Getting Started With Isio

Introduction

This lab will get you started with istio showing you some of it’s core features and and the motivation for using it.

A prerequisite for this lab is a basic understanding of kubernetes

# Lab 1. Setup and Requirements

## Task 1. environment setup

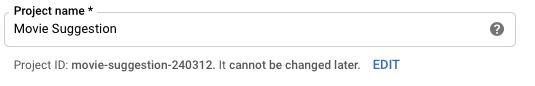
1. If you don't already have a Google Account , you can easily [create one](https://accounts.google.com/SignUp). Sign-in to Google Cloud Platform console ([console.cloud.google.com](http://console.cloud.google.com/))



2. From the popup window on the right side click the new project:



3. Type Movie Suggestion as the name of the project,and click create



4. Next, you'll need to [enable billing](https://console.cloud.google.com/billing) in the Cloud Console in order to use Google Cloud resources.

New users of Google Cloud Platform are eligible for a [$300 free trial](https://console.developers.google.com/billing/freetrial?hl=en).

5.Choose your project

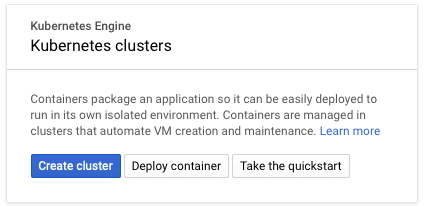


## Task 2. create the kubernetes cluster

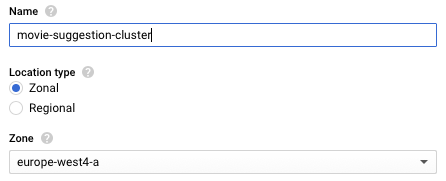
1. On the left side bar choose kubernetes engine, and then clusters



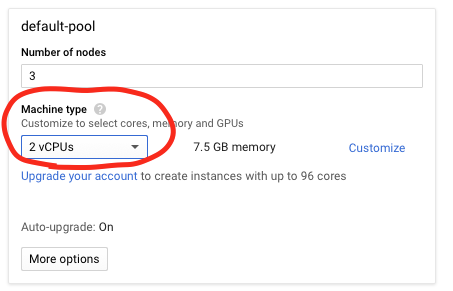
2. Click create cluster:



3. Name the cluster movie-suggestion-cluster, you can also change the zone to your nearest.

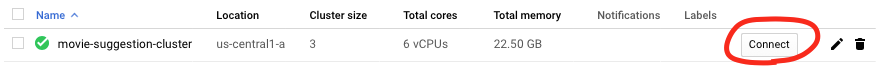


4. Change the machine type to n1-standard-2 and leave the 3 node selection as is

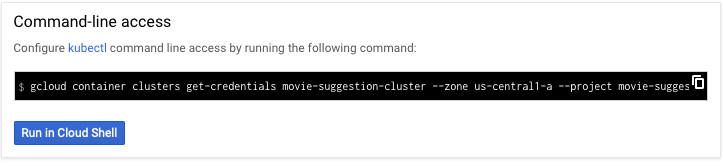


5. Click create and wait for the cluster to be created , it could take a few minutes.

6. Once the cluster is created Connect using the button as shown below



7. Click the “run in cloud shell” button and click enter when the cloud shell opens to execute the inserted command!



[Google Cloud Shell is a command line environment running in the Cloud](https://cloud.google.com/developer-shell/#how_do_i_get_started). This Debian-based virtual machine is loaded with all the development tools you’ll need (docker, gcloud, kubectl and others) and offers a persistent 5GB home directory. Open the Google Cloud Shell by clicking the button on the right:

## 

## 

# 

# 

# 

# 

# 

# Lab 2. Install istio on kubernetes

1. First we need to grant cluster administrator (admin) permissions to the current user. To create the necessary RBAC rules for Istio, the current user requires admin permissions.

Copy and execute this command in the cloud shell

|  |
| --- |
| $ **kubectl create clusterrolebinding cluster-admin-binding \**  **--clusterrole=cluster-admin \**  **--user=$(gcloud config get-value core/account)** |

1. Download the istio release and add it to the path by executing the following:

|  |
| --- |
| $ **cd ~/**  $ **curl -L https://git.io/getLatestIstio | ISTIO\_VERSION=1.1.5 sh -**  $ **export PATH=~/istio-1.1.5/bin:$PATH**  $ **echo 'export PATH=~/istio-1.1.5/bin:$PATH' >> ~/.bashrc** |

1. Install all the Istio [Custom Resource Definitions](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#customresourcedefinitions) (CRDs) by running kubectl apply as shown below, and wait a few seconds for the CRDs to be committed in the Kubernetes API-server:

|  |
| --- |
| $ **for i in istio-1.1.5/install/kubernetes/helm/istio-init/files/crd\*yaml; do kubectl apply -f $i; done**  $ **kubectl apply -f istio-1.1.5/install/kubernetes/istio-demo.yaml** |

1. Wait until all of the Istio components have the Running or Completed status.

|  |
| --- |
| $ **watch kubectl get pods -n istio-system** |

When ready press Control+c

1. To start using Istio functionality, no application changes are needed. Envoy sidecars will be injected automatically into each pod belonging to a service.

in order for istio to inject the sidecar envoy proxy to each and every pod as soon as it is

being added to the default namespace, we need to run the following command:

|  |
| --- |
| $ **kubectl label namespace default istio-injection=enabled** |

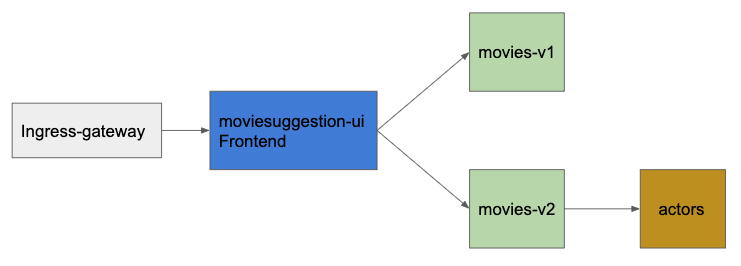
|  |
| --- |
| Please note that if you don’t want all the pods to join istio service mesh automatically like we are doing here,you can use istioctl .For example you can just run istioctl on the relevant deployment to affect the pods there, and not the rest of the pods in the cluster:  istioctl kube-inject -f <your-app-spec>.yaml | kubectl apply -f - |

# Lab 3. Deploy the sample app

1. We will start by Installing the movie suggestion app into the cluster.

the movie suggestion is a simple app that randomly returns a single popular movie. It has three microservices:

* 1. moviesuggestion-ui - the frontend microservice
  2. movies-v1 - returns a random movie with description from a popular movies list
  3. movies-v2 - returns a random movie with description from a popular movies list plus all the starring actors in the movie
  4. actors- returns a list of actors belonging to a specific movie



1. install the moviesuggestion sample app:

|  |
| --- |
| $ **cd ~/**  $ **git clone** [**https://github.com/code-sparks/examples.git**](https://github.com/code-sparks/examples.git)  **$ cd ~/examples/MovieSuggestion**  **$ kubectl apply -f kubernetes/movies-suggestion.yaml** |

2. Check that all components have the Running status, and that Ready column shows 2/2.

|  |
| --- |
| $ **watch kubectl get pods** |

3. When ready press Control+c

Please note that the number of container per pods are 2 this is due to the fact that istio injects the envoy proxy for each and every pods that is being added to the default namespace

# Lab 4. Traffic Management using Virtual Services

## Task 1. Access The Cluster From the public internet

So now after the application is deployed to the cluster we need a way to access it from the outside.

By default, any service running inside the service mesh is not automatically exposed outside from the cluster which means that we can’t get to it from the public Internet. Similarly, services within the mesh shouldn’t have access to anything running outside of the cluster either.

(although the default behaviour from istio version 1.1 is that services have access to resources outside the cluster, this behaviour can be changed by configuring istio policy to mode **REGISTRY\_ONLY**)

To allow incoming traffic to the frontend service “moviesuggestion-ui” that runs inside the cluster, we need to create an external load balancer first. As part of the istio installation, Istio creates an istio-ingressgateway service that is of type LoadBalancer and, with the corresponding Istio Gateway resource, can be used to allow traffic to the cluster.

1. To find the external ip for the cluster we should run the following commands:

|  |
| --- |
| **$ export INGRESS\_HOST=$(kubectl -n istio-system get svc istio-ingressgateway \**  **-o jsonpath='{.status.loadBalancer.ingress[0].ip}')**  **$ echo $INGRESS\_HOST** |

2. If we try to connect using the external ip - we get an error...

|  |
| --- |
| $ curl http://$**INGRESS\_HOST**  url: (7) Failed to connect to … port 80: Connection refused |

This happens because although we have an exposed external ip address , the ingress service still needs to get the configuration for how to route the traffic that enters the cluster. this configuration could be achieved using the **Gateway** resource.

Gateway describes a load balancer operating at the edge of the mesh receiving incoming or outgoing HTTP/TCP connections.

3. Add the Gateway and try to connect again ,now we get 404… error

|  |
| --- |
| $ kubectl apply -f istio-rules/moviesuggestion-gateway.yaml  $ curl -v http://$**INGRESS\_HOST** |

The reason for that is because although the gateway was defined we still need to bind it to a VirtualService**,Remember Gateway defines what is allowed to enter the cluster and VirtualService configures how to route requests in the cluster**

A **VirtualService** defines the rules that control how requests for a service are routed within an Istio service mesh

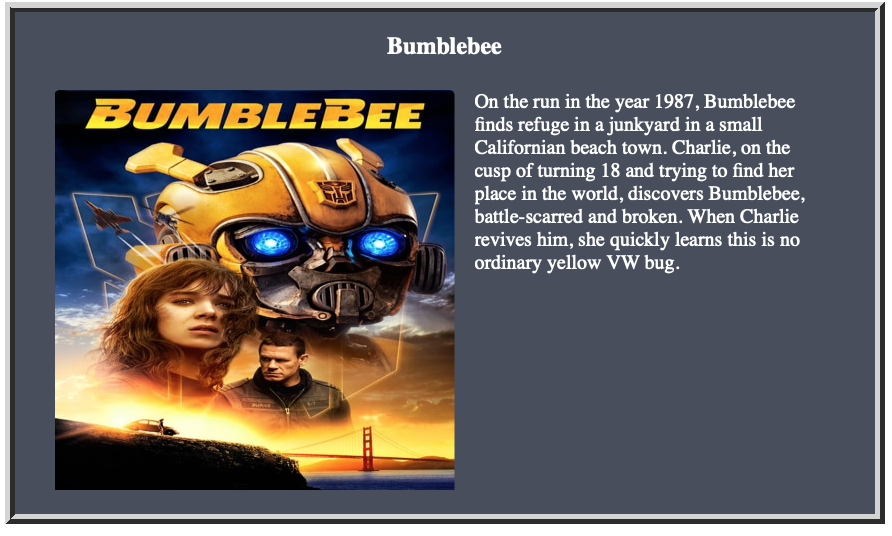
For example, a virtual service could route requests to different versions of a service or to a completely different service than was requested. Requests can be routed based on the request source and destination, HTTP paths and header fields, and weights associated with individual service versions.

4. Lets add the Virtual service

|  |
| --- |
| $ kubectl apply -f istio-rules/moviesuggestion-ui-vs.yaml |

5. Open the browser at http://<your $**INGRESS\_HOST** ip>/movie/suggestion

You should see the movie suggestion page ,something like that:



6. For making it easier to access , we can also configure the virtual service to rewrite the long url to just using the host/ip

|  |
| --- |
| $ kubectl apply -f istio-rules/moviesuggestion-ui-vs-rewrite.yaml |

7. access it by opening in your browser http://<your $**INGRESS\_HOST** ip >

## 

## Task 2 . Traffic Splitting

The Movie Suggestion sample deploys two versions of the **movies** microservice. When you access the application several times, you should notice that 50% of the time the page contains the starring actors and the other 50% of the time it does not.

This is because the movies microservice has two versions and by default the load balancer split the traffic in a round robin way .

You can control the policy for the load balancing and also add weight on a specific version to have more control for where the requests are being routed to.

For example, the following rule will route 25% of the traffic for the ***movies*** service to instances with the “v2” label and the remaining 75% of traffic to “v1”:

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: movies

spec:

hosts:

- movies

http:

- route:

- destination:

host: movies

subset: v1

weight: 75

- destination:

host: movies

subset: v2

weight: 25

The hosts field in the virtual service destination identify the requested hosts, the route destination may be the same hostname requested or even a different one

In case of a service having two or more versions , the route section in the virtual service enables us to control the way traffic will be routed to each version

1. Run the above virtual service configuration in the cloud shell:

|  |
| --- |
| $ kubectl apply -f istio-rules/movies-trafficsplit-vs.yaml |

2. open your browser now at http://<your $**INGRESS\_HOST** ip >/movie/suggestion

you will should see 503 error

This is because when we define a subset in the virtualservice routing configuration we need to bind it to a subset field in a **DestinationRule.**

A **DestinationRule** configures the set of policies to be applied to a request after **VirtualService** routing has occurred. They are intended to be authored by service owners, describing the circuit breakers, load balancer settings, TLS settings, and other settings.A **DestinationRule** also defines addressable subsets, meaning named versions, of the corresponding destination host

**For Example:**

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

**kind: DestinationRule**

**metadata:**

**name: movies**

**spec:**

**host: movies**

**trafficPolicy:**

**loadBalancer:**

**simple: RANDOM**

**subsets:**

**- name: v1**

**labels:**

**version: v1**

**- name: v2**

**labels:**

**version: v2**

Please Note that the v1 and v2 subsets are also defined in the virtualservice

3. run the following command to add the DestinationRule

|  |
| --- |
| $ kubectl apply -f istio-rules/movies-trafficsplit-dr.yaml |

4. open your browser at http://<your $**INGRESS\_HOST** ip >/movie/suggestion

refresh several times and you should see movies version v1 75% of the time and movies v2 (that also calls the actors service) only 25% of the time



## Task 3. Fault Injections

Often we wish to test the resiliency of our microservices, and the way they behave when a certain problematic conditions are met, for example , let’s say there is a network failure when microservice A calls microservice B. how does microservice A handles it ? is it fault tolerant as expected? How can we test it?

Istio provides us with handful of configurations that can be added to a VirtualService to achieve that:

For example, the following rule delays by 5 seconds all requests from the *moviesuggestion-ui* service to the *movie* service “v1” and then aborts 50% of them:

1. Run The Following command and in your cloud shell

|  |
| --- |
| $ kubectl apply -f istio-rules/movies-faultinjection.yaml |

2. Open your browser at http://<your $**INGRESS\_HOST** ip >/movie/suggestion

and reload a couple of time , you should now see that 50% of the time, you are getting an error

## 

## 

## 

## 

## 

## 

## 

## Task 4. Handling failures

Envoy proxy provides a set of out-of-the-box *opt-in* failure recovery features like (timeouts, retries, circuit breakers etc..) that can be taken advantage of by the services in an application.

Istio’s traffic management rules allow you to set defaults for failure recovery per service and version that apply to all callers or just to a specific ones.

In the following task we will examine how we can retry on failure for an upstream service that sometimes responds with an http 409 error code (actually for the purpose of the feature demonstration, the service was written in a way that it will fail around 50 percent of the time).

1. lets add another deployment for a third version of the movie service.this version of the movies microservice fails around 50% of the time

|  |
| --- |
| $ kubectl apply -f kubernetes/movies-v3.yaml |

2. Wait for the pods to become ready:

|  |
| --- |
| $ **watch kubectl get pods** |

3. When ready press Control+c

4. update the Virtual service and the destination rule to point to the movies-v3 service only

|  |
| --- |
| $ kubectl apply -f istio-rules/movies-v3-vs.yaml  $ kubectl apply -f istio-rules/movies-v3-dr.yaml |

5. Open the browser at http://<your $**INGRESS\_HOST** ip >/movie/suggestion

and reload the page a couple of times, you should see that the requests are failing 50% of the time.

6. by updating the movies VirtualService to retry 2 times in case of a 4xx error response, we should not see any failure now.

|  |
| --- |
| $ kubectl apply -f istio-rules/movies-v3-vs-with-retry.yaml |

7. Restore the movies configuration v1 and v2 traffic split

|  |
| --- |
| $ kubectl apply -f istio-rules/movies-trafficsplit-vs.yaml  $ kubectl apply -f istio-rules/movies-trafficsplit-dr.yaml |

|  |
| --- |
| NOTE: consumers of a service can also override timeout and retry defaults by providing request-level overrides through special HTTP headers. With the Envoy proxy implementation, the headers are x-envoy-upstream-rq-timeout-ms and x-envoy-max-retries, respectively. |

# Lab 5. Monitoring and observing our microservices

## Task 1 . Monitoring

There are several backends system that can be connected to istio’s mixer component through what is known as adapters. when we installed istio, we’ve also installed a few backends with their relevant adapters : prometheus and grafana for monitoring, and kiali and jaeger for getting service observability.

Adapters are the mixer components that knows how to translate the telemetry data that envoy proxy send into a readable form that the backend system (prometheus for example) can read.

The installation already pre configured several metrics that will be sent to backends system to process.

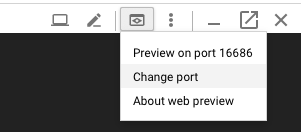
The Envoy sidecar logically calls Mixer before each request to perform precondition checks, and after each request to report telemetry.

Istio Provides a handful of default metrics out of the box that can be used immediately

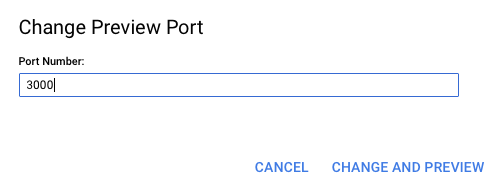
1. Let’s start by connecting to grafana using port forwarding

|  |
| --- |
| $ **kubectl -n istio-system port-forward $(kubectl -n istio-system get pod \**  **-l app=grafana -o jsonpath='{.items[0].metadata.name}') 3000:3000** |

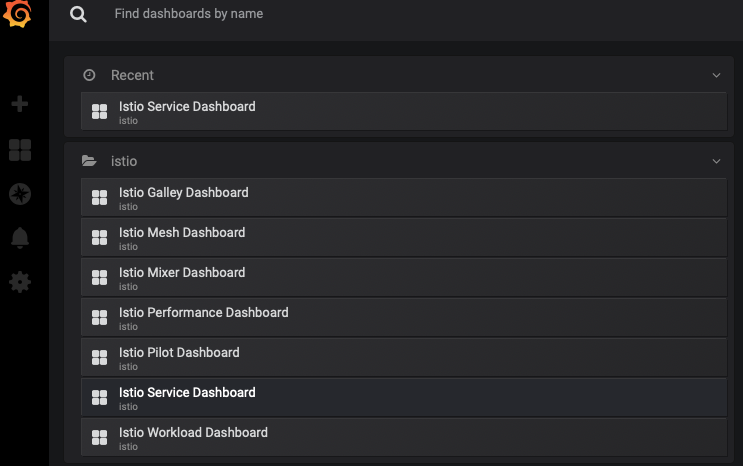
2. Using cloud shell ,click web preview:



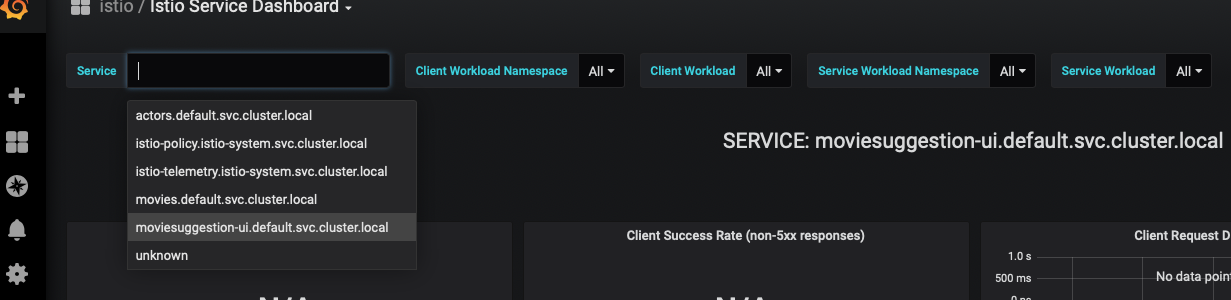
3. Then Change port to 3000, and click “**CHANGE AND PREVIEW**”

****

4. In the Grafana Dashboard, click **Home → Istio Service Dashboard**

****

5. In Service, find and click **moviesuggestion-ui.default.svc.cluster.local**

****

**This will pull up the metrics specific to the movie suggestion UI service.**

**Make a few requests to the moviesuggestion-ui from your browser, and you should also see the Istio Dashboard update with the metrics.**

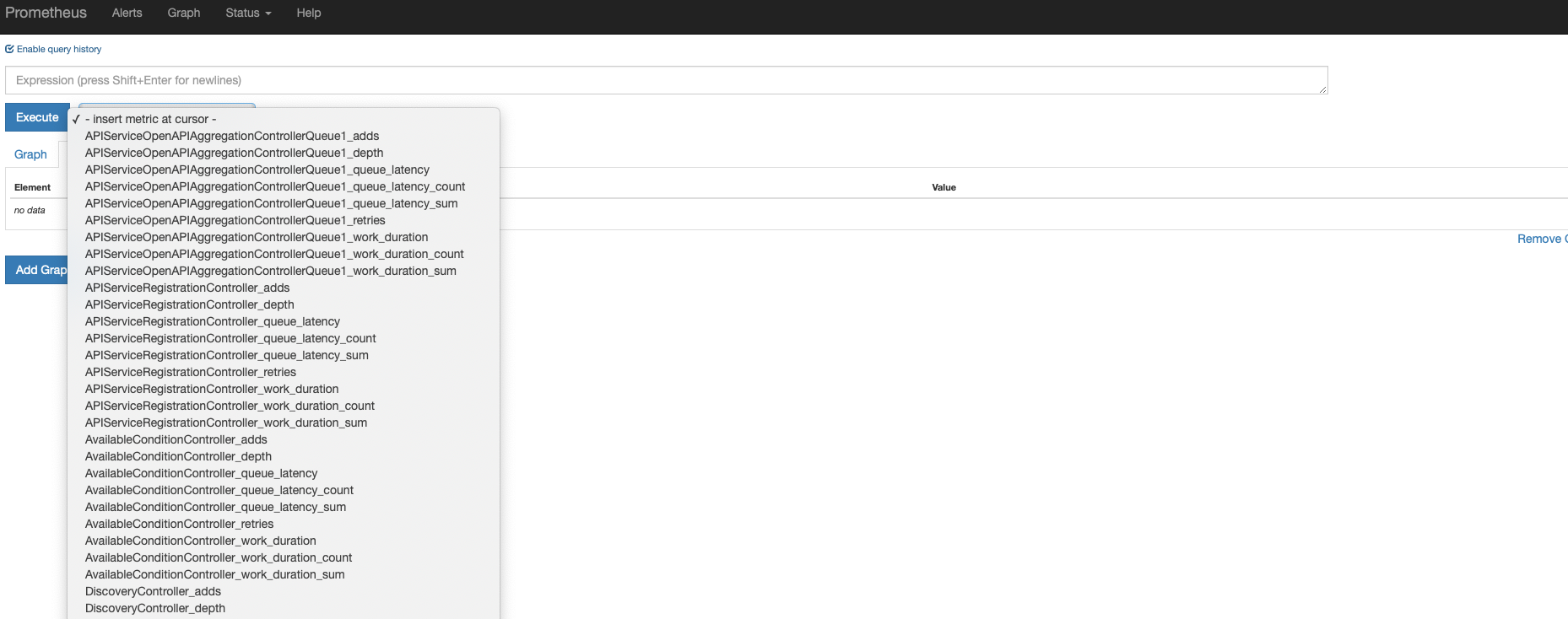
**What actually happens is that Prometheus which also exists in this installation is scraping metrics from Envoy proxy, which has all the monitoring metrics such as latency, response status distributions, etc, and send them to grafana.**

|  |
| --- |
| Please note that you can open a new cloud shell tab and leave grafana running using port forwarding, or you can just Control+c to continue using your cloud shell |

6. Let's Establish a tunnel to Prometheus.

|  |
| --- |
| **$ kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l \**  **app=prometheus -o jsonpath='{.items[0].metadata.name}') 9090:9090** |

7. Use Cloud Shell web preview to preview port 9090. This will take you to the Prometheus console.

****

## 8. We can add our own configuration for a new metric that prometheus wil scrap for us

|  |
| --- |
| **$** kubectl apply -f istio-rules/movie-v1-request-count.yaml |

9. Open your browser and generate a few requests to the moveisuggestion-ui

10. open prometheus and look for the metric you just created (please note that sometimes it can take a few minutes for the metric to show on prometheus).

## 

## Task 2. Observing our microservices

*Kiali project provides answers to the questions:*

*What microservices are part of my Istio service mesh and how are they connected?*

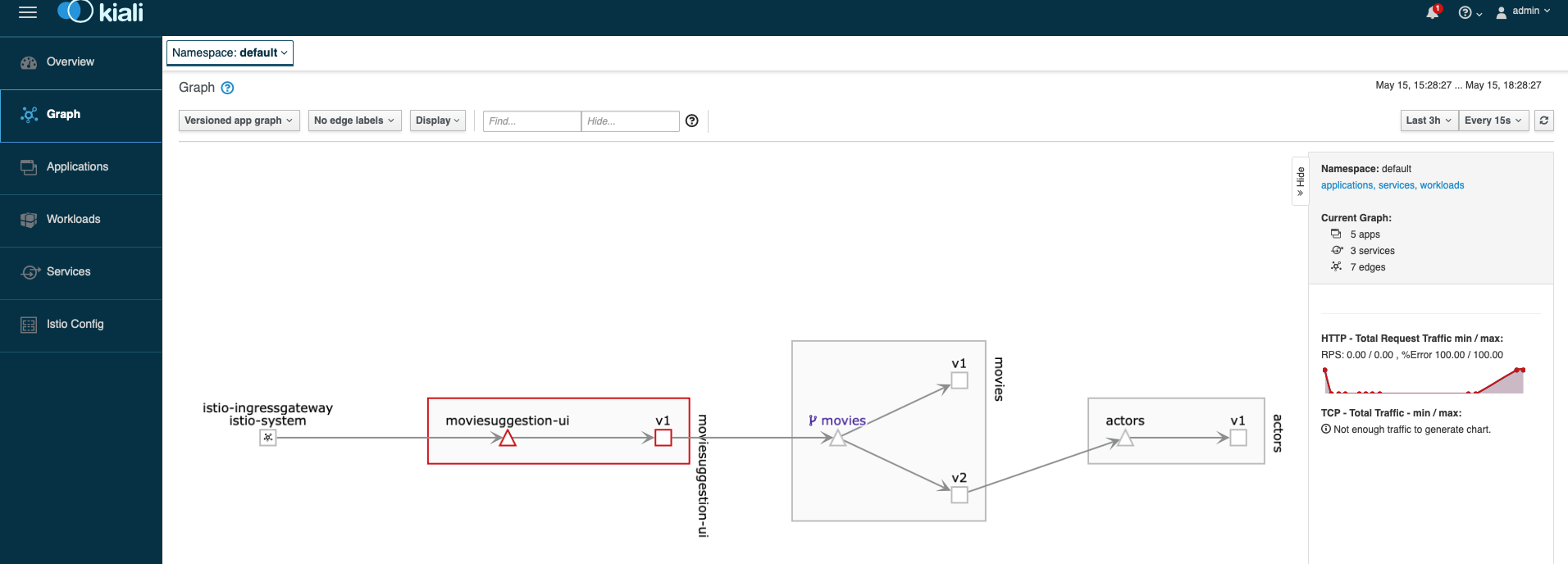
1. Open a connection to kiali using port forwarding as before change the port to 20001

Once you click the “change and preview” button another tab window open but gives you 404 not found error, make sure you add to the opened url“https://20001-dot-7302734-dot-devshell.appspot.com/kiali”

|  |
| --- |
| **$ kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l \**  **app=kiali -o jsonpath='{.items[0].metadata.name}') 20001:20001** |

2. For the username and password type admin admin

Now you should see something similar to that:



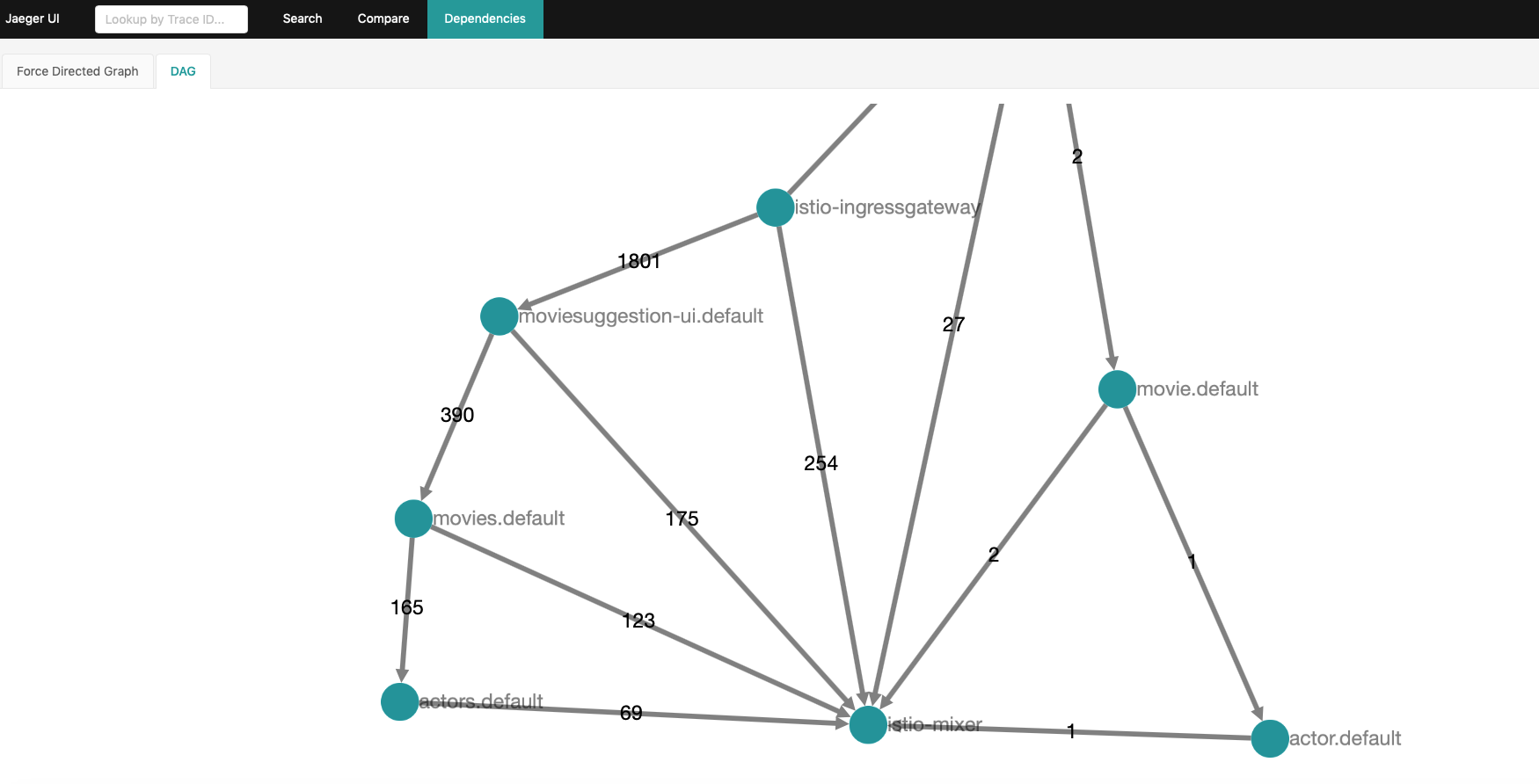
4. Open a connection to jaeger using port forwarding as before change the port to 16686

|  |
| --- |
| **$ kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l \**  **app=jaeger -o jsonpath='{.items[0].metadata.name}') 16686:16686** |

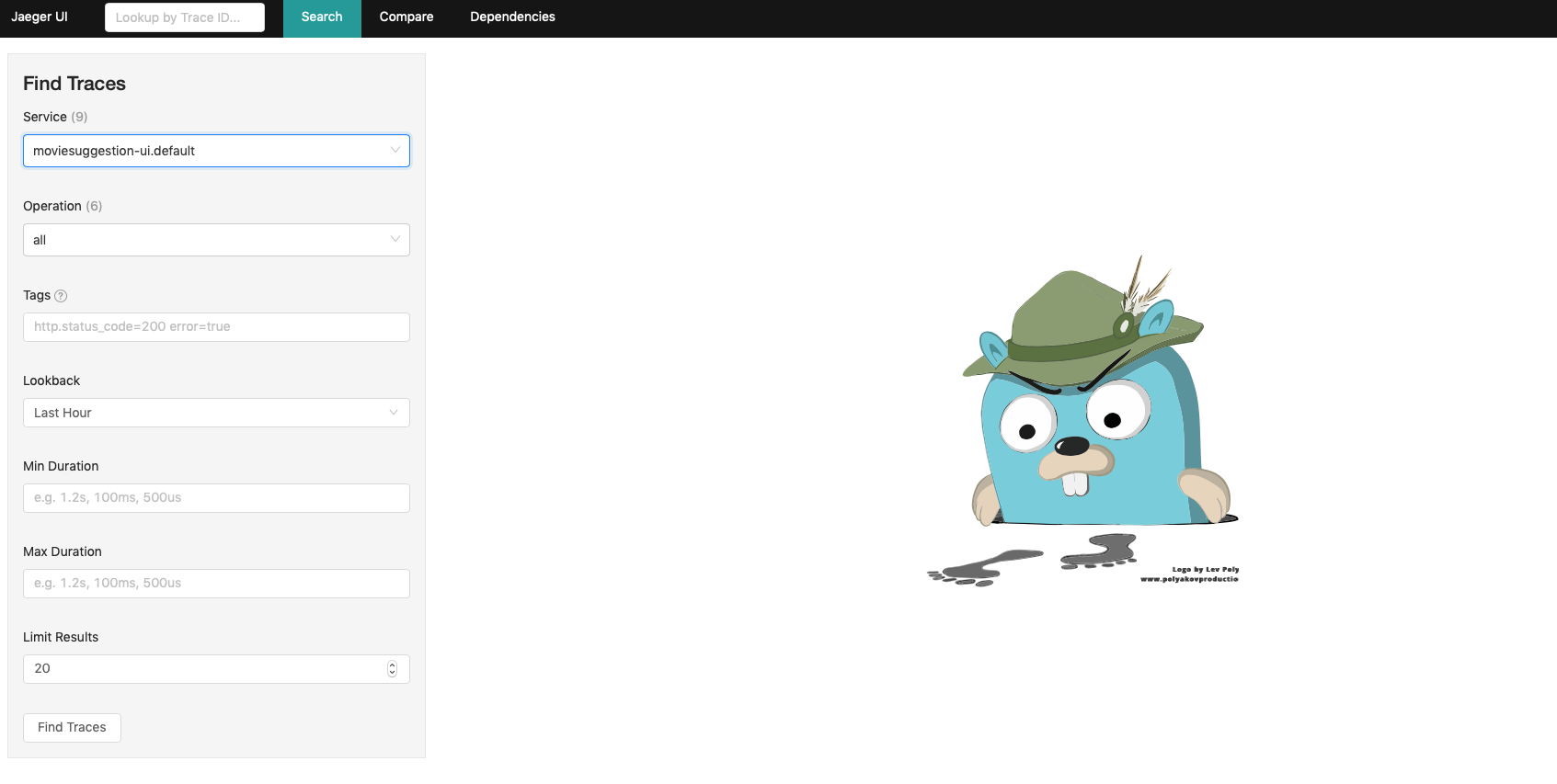
in short jaeger is used to help us with the following:

1. Spans of service
2. Time taken by each service
3. Latency between the services
4. Hierarchy of services
5. Errors or exceptions during execution of each service.

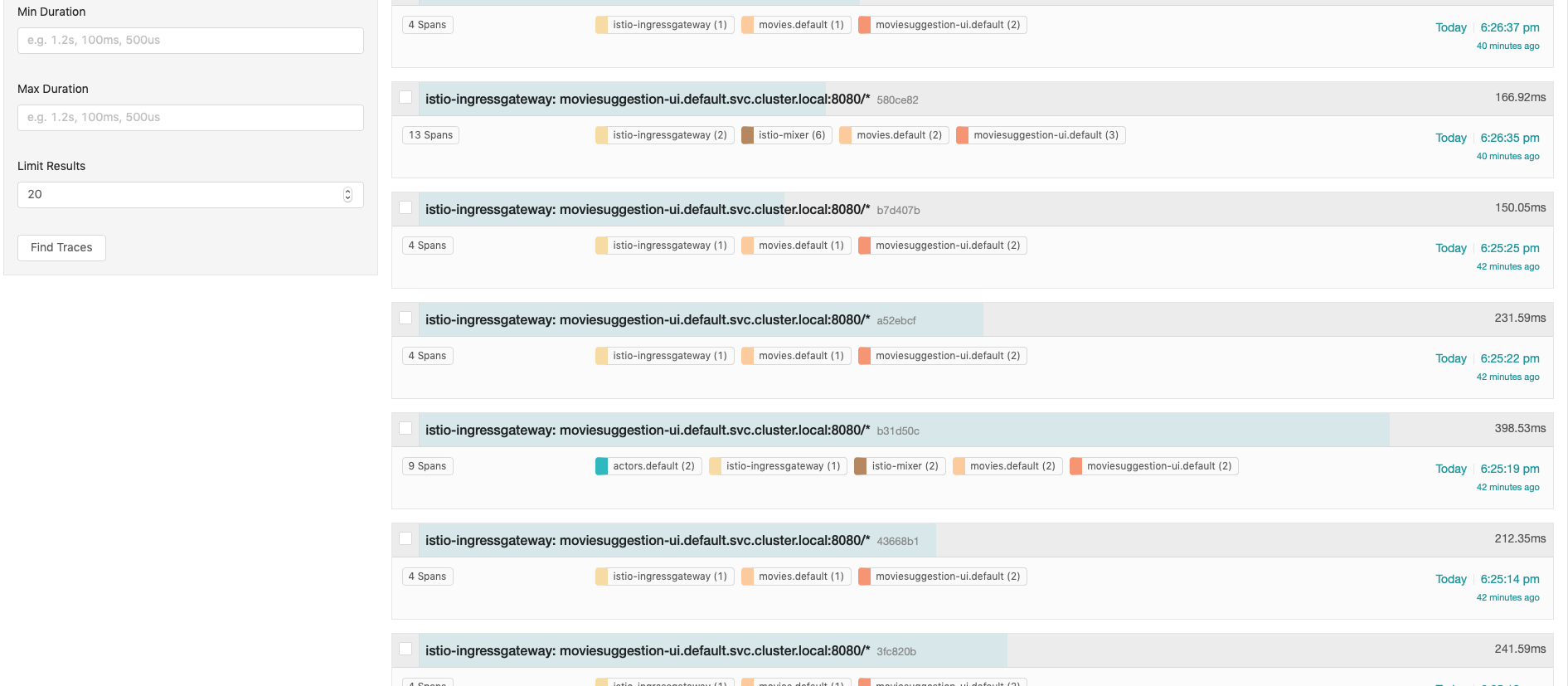
5. you should see the following when clicking the dependencies:



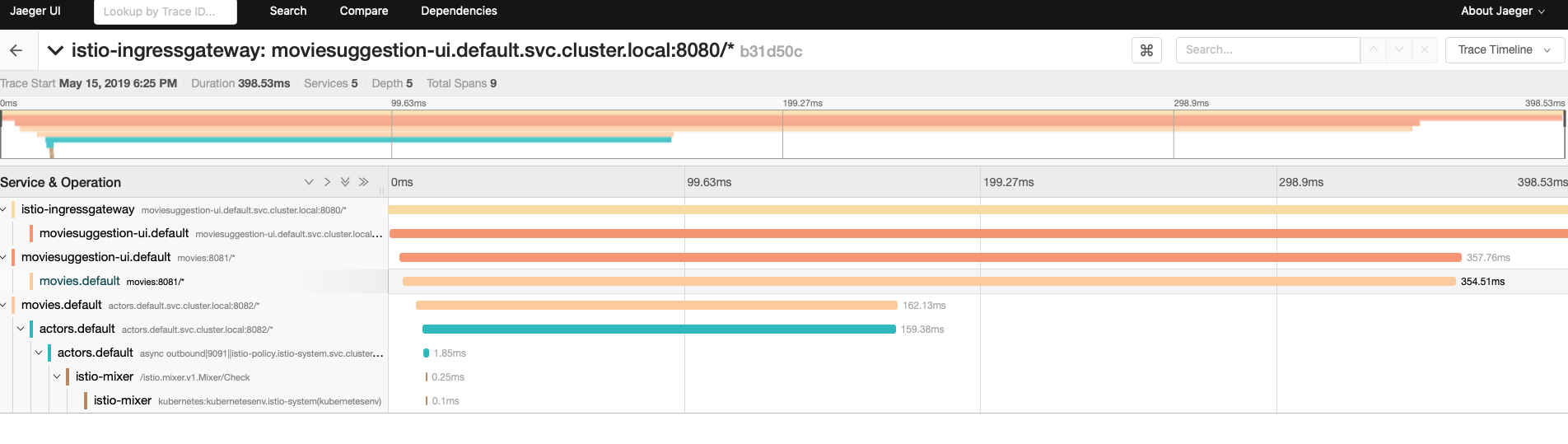
6. Now click the jaeger ui on the top left of the screen and look for traces for the movisuggestion-ui service.



7. Click on one of the traces results on the right:

****

8. And you should see a more detailed screen

****