# Steps to implement a serviceMesh implementation

## Prerequisties

1. You have a container registry (I have ACR)
2. You have a cluster set up on Azure
3. You have established Kubectl connection with your cluster from your machine
4. You have azure cli installed in your client machine
5. You have installed helm in your client machine
6. You have installed istioctl in your client machine
7. You have deployed istio in the cluster using helm and istio-system namespace is present
8. You have created a namespace in the cluster which is istio injection enabled

## Create a Docker image

1. If Docker is installed in your system, make sure you have a docker file in the root structure,and then run
2. Run this command **docker build . -t “image:tag”**

## Push the Docker Image to ACR

1. Open cmd prompt – say  **az login** – This will start a browser and you will need to login to azure portal, make sure login is successful

* If login is successful you will see some messages in cmd prompt which says that it was successful.

1. Now login to your container registry - az acr login –name <your registry name>

* If login is success you will see – “Login succeeded” in the cmd prompt

1. Choose the image that you want to push to the container registry

* Any image that you wish to use with ACR(Azure container registry) .The image needs to be tagged with the login server address of your registry. This tag is used for routing when pushing container images to an image registry. To see the list of images – say **docker images**
* Now tag the image with the container registry path.

**docker tag <image that you want to push from docker>:<tag of image in docker> <ACR path>/<image name on ACR>:<tag on ACR>**

* You can see the ACR path by logging in to the azure portal and checking the container registry resource.
* **Docker push imagename**

1. Verify that the image was pushed – run **acr repository list --name <name of your registry> --output table**
2. Check the tag just to reverify - **az acr repository show-tags --name <name of your registry> --repository <image name> --output table**

## Deploy to K8 Cluster

1. Create a namespace on K8 cluster– any name using **kubectl create namespace <namespace>**
2. Label the namespace to inject istio components – so that we can use service mesh with istio

* **kubectl label namespace <namespace> istio-injection=enabled**

1. Write your deployment and service config for the application

**#TODO write the deployment and service config here**

1. Deploy the application to K8 cluster using – **Kubectl apply -f <path to your deployment yaml> --namespace <the namespace which was created in the last step>**
2. Check if the pod was created – **kubectl get pods –namespace <namespace where pod was deployed>**
3. If you say **Kubectl describe pod <pod name>** **–namespace <namespace where pod was deployed>** then you will see there are 2 containers running within the pod – one is your application and other is a side car injected by istio
4. Check if service was also created – **kubectl get services –namespace <namespace where pod was deployed>** - you shall see the service you created

## Expose the service to outside world using Istio Ingress Gateway and Virtual Service

### IngressGateway

* A Gateway is a component at the edge of the service mesh that receives inbound or outbound HTTP and TCP traffic.
* Is deployed to Cluster within a specific namespace , through a config file

### VirtualService

* VirtualService holds the re routing rules from host to destination
* In this example the host would be the Gateway while the destination would be the Service of the application

### Create IngressGateway

* Create a config file to provide specifications of the gateway
* Apply the Gateway config file to Cluster using

**kubectl apply -f <path of your yaml> --namespace <isito enabled namespace>**

### Create VirtualService

* Create a config file to provide specifications of the virtual Service, it would include the host and the destination and the port number at which the destination is listening
* Apply the Gateway config file to Cluster using

**kubectl apply -f <path of your yaml> --namespace <isito enabled namespace>**

### Test Your Service from external world

* Get the external Ip of the ingress gateway

**kubectl get service istio-ingressgateway --namespace istio-system -o jsonpath='{.status.loadBalancer.ingress[0].ip}'**

* Hit the endpoint of your app and you will see the result

## Canary Deployment

## What is Canary Deployment

* Imagine a new version of your app has been built and you want to move to the new version of app gradually , you do not want any downtime, you want a set of specific users only to access the new version of your app, Canary deployment is the way
* This can be achieved with a combination of rules in virtual service.

1. Create a new version of your application – change the response of one of the apis, clean build and test it from your local machine.
2. Create a docker image of your application with a new tag – v2
3. Tag the new docker image to get it pushed to the ACR, add the new version add the end
4. Push the docker image to container registry
5. Create a deployment config for your file to deploy to the cluster , service was already created for the app during the first deployment , hence makes sure that you do not try to create a new service for it.
6. Make sure the template.metadata.lables has the same key value pair as it had in the first deployment. Make sure the metada-name value is different and version value is different then the 1st deployment.
7. This will help service in targeting all the pods with the same lables
8. After deployment is complete, the initially created service will start to target both the pods
9. Now if you hit the same end point as you were earlier, you will see that Kubernetes routes the request to both the pods on round robin basis.

The Istio Service Mesh provides the following functionalities:

1. Routing. For example 90% of the traffic goes to the version 1 of a microservice and the remaining 10% goes to the version 2. Or some specific requests go to the version 1 and all the others to the version 2, according to some condition. And also: a) rewrite b) redirect
2. Support for microservices development, deployment and testing: a) timeouts b) retries c) circuit breakers d) load balancing e) fault injection for testing
3. Reporting: Logging, Distributed Tracing, Telemetry
4. Policy enforcement
5. Secure communication between micro services and strong identity.

Pilot is responsible for the items 1 and 2. Mixer is responsible for the items 3 and 4. Citadel (previously CA, previously Auth) is responsible for the item 5.

Envoy, the sidecar proxy, gets its routing and configuration tables from Pilot to implement the items 1 and 2. Envoy reports to Mixer about each request, to implement the item 3. Envoy asks Mixer to allow or forbid requests, to implement the item 4. Envoy gets certificates from Citadel to implement the item 5.