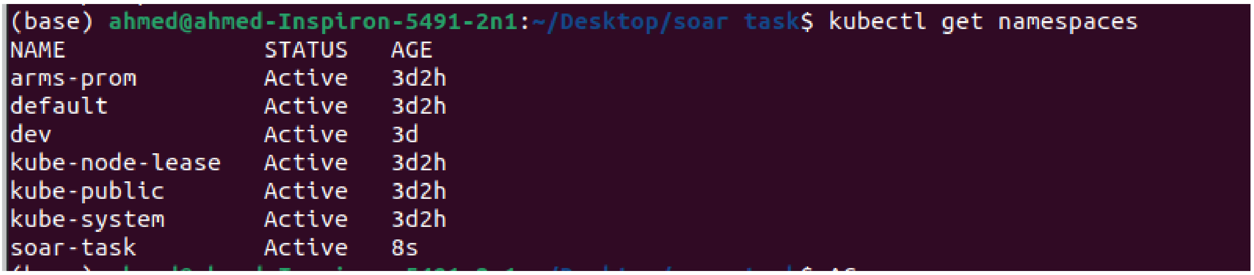
SOAR Service Mesh

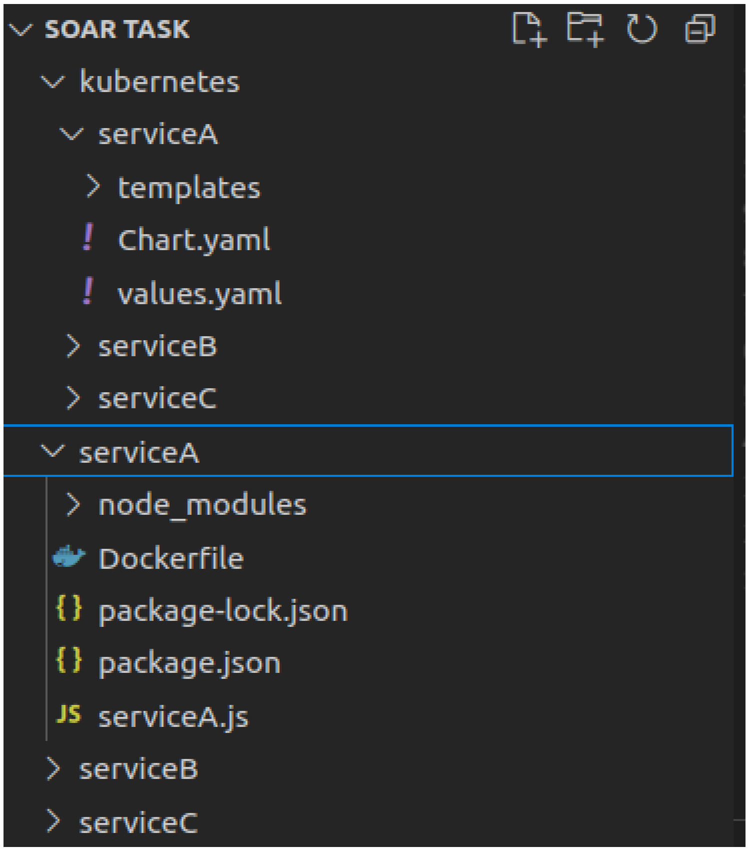
Deploying Micorservices

Creating a namespace soar-task



```
(base) ahmed@ahmed-Inspiron-5491-2n1:~/Desktop/soar task$ kubectl get namespaces
NAME              STATUS   AGE
arms-prom         Active   3d2h
default           Active   3d2h
dev               Active   3d
kube-node-lease   Active   3d2h
kube-public       Active   3d2h
kube-system       Active   3d2h
soar-task         Active   8s
```

Setting the folder structure we will assume a monorepo for simplicity's sake



```
SOAR TASK
  kubernetes
    serviceA
      templates
      Chart.yaml
      values.yaml
    serviceB
    serviceC
  serviceA
    node_modules
    Dockerfile
    package-lock.json
    package.json
    serviceA.js
  serviceB
  serviceC
```

Kubernetes : helm packages for the microserivces

Adding express docker files

```
FROM node:18-alpine


WORKDIR /usr/src/app

COPY package*.json ./

RUN npm ci --verbose


COPY . .

CMD ["node", "serviceA.js"]
```

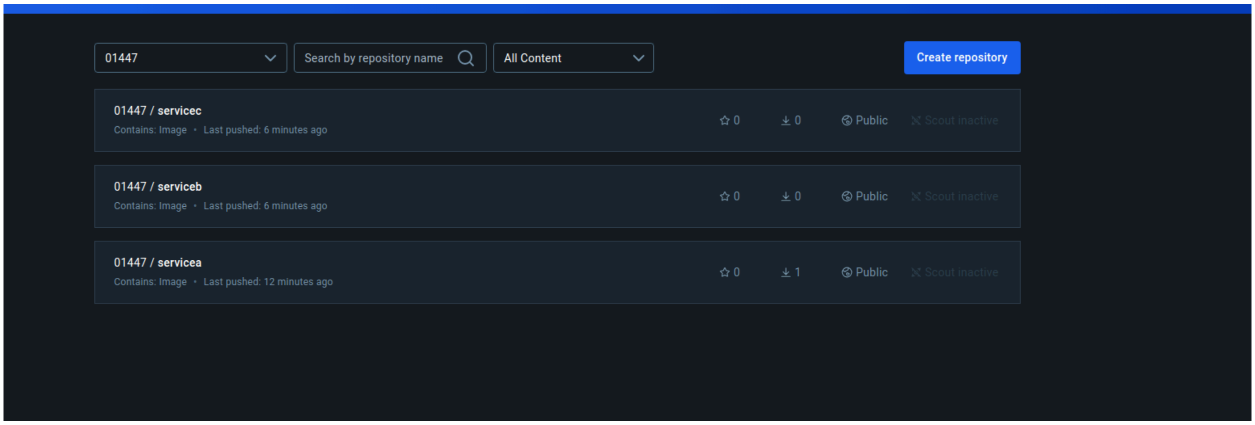
Node18-alpine light weight version of node

Setting the working directory to /usr/src/app

Copying packages first then copying the whole dir as the source code is latest thing that would change

Npm ci is clean installation making sure package.json is maintainable in prod

Build all images and push them to dockerhub



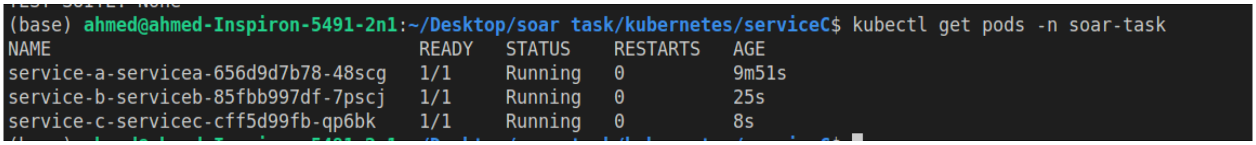
01447 / servicec
Contains: Image • Last pushed: 6 minutes ago
☆ 0 ⤓ 0 Public Scout inactive

01447 / serviceb
Contains: Image • Last pushed: 6 minutes ago
☆ 0 ⤓ 0 Public Scout inactive

01447 / servicea
Contains: Image • Last pushed: 12 minutes ago
☆ 0 ⤓ 1 Public Scout inactive

After Adding helm packages

Each service has been added to its own helm package for modularity.
Ex:
helm upgrade servicec ./ -n soar-task --install



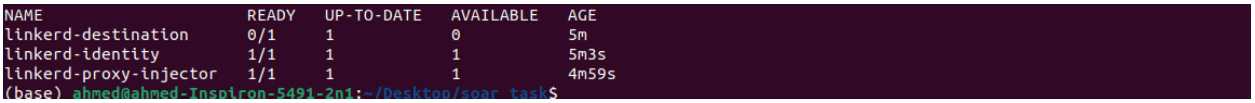
```
TEST SUITE: None
(base) ahmed@ahmed-Inspiron-5491-2n1:~/Desktop/soar task/kubernetes/serviceC$ kubectl get pods -n soar-task
NAME                                READY   STATUS    RESTARTS   AGE
service-a-servicea-656d9d7b78-48scg   1/1     Running   0          9m51s
service-b-serviceb-85fbb997df-7pscj   1/1     Running   0          25s
service-c-servicec-cff5d99fb-qp6bk    1/1     Running   0          8s
```

Since we still did not install linkerd it will only be 1/1 replica pods

Installing linkerd

Following the docs there are multiple sources to install from their opensource docs is the most informative :
Linkerd deployment models:

Host based and sidecar proxy based since wer deploying dockerized microservices on kubernetes we will be using a sidecar proxy

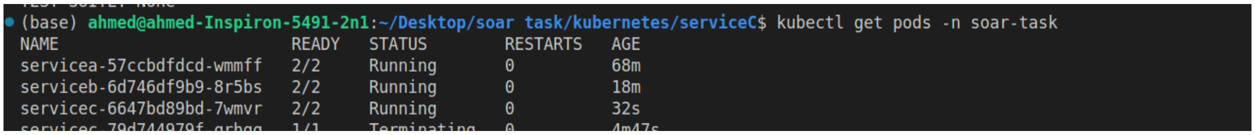


```
NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
linkerd-destination       0/1     1            0           5m
linkerd-identity          1/1     1            1           5m3s
linkerd-proxy-injector    1/1     1            1           4m59s
(base) ahmed@ahmed-Inspiron-5491-2n1:~/Desktop/soar task$
```

Injecting the sidecar:
Either imperative or declarative inside the kuebrentes manifest we went with the adding it inside the kubernetes manifest in the following snippet

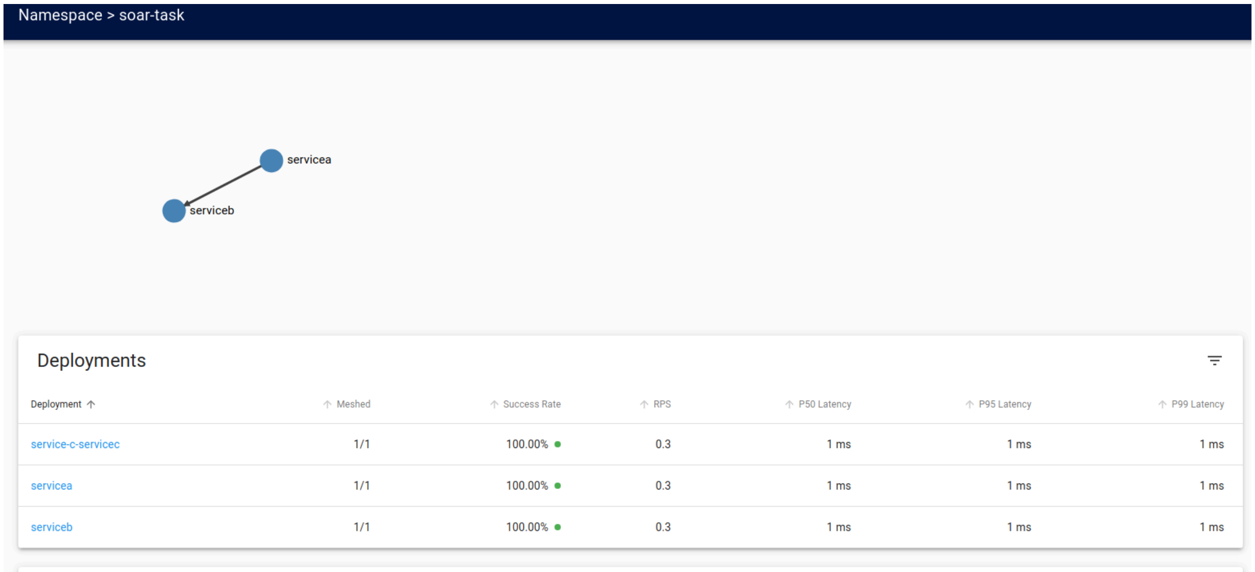
```
template:
  metadata:
    labels:
      app: {{ include "servicec.name" . }}
      release: {{ .Release.Name }}
    annotations:
      linkerd.io/inject: enabled
```

After injecting the sidecar:



```
TEST SUITE: None
(base) ahmed@ahmed-Inspiron-5491-2n1:~/Desktop/soar task/kubernetes/serviceC$ kubectl get pods -n soar-task
NAME                         READY   STATUS        RESTARTS   AGE
servicea-57ccbdfdcd-wmmff    2/2     Running       0          68m
serviceb-6d746df9b9-8r5bs    2/2     Running       0          18m
servicec-6647bd89bd-7wmvr    2/2     Running       0          32s
servicec-79d744979f-grbqg    1/1     Terminating   0          4m47s
```

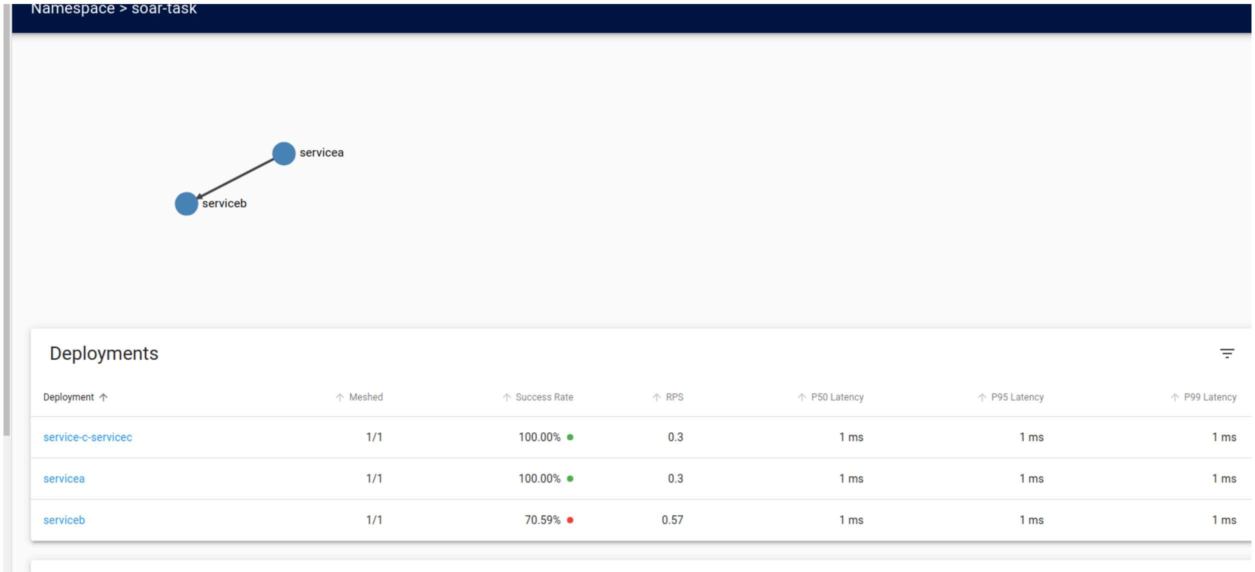
At a first glance all services seem to be working which doesnt seem to be making sense the reason is serviceB’s error is handled in a try catch statement



Deployments

Deployment ↑	↑ Meshed	↑ Success Rate	↑ RPS	↑ P50 Latency	↑ P95 Latency	↑ P99 Latency
service-c-servicec	1/1	100.00% 🟢	0.3	1 ms	1 ms	1 ms
servicea	1/1	100.00% 🟢	0.3	1 ms	1 ms	1 ms
serviceb	1/1	100.00% 🟢	0.3	1 ms	1 ms	1 ms

We’ll change microseviceB so instead of handling the error it can sometimes throw a 500 error and we will send a curl request from servicea to serviceb:

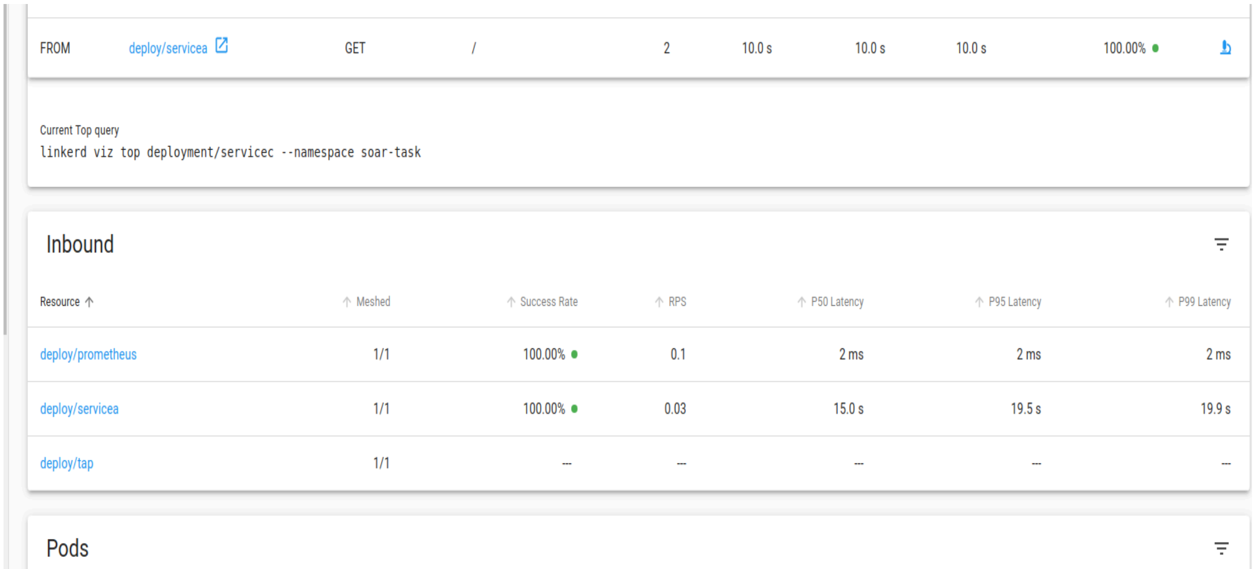


Deployments

Deployment ↑	↑ Meshed	↑ Success Rate	↑ RPS	↑ P50 Latency	↑ P95 Latency	↑ P99 Latency
service-c-servicec	1/1	100.00% 🟢	0.3	1 ms	1 ms	1 ms
servicea	1/1	100.00% 🟢	0.3	1 ms	1 ms	1 ms
serviceb	1/1	70.59% 🔴	0.57	1 ms	1 ms	1 ms

It reflects right away !
So for serviceB using error handling would do the trick

Now that we finally got through with service B lets check serviceC the high latency one when we start hitting its endpoint we can see it reflects in the latency p50, p95 and p99 latencies all represent the percentiles of different latencies so under normal circumstances if the total number of request latencies is normally distributed our p99 latency would mean that around 99 percent of the requests hitting our microservice would be 99 percent and same applies to p95 and p50.



FROM	deploy/servicea	GET	/	2	10.0 s	10.0 s	10.0 s	100.00% ●

Current Top query

```
linkerd viz top deployment/servicec --namespace soar-task
```

Inbound

Resource ↑	↑ Meshed	↑ Success Rate	↑ RPS	↑ P50 Latency	↑ P95 Latency	↑ P99 Latency
deploy/prometheus	1/1	100.00% ●	0.1	2 ms	2 ms	2 ms
deploy/servicea	1/1	100.00% ●	0.03	15.0 s	19.5 s	19.9 s
deploy/tap	1/1	---	---	---	---	---

Pods

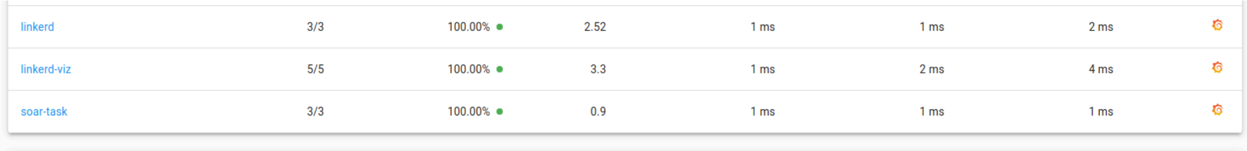
Reflection: since its a generic timeout function it is apparent that latency in response times would drop once we reduce the time in the function however in production environments it could be multiple factors such as the cpu resources of our kubernetes nodes, the allowed cpu the microservice can take up from the node and finally the source code itself where it might be slow from the data layer maybe slow query performance or the endpoint calling multiple functions that could be better broken down.

Prometheus and Grafanna

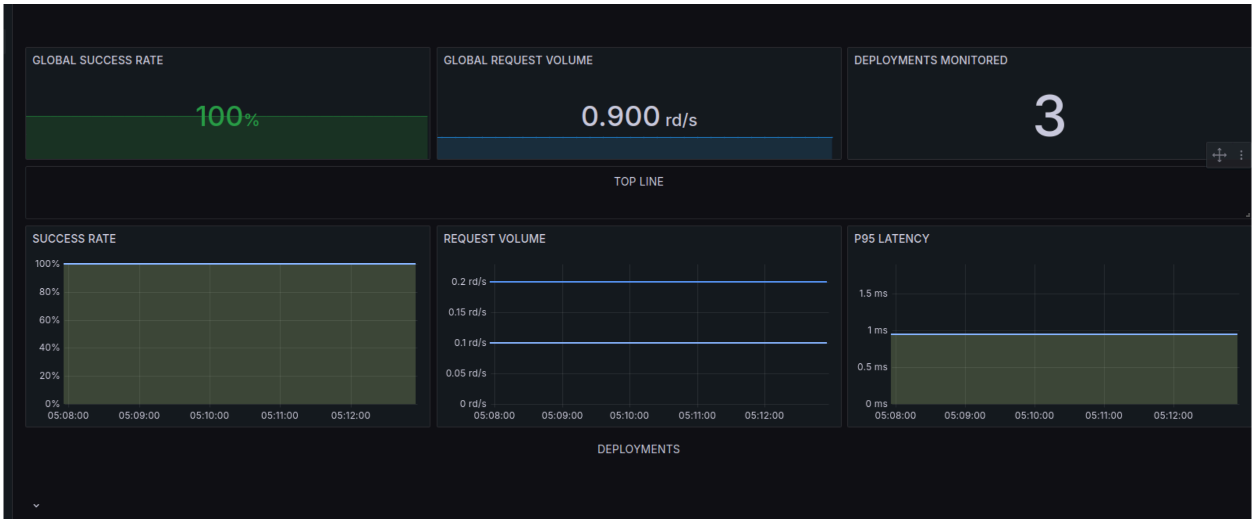
Now that we’ve finished all of our service mesh setup and put down our reflection we need one final integration along with linkerd inorder to persist the metrics getting stored.

Linkerd viz extension that we’ve installed installs prometheus so our final integration is to install grafana.

After the follow along get a grafana icon next to linkerd:



linkerd	3/3	100.00% ●	2.52	1 ms	1 ms	2 ms
linkerd-viz	5/5	100.00% ●	3.3	1 ms	2 ms	4 ms
soar-task	3/3	100.00% ●	0.9	1 ms	1 ms	1 ms



Once we click on a deployment we get the full grafana dashboard with the linkerd metrics already present.