempts: 3
    perTryTimeout: 1s
    retryOn: "gateway-error,connect-failure,refused-stream"
analysis:
  # schedule interval (default 60s)
  interval: 1m
  # max number of failed metric checks before rollback
  threshold: 5
  # max traffic percentage routed to canary
  # percentage (0-100)
  maxWeight: 50
  # canary increment step
  # percentage (0-100)
  stepWeight: 10
  metrics:
  - name: request-success-rate
    # minimum req success rate (non 5xx responses)
    # percentage (0-100)
    thresholdRange:
      min: 99
    interval: 1m
  - name: request-duration
    # maximum req duration P99
    # milliseconds
    thresholdRange:
      max: 500
    interval: 30s
  # testing (optional)
webhooks:
  - name: acceptance-test
    type: pre-rollout
    url: http://flagger-loadtester.test/
    timeout: 30s
  metadata:

```

```

    type: bash
    cmd: "curl -sd 'test' http://podinfo-canary:9898/token | grep token"
  - name: load-test
    url: http://flagger-loadtester.test/
    timeout: 5s
    metadata:
      cmd: "hey -z 1m -q 10 -c 2 http://podinfo-canary.test:9898/"apiVersion: f
lagger.app/v1beta1
kind: Canary
metadata:
  name: podinfo
  namespace: test
spec:
  # deployment reference
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  # the maximum time in seconds for the canary deployment
  # to make progress before it is rollback (default 600s)
  progressDeadlineSeconds: 60
  # HPA reference (optional)
  autoscalerRef:
    apiVersion: autoscaling/v2beta2
    kind: HorizontalPodAutoscaler
    name: podinfo
  service:
    # service port number
    port: 9898
    # container port number or name (optional)
    targetPort: 9898
    # Istio gateways (optional)
    gateways:
      - public-gateway.istio-system.svc.cluster.local
    # Istio virtual service host names (optional)
    hosts:
      - '*'
    # Istio traffic policy (optional)
    trafficPolicy:
      tls:
        # use ISTIO_MUTUAL when mTLS is enabled
        mode: DISABLE
    # Istio retry policy (optional)
    retries:
      attempts: 3
      perTryTimeout: 1s
      retryOn: "gateway-error,connect-failure,refused-stream"
  analysis:
    # schedule interval (default 60s)
    interval: 1m
    # max number of failed metric checks before rollback
    threshold: 5
    # max traffic percentage routed to canary
    # percentage (0-100)

```

```

maxWeight: 50
# canary increment step
# percentage (0-100)
stepWeight: 10
metrics:
- name: request-success-rate
  # minimum req success rate (non 5xx responses)
  # percentage (0-100)
  thresholdRange:
    min: 99
  interval: 1m
- name: request-duration
  # maximum req duration P99
  # milliseconds
  thresholdRange:
    max: 500
  interval: 30s
# testing (optional)
webhooks:
- name: acceptance-test
  type: pre-rollout
  url: http://flagger-loadtester.test/
  timeout: 30s
  metadata:
    type: bash
    cmd: "curl -sd 'test' http://podinfo-canary:9898/token | grep token"
- name: load-test
  url: http://flagger-loadtester.test/
  timeout: 5s
  metadata:
    cmd: "hey -z 1m -q 10 -c 2 http://podinfo-canary.test:9898/"

```

- `stepWeight` : the percentage by which the traffic routed to the application is to be progressively increased. In this example, set the value to 10.
- `max` : the value of P99 latency that triggers traffic routing.
- `interval` : the duration of the value of P99 latency that triggers traffic routing.

ii. Run the following command to deploy the canary resource:

```
kubectl --kubeconfig <Path of the kubeconfig file of the ACK cluster> apply -f resources_canary/podinfo-canary.yaml
```

8. Run the following command to update the podinfo application from V3.1.0 to V3.1.1:

```
kubectl --kubeconfig <Path of the kubeconfig file of the ACK cluster> -n test set image deployment/podinfo podinfo:stefanprodan/podinfo:3.1.1
```

Verify whether the canary release is implemented as expected

Run the following command to view the process of progressive traffic routing:

```
while true; do kubectl --kubeconfig <Path of the kubeconfig file of the ACK cluster> -n test describe canary/podinfo; sleep 10s;done
```

Expected output:

```

Events:
  Type          Reason      Age           From          Message
  ----          -
Warning        Synced      39m           flagger       podinfo-primary.test not ready: waiting for
rollout to finish: observed deployment generation less then desired generation
Normal         Synced      38m (x2 over 39m)  flagger       all the metrics providers are available!
Normal         Synced      38m           flagger       Initialization done! podinfo.test
Normal         Synced      37m           flagger       New revision detected! Scaling up podinfo.te
st
Normal         Synced      36m           flagger       Starting canary analysis for podinfo.test
Normal         Synced      36m           flagger       Pre-rollout check acceptance-test passed
Normal         Synced      36m           flagger       Advance podinfo.test canary weight 10
Normal         Synced      35m           flagger       Advance podinfo.test canary weight 20
Normal         Synced      34m           flagger       Advance podinfo.test canary weight 30
Normal         Synced      33m           flagger       Advance podinfo.test canary weight 40
Normal         Synced      29m (x4 over 32m)  flagger       (combined from similar events): Promotion co
mpleted! Scaling down podinfo.test

```

The result indicates that the traffic routed to the podinfo application V3.1.1 is progressively increased from 10% to 40%.

6. Authorize and control services in namespaces

6.1. Use an authorization policy to control service access across namespaces

By default, services can access each other across namespaces in a Kubernetes cluster. For example, services that are deployed to a namespace in a development environment can access services in a production environment. The zero-trust security system of Alibaba Cloud Service Mesh (ASM) allows you to dynamically configure authorization policies to prevent all services in one namespace from accessing services in another namespace. This helps reduce risks. This topic describes how to use an authorization policy to control service access across namespaces. The demo-frontend and demo-server namespaces are used in the example.

Prerequisites

Step 1: Enable automatic sidecar injection

You can enable automatic sidecar injection for a namespace so that you can authorize and manage services in the namespace.

1. Create a namespace named demo-frontend and a namespace named demo-server.
 - i.
 - ii.
 - iii.
 - iv.
 - v. In the **Create Namespace** panel, enter demo-frontend in the Name field, and then click **OK**.
 - vi. Repeat the preceding steps to create a namespace named demo-server.
2. Enable automatic sidecar injection for the demo-frontend and demo-server namespaces.
 - i. On the page, find the demo-frontend namespace and click **Enable Automatic Sidecar Injection** in the **Automatic Sidecar Injection** column.
 - ii. In the **Submit** message, click **OK**.
 - iii. Repeat the preceding steps to enable automatic sidecar injection for the demo-server namespace.

Step 2: Create test services

Create a service named sleep in the demo-frontend namespace and a service named httpbin in the demo-server namespace. The sleep service is used to send requests to access the httpbin service.

1. [Connect to a Container Service for Kubernetes \(ACK\) cluster by using kubectl.](#)
2. Create a service named sleep in the demo-frontend namespace.

- i. Create a `sleep.yaml` file that contains the following content:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sleep
---
apiVersion: v1
kind: Service
metadata:
  name: sleep
  labels:
    app: sleep
    service: sleep
spec:
  ports:
    - port: 80
      name: http
  selector:
    app: sleep
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sleep
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sleep
  template:
    metadata:
      labels:
        app: sleep
    spec:
      terminationGracePeriodSeconds: 0
      serviceAccountName: sleep
      containers:
        - name: sleep
          image: curlimages/curl
          command: ["/bin/sleep", "3650d"]
          imagePullPolicy: IfNotPresent
          volumeMounts:
            - mountPath: /etc/sleep/tls
              name: secret-volume
      volumes:
        - name: secret-volume
          secret:
            secretName: sleep-secret
            optional: true
---
```

- ii. Run the following command to create the sleep service:

```
kubectl apply -f sleep.yaml -n demo-frontend
```

3. Create a service named httpbin in the demo-server namespace.

- i. Create an *httpbin.yaml* file that contains the following content:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: httpbin
---
apiVersion: v1
kind: Service
metadata:
  name: httpbin
  labels:
    app: httpbin
    service: httpbin
spec:
  ports:
    - name: http
      port: 8000
      targetPort: 80
  selector:
    app: httpbin
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpbin
      version: v1
  template:
    metadata:
      labels:
        app: httpbin
        version: v1
    spec:
      serviceAccountName: httpbin
      containers:
        - image: docker.io/kennethreitz/httpbin
          imagePullPolicy: IfNotPresent
          name: httpbin
          ports:
            - containerPort: 80
```

- ii. Run the following command to create the httpbin service:

```
kubectl apply -f httpbin.yaml -n demo-server
```

4. Verify that a sidecar proxy is injected into the sleep and httpbin services.
 - i.
 - ii.
 - iii.
 - iv.
 - v. On the **Pods** page, click the pod name of the sleep service.

On the **Container** tab, a sidecar proxy named istio-proxy is displayed. This indicates that a sidecar proxy is injected into the sleep service.
 - vi. Repeat the preceding steps to verify that a sidecar proxy is injected into the httpbin service.

Step 3: Create peer authentication policies

You can create a peer authentication policy for a namespace so that you can use an authorization policy to authorize services in the namespace based on Transport Layer Security (TLS).

- 1.
- 2.
- 3.
- 4.
5. On the page, click **Create mTLS Mode**.
6. Select demo-frontend from the **Namespace** drop-down list, enter a name in the Name field, select STRICT - Strictly Enforce mTLS from the **mTLS Mode (Namespace-wide)** drop-down list, and then click **Create**.
7. Repeat the preceding steps to create a peer authentication policy for the demo-server namespace to enable mutual Transport Layer Security (mTLS) authentication.

Step 4: Verify that an authorization policy can be used to control service access across namespaces

You can create an authorization policy and modify the action parameter in the authorization policy to deny or allow access requests from services in the demo-frontend namespace to services in the demo-server namespace. This way, you can control service access across namespaces.

1. Create an authorization policy to deny access requests from the demo-frontend namespace to the demo-server namespace.
 - i.
 - ii.
 - iii.
 - iv.

- v. On the Create page, set the parameters that are described in the following table and click **Create**.

Parameter	Description
Namespace	The name of the namespace to which the authorization policy belongs. In this example, demo-server is selected.
Name	The name of the authorization policy.
Policies	The policy. In this example, RULES is selected.
Action	The action on requests that meet specified requirements. In this example, DENY is selected.
Request Source	Specifies whether to authenticate the sources of requests. Turn on Request Source , click Add Request Source to List , and then click Add Request Source . Then, select namespaces from the Request Source Domain drop-down list and set the Value parameter to demo-frontend .

2. Access the httpbin service.

- i.
- ii.
- iii.
- iv.
- v. On the **Pods** page, find the pod name of the sleep service and click **Terminal** in the **Actions** column. Then, click **Container: sleep**.
- vi. Run the following command on the terminal of the sleep container to access the httpbin service:

```
curl -I httpbin.demo-server.svc.cluster.local:8000
```

Expected output:

```
HTTP/1.1 403 Forbidden
```

The preceding output indicates that access requests to the httpbin service fail. Services in the demo-frontend namespace fail to access services in the demo-server namespace.

3. Change the value of the action parameter in the authorization policy to **ALLOW** to allow access requests from the demo-frontend namespace to the demo-server namespace.
 - i.
 - ii.
 - iii.
 - iv.

- v. On the page, find the authorization policy that you want to manage and click **YAML** in the **Actions** column.
 - vi. In the **Edit** panel, change the value of the action parameter to *ALLOW*, and then click **OK**.
4. Run the following command on the terminal of the sleep container to access the httpbin service:

```
curl -I httpbin.demo-server.svc.cluster.local:8000
```

Expected output:

```
HTTP/1.1 200 OK
```

The preceding output indicates that access requests to the httpbin service are successful. Services in the demo-frontend namespace can access services in the demo-server namespace.

To sum up, if you specify the DENY action in the authorization policy, services in the demo-frontend namespace fail to access services in the demo-server namespace. If you specify the ALLOW action in the authorization policy, services in the demo-frontend namespace can access services in the demo-server namespace. The test results indicate that an authorization policy can be used to control service access across namespaces.

6.2. Use an authorization policy to control access traffic from services in a namespace to an external database

To secure a database, you need to restrict the services that are allowed to access the database. For example, you can specify that only services in specific namespaces in a production environment are allowed to access databases in the production environment. This way, you can deny access traffic from services in a development environment to the production environment. The zero-trust security system of Alibaba Cloud Service Mesh (ASM) allows you to dynamically configure authorization policies to control access traffic from services in a namespace to an external database. This helps reduce risks. This topic describes how to use an authorization policy to control access traffic from services in a namespace to an external ApsaraDB RDS database. The demo-server namespace is used in the example.

Prerequisites

Step 1: Enable automatic sidecar injection

You can enable automatic sidecar injection for a namespace so that you can authorize and manage services in the namespace.

1. Create a namespace named demo-server.
 - i.
 - ii.
 - iii.
 - iv.
 - v. In the **Create Namespace** panel, enter demo-server in the Name field, and then click **OK**.
2. Enable automatic sidecar injection for the demo-server namespace.

- i. On the page, find the demo-server namespace and click **Enable Automatic Sidecar Injection** in the **Automatic Sidecar Injection** column.
- ii. In the **Submit** message, click **OK**.

Step 2: Create a database client

In the demo-server namespace, create a client that is used to send requests to connect to a specific external database.

1. Open a CLI on your on-premises PC and run the following command to encode the password that is used to connect to the external database in Base64:

```
echo <Database connection password> | base64
```

2. [Connect to a Container Service for Kubernetes \(ACK\) cluster by using kubectl](#).
3. Create a MySQL client in the demo-server namespace.
 - i. Create a `k8s-mysql.yaml` file that contains the following content:

```
apiVersion: v1
data:
  password: {yourPasswordBase64} # The database connection password that is encoded in Base64.
kind: Secret
metadata:
  name: mysql-pass
type: Opaque
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    name: lbl-k8s-mysql
    name: k8s-mysql
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      name: lbl-k8s-mysql
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        name: lbl-k8s-mysql
    spec:
      containers:
        - env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
```

```
      secretKeyRef:
        key: password
        name: mysql-pass
    image: 'mysql:latest'
    imagePullPolicy: Always
    name: mysql
    ports:
      - containerPort: 3306
        name: mysql
        protocol: TCP
    resources:
      limits:
        cpu: 500m
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
      - mountPath: /var/lib/mysql
        name: k8s-mysql-storage
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
    volumes:
      - emptyDir: {}
        name: k8s-mysql-storage
```

- ii. Run the following command to create the MySQL client:

```
kubectl apply -f k8s-mysql.yaml -n demo-server
```

4. Verify that a sidecar proxy is injected into the MySQL client.

- i.
- ii.
- iii.
- iv.

- v. On the **Pods** page, click the pod name of the MySQL client.

On the **Container** tab, a sidecar proxy named `istio-proxy` is displayed. This indicates that a sidecar proxy is injected into the MySQL client.

Step 3: Create an egress gateway

You can use an egress gateway to control access traffic from services in an ASM instance to an external website. After you configure an authorization policy for an egress gateway, you can also specify conditions to control whether to allow access to an external database.

- 1.
- 2.
- 3.
- 4.
5. Enter a name for the egress gateway that you want to create, select a cluster from the **Cluster**

drop-down list, and then select **North-South EgressGateway** from the **Gateway types** drop-down list. Click **Add Port** next to **Port Mapping** and set the **Protocol** parameter to **TCP** and the **Service Port** parameter to **13306**. Then, click **Create**. In this example, the name of the egress gateway is set to **egressgateway**.

Step 4: Create a peer authentication policy

You can create a peer authentication policy for a namespace so that you can use an authorization policy to authorize services in the namespace based on Transport Layer Security (TLS).

- 1.
- 2.
- 3.
- 4.
5. On the page, click **Create mTLS Mode**.
6. Select **demo-server** from the **Namespace** drop-down list, enter a name in the **Name** field, select **STRICT - Strictly Enforce mTLS** from the **mTLS Mode (Namespace-wide)** drop-down list, and then click **Create**.

Step 5: Configure a policy for accessing external services

By default, services in an ASM instance are allowed to access all external services. To control access to a specific external website, set the **External Access Policy** parameter to **REGISTRY_ONLY** for an ASM instance in the ASM console. In this case, external services that are registered as service entries cannot be accessed by services in the ASM instance.

1. Configure a policy for accessing external services.
 - i.
 - ii.
 - iii.
 - iv.
 - v. On the **Global** tab, click **External service access strategy**, set the **External Access Policy** parameter to **REGISTRY_ONLY**, and then click **Update Settings**.
2. Register the external database as a service entry.
 - i.

- ii. On the Create page, select `istio-system` from the **Namespace** drop-down list and copy the following content to the code editor. Then, click **Create**.

```
apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
  name: demo-server-rds
  namespace: demo-server
spec:
  endpoints:
    - address: rm-xxxxxxx.mysql.xxx.rds.aliyuncs.com # The address of the external database.
  ports:
    tcp: 3306
  hosts:
    - rm-xxxxxxx.mysql.xxx.rds.aliyuncs.com
  location: MESH_EXTERNAL
  ports:
    - name: tcp
      number: 3306 # The port of the external database.
      protocol: TCP # The protocol used by the external database.
  resolution: DNS
```

Step 6: Create a traffic policy

Create an Istio gateway, a destination rule, and a virtual service to route traffic from the demo-server namespace to port 13306 of the egress gateway and then to port 3306 of the external database.

1. Create an Istio gateway.

- i.
- ii.
- iii.
- iv.

- v. On the Create page, select `istio-system` from the **Namespace** drop-down list and copy the following content to the code editor. Then, click **Create**.

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: istio-egressgateway
  namespace: istio-system
spec:
  selector:
    istio: egressgateway
  servers:
  - hosts:
    - '*'
    port:
      name: http-0
      number: 13306
      protocol: TLS
    tls:
      mode: ISTIO_MUTUAL
```

Set the mode parameter to `ISTIO_MUTUAL` to enable mutual Transport Layer Security (mTLS) authentication. This means that services in an ASM instance must pass TLS authentication before they can access external websites.

2. Create a destination rule.

- i.
- ii. On the Create page, select `demo-server` from the **Namespace** drop-down list and copy the following content to the code editor. Then, click **Create**.

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: demo-server-egress-gateway
  namespace: demo-server
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
  - name: mysql-gateway-mTLS
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
      portLevelSettings:
      - port:
          number: 13306 # The port of the egress gateway.
        tls:
          mode: ISTIO_MUTUAL
          sni: rm-xxxxxxx.mysql.xxx.rds.aliyuncs.com # The host address of the external database.
```

Set the mode parameter to `ISTIO_MUTUAL` to enable mutual Transport Layer Security (mTLS) authentication. This means that external websites must pass TLS authentication before they can access services in an ASM instance.

3. Create a virtual service.

- i.
- ii. On the Create page, select `demo-server` from the **Namespace** drop-down list and copy the following content to the code editor. Then, click **Create**.

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: demo-server-through-egress-gateway
  namespace: demo-server
spec:
  exportTo:
    - istio-system
    - demo-server
  gateways:
    - mesh
    - istio-system/istio-egressgateway
  hosts:
    - rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com
  tcp:
    - match:
      - gateways:
          - mesh
        port: 3306
      route:
        - destination:
            host: istio-egressgateway.istio-system.svc.cluster.local
            port:
              number: 13306
            subset: mysql-gateway-mTLS
            weight: 100
    - match:
      - gateways:
          - istio-system/istio-egressgateway
        port: 13306
      route:
        - destination:
            host: rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com
            port:
              number: 3306
            weight: 100
```

In the `tcp` section in the preceding code, two matching rules are configured. In the first matching rule, the `gateways` parameter is set to `mesh`. This indicates that the first matching rule applies to the sidecar proxy injected into the `demo-server` namespace and is used to route traffic from the `demo-server` namespace to port 13306 of the egress gateway. In the second matching rule, the `gateways` parameter is set to `istio-system/istio-egressgateway`. This indicates that the matching rule is used to route traffic from the egress gateway to port 3306 of the registered database.

Step 7: Verify that an authorization policy can be used to control access traffic from services in the demo-server namespace to an external database

You can create an authorization policy and modify the action parameter in the authorization policy to deny or allow access traffic from services in the demo-server namespace to an external database. This way, you can control access to the external database.

1. Create an authorization policy to deny access traffic from the demo-server namespace to the external database.
 - i.
 - ii.
 - iii.
 - iv.
 - v. On the Create page, set the parameters that are described in the following table and click **Create**.

Parameter	Description
Namespace	The name of the namespace to which the authorization policy belongs. In this example, demo-server is selected.
Name	The name of the authorization policy.
Policies	The policy. In this example, RULES is selected.
Action	The action on requests that meet specified requirements. In this example, DENY is selected.
Workload Label Selection	Specifies whether to enable workload label selection. Turn on Workload Label Selection and click Add Matching Label . Then, add a label by setting the Name parameter to istio and the Value parameter to egressgateway .
Request Source	Specifies whether to authenticate the sources of requests. Turn on Request Source , click Add Request Source to List , and then click Add Request Source . Then, select namespaces from the Request Source Domain drop-down list and set the Value parameter to demo-server .

2. Access the external database.
 - i.
 - ii.
 - iii.
 - iv.

- v. On the **Pods** page, find the `k8s-mysql` container and click **Terminal** in the **Actions** column. Then, click **Container: mysql**.
- vi. Run the following command on the terminal of the `k8s-mysql` container to access the external database:

```
mysql --user=root --password=$MYSQL_ROOT_PASSWORD --host rm-xxxxxxx.mysql.xxx.rds.aliyuncs.com
```

The `ERROR 2013` error is returned, which indicates that services in the `demo-server` namespace fail to access the external database.

3. Change the value of the action parameter in the authorization policy to `ALLOW` to allow access traffic from the `demo-server` namespace to the external database.

- i.
- ii.
- iii.
- iv.

- v. On the page, find the authorization policy that you want to manage and click **YAML** in the **Actions** column.

- vi. In the **Edit** panel, change the value of the action parameter to `ALLOW`, and then click **OK**.

4. Run the following command on the terminal of the `k8s-mysql` container to access the external database:

```
mysql --user=root --password=$MYSQL_ROOT_PASSWORD --host rm-xxxxxxx.mysql.xxx.rds.aliyuncs.com
```

The `Welcome to the MySQL monitor` message is returned, which indicates that services in the `demo-server` namespace can access the external database.

The test results indicate that an authorization policy can be used to control access traffic from services in a namespace to an external database.