

Abstract

Expanse and movement to clouds brings new challenges for developers. To utilize the most of cloud features new applications should be scalable, resilient and fast as while developing them, testing or pushing to production. One of the solutions is rethinking of old monolith architectures and refactor them to microservices or start to use cloud native development patterns for completely new projects. This transition to microservices scales very well with newly adopted container technologies.

Splitting one monolith application in number of microservices (often huge number) brings new challenges in software engineering processes. Especially completely new methods need to be used in operations departments to monitor, scale and deliver resilient workflow in software life cycle. One of the most important things to consider when running a complex distributed application is resiliency.

In this thesis service mesh Istio running on top of kubernetes cluster will be introduced as a solution to provide visibility, control, security and fault tolerance to your deployments. A working demo with the possibility to try out resiliency features of Istio is the final goal of the thesis.

Keywords

Microservices, Kubernetes, Service Mesh, resiliency, Istio

Table of Contents

Abstract.....	1
Keywords.....	1
Motivation.....	3
Related work.....	3
Major idea.....	3
Microservices.....	3
Service mesh.....	3
Istio.....	3
Resiliency.....	4
Demo.....	5
Implementation.....	5
The Twelve Factors App.....	5
Deploy with Kubernetes.....	7
Deploy with Istio.....	7
How to run.....	7
Evaluation.....	7
Routing.....	8
Load balancing.....	13
Fault injection.....	14
Timeout.....	17
Retries.....	19
Circuit breaker.....	21
Discussion.....	21
Conclusion.....	21
Future Work.....	22
References.....	22
Supplemental Material.....	22

Motivation

adoption of containers and docker changed everything, applications are packaged in images that run the same way by developer as in production environment. Containers are more lightweight and blazing fast in startup in compare with virtual machines.

migration to clouds → microservices, devops, fast code-to-market, leave only business logic for developers

The problem of delivering code from developers to productions is solved with packaging application and dependencies in images.

number of microservices grows, lack of visibility and control,

kubernetes, no possibility to deal with network errors – focus on pods

goals, metrics: deploy microservices app, compare resiliency with and without istio

cc project as template (refactored, adopted), deploy istio, demo in minikube, test resiliency

Related work

Need for service meshes

compare API gateway and service mesh

other service meshes/libraries, pros/cons, trend

- libs: cons - code change (hystrix, ribbon)
- node agent (linkerd)
- sidecar (istio, linkerd2, consul)

Major idea

Microservices

Motivation

pros/cons

Service mesh

Motivation

types

sidecar proxies

pros/cons

Istio

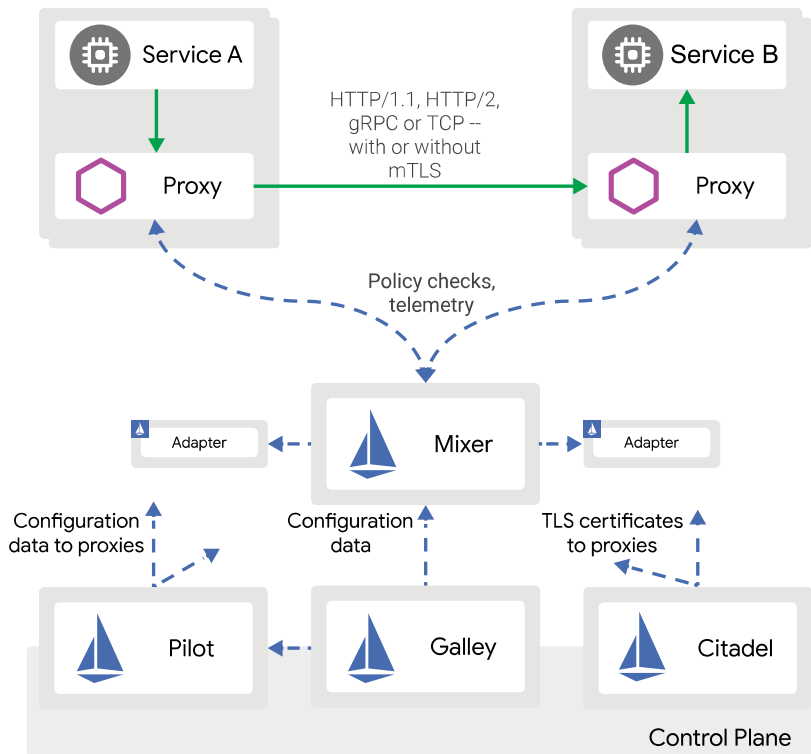
features, API installed as kubernetes CRD

- service mesh
 - security – who talks whom, trusted communication
 - observability – tracing, metrics, alerting
 - routing control
 - load balancing
 - communication resiliency
 - API (kubernetes CRD)
- architecture

- data plane – traffic routing
- control plane – tls, policies

puts resilience into the infrastructure

pilot – get rules and send them to proxies, works dynamically on the fly, without restart needed, looks into all registries in system and understands topology of deployment, uses service discovery adapter (k8s, consul)



mixer – take telemetry to analyze, has policies, all side cars calls mixer, if request is allowed, quotas, authZ backends, turns data into info → high cpu load, has caching → not single point of failure

citadel – certificates mTLS

galley – holds configs

Manifests:

Virtual services – route traffic (headers, weight, URL), retries, timeouts, fault injection

destination rules – named subsets, circuit breaker, load balancing

gateways – Virtual Service to allow L7 routing, use default or deploy own

- ingress – to expose service with kubernetes
- egress – by default all external traffic is blocked, enabled in Service entry

Service entry – automatic from pilot, from k8s - service names and ports,

Kiali – visualize services that are deployed

Grafana with prometheus as backend

- runs in its own namespace – isolated from other procs
- fault injection:
 - http error codes, eg 400
 - delays

Resiliency

definition

+ Health checks – uses native k8s mechanisms. Http health checks work only with mTLS enabled.

+ Client-side load balancing

+ Timeout – virt svc, default = 15 sec
+ Retry – virt svc, default = NO
circuit breaker – dest rule – LB least request
Pool ejection = Outlier Detection

Demo

Git, Minikube, docker images, kubernetes, istio, resiliency try out with Makefile

Implementation

The Twelve Factors App

There are plenty of tutorials online that utilize a sample application from istio web site (“Bookinfo” application) to show typical service mesh and specific istio features. The idea of this thesis is to take the already implemented project, adopt it a little bit and provide a working demo of istio resiliency features. The project itself is a part of cloud computing course. The text of the original assignment can be found in appendix.

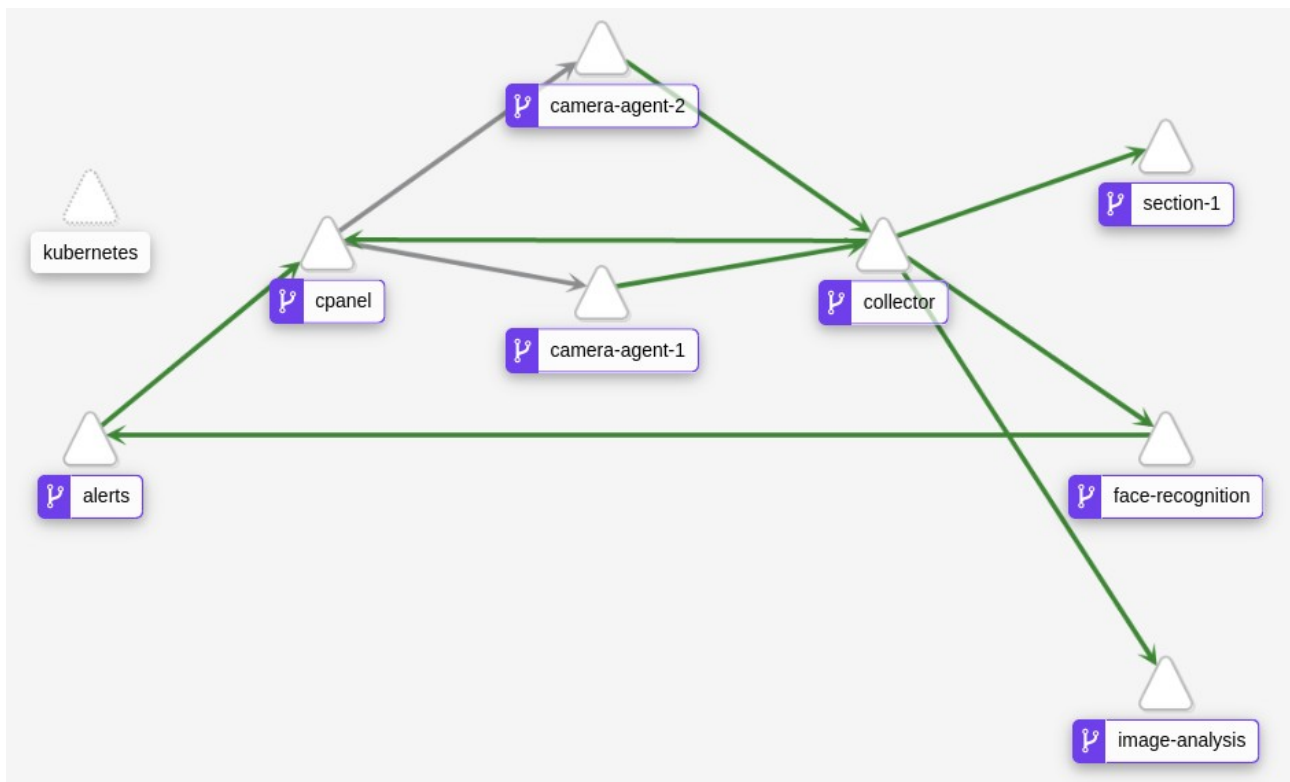
Other possibility was to take a ready open sourced project from github and deploy it with istio. After researching and looking into some of such projects the decision to take the application was developed by myself was made.

Application itself is a simulation of airport security system.

There are camera agents to stream image frames from dedicated airport sections. Cameras can be placed on entry or exit from the section. There is a configuration file for control panel that provides this information to system.

```
"cameras": [{  
    "id": 1,  
    "description": "exit camera section 1",  
    "url": "http://camera-agent-1:8080",  
    "section": 1,  
    "type": "exit"  
},
```

For simplicity of simulation “config.json” is packaged with docker image. So to update it you need to rebuild image or change it manually inside of running container and then update via special control panel endpoint.



Collector receives frames from camera agents in json format and forward them to other microservices for analysis.

Image analysis takes frame and responses back with statistics about how many people are there, their gender and age. After that collector forwards statistics information about current image to section microservice.

Section stores the statistical information from current frame in json file.

Face recognition forwards response if there are any persons of interest on the image to alert microservice.

Replication of pods is configured for collector, image analysis and face recognition. Camera agents, face recognition and image analysis microservices were already implemented and provided as docker images. The rest of microservices (collector, section, alerts and cpanel) were developed during the cloud computing course.

More detailed description of the initial API and the hole system itself can be found in cloud computing assignment [cc].

Additional endpoints were implemented in each microservice:

alerts: /status

collector: GET /status

section: GET /status

cpanel: GET /status

GET /, /index

GET/POST /analysis

GET/POST /alert

The Twelve Factors App, build working demo to play around, changes made, resiliency in project, no down time, user satisfaction, easy to monitor, screenshots, cpanel v1/v2

- architecture, API, REST, diagrams
- k8s to deploy
- docker – runtime / packaging
- istio as service mesh

The Twelve Factors App

1. **Codebase**

One codebase tracked in revision control, many deploys - **GitHub**

2. **Dependencies**

Explicitly declare and isolate dependencies - **requirements.txt**

3. **Config**

Store config in the environment - **env variables**

4. **Backing Services**

Treat backing services as attached resources – **NO (json) or mount volume. It is recommended to use databases.**

5. **Build, release, run**

Strictly separate build and run stages – **docker images with env vars and versions**

6. **Processes**

Execute the app as one or more stateless processes – **Docker**

7. **Port binding**

Export services via port binding - **completely self-contained, exports HTTP as a service by binding to a port, gunicorn**

8. **Concurrency**

Scale out via the process model – **LB with docker containers**

9. **Disposability**

Maximize robustness with fast startup and graceful shutdown - **Docker**

10. **Dev/Prod parity**

Keep development, staging, and production as similar as possible - **Docker**

11. **Logs**

Treat logs as event streams – **logs to stdout**

12. **Admin Processes**

Run admin/management tasks as one-off processes - ???

- **refactor** and **expanse** of cc project
 - 12 factor
 - unit tests
 - frontend v1/v2
 - canary, blue/green deployment, user resiliency
 - python + docker best practices:
 - gunicorn, root, alpine, no cache
 - scaling deployment:
 - collector, image-analysis, face-recognition
 - docker compose for local development, but telepresence is better

Deploy with Kubernetes

- services – fqdn, service discovery
- deployments with pods
- readiness/liveness - resiliency
- resources limits – to protect pods from starvation

Deploy with Istio

istio verify install done in script

single cluster deployment

virtual services , destination rules, ingress

fault injection: delays and aborts, retries, timeouts, circuit braking
best practices: add dest rules and virt svc for all microservices

How to run

github, virtualbox, curl, docker, shell scripts, yaml, minikube with kubectl, istio, install requirements (ram, cpu),

- dirty tricks:
 - sharing containers host/guest minikube

Evaluation

- demo with Makefile, tests - screenshots/results
- comparison to k8s only

Kubernetes has only round robin load balancing. Istio with the help of destinations rules extends native kubernetes load balancing and presents the following types: random, round robin, weighted least request, ring hash (#istio). In such a case istio can give any microservice replica set it's own load balancer. To show how istio load balancing can be configured, we need first to learn about routing mechanism provided by istio.

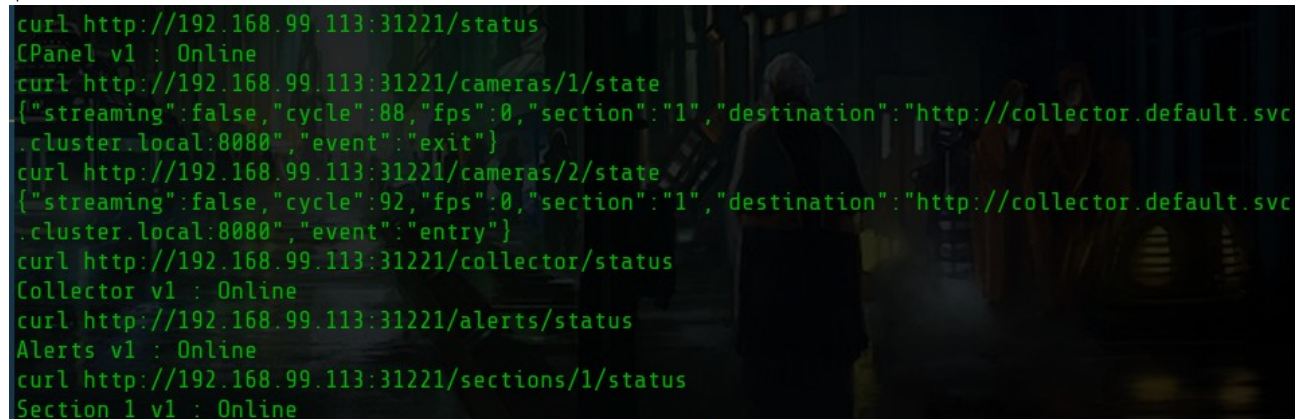
Routing

This solution can be used to make canary deployments and also make user experience more resilient - "user resilience". For example, new version of service can be made available only to one group of users (test group). It can be as much as only 1% of of the hole traffic. Users can be filtered by headers in http request. If something goes wrong with new version of service it is very easy to rollback and switch all the traffic back to production version. This mechanism allows also to do blue/green deployments.

```
$ make deploy-app-default
```

```
$ make deploy-istio-default
```

```
$ make health
```

A terminal window with a dark background and green text. It shows a series of curl commands being executed against a service running on 192.168.99.113:31221. The responses indicate that various components like CPANEL v1, Collector v1, Alerts v1, and Section 1 v1 are all 'Online'.

```
curl http://192.168.99.113:31221/status
CPANEL v1 : Online
curl http://192.168.99.113:31221/cameras/1/state
{"streaming":false,"cycle":88,"fps":0,"section":"1","destination":"http://collector.default.svc
.cluster.local:8080","event":"exit"}
curl http://192.168.99.113:31221/cameras/2/state
{"streaming":false,"cycle":92,"fps":0,"section":"1","destination":"http://collector.default.svc
.cluster.local:8080","event":"entry"}
curl http://192.168.99.113:31221/collector/status
Collector v1 : Online
curl http://192.168.99.113:31221/alerts/status
Alerts v1 : Online
curl http://192.168.99.113:31221/sections/1/status
Section 1 v1 : Online
```

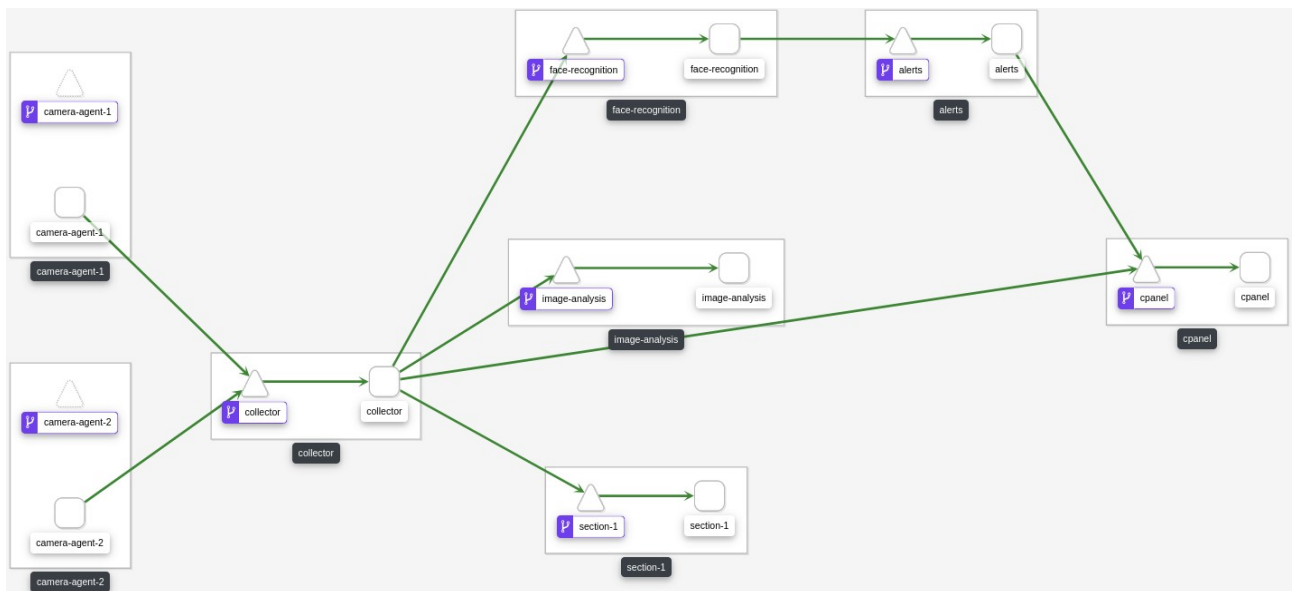
```
$ make start-cameras
```

```
$ make health
```

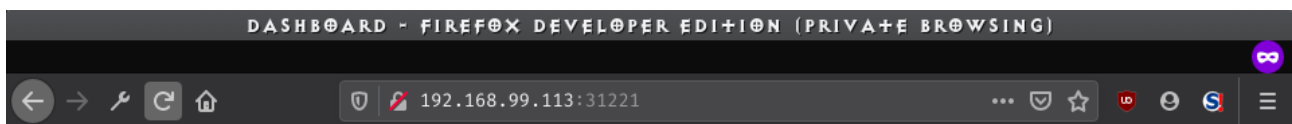


```
curl http://192.168.99.113:31221/status
CPanel v1 : Online
curl http://192.168.99.113:31221/cameras/1/state
{"streaming":true,"cycle":7,"fps":0,"section":"1","destination":"http://collector.default.svc.c
luster.local:8080","event":"exit"}
curl http://192.168.99.113:31221/cameras/2/state
{"streaming":true,"cycle":5,"fps":0,"section":"1","destination":"http://collector.default.svc.c
luster.local:8080","event":"entry"}
curl http://192.168.99.113:31221/collector/status
Collector v1 : Online
curl http://192.168.99.113:31221/alerts/status
Alerts v1 : Online
curl http://192.168.99.113:31221/sections/1/status
Section 1 v1 : Online
```

```
curl http://192.168.99.113:31221/status
CPanel v1 : Online
curl http://192.168.99.113:31221/cameras/1/state
{"streaming":true,"cycle":7,"fps":0,"section":"1","destination":"http://
collector.default.svc.cluster.local:8080","event":"exit"}
curl http://192.168.99.113:31221/cameras/2/state
{"streaming":true,"cycle":5,"fps":0,"section":"1","destination":"http://
collector.default.svc.cluster.local:8080","event":"entry"}
curl http://192.168.99.113:31221/collector/status
Collector v1 : Online
curl http://192.168.99.113:31221/alerts/status
Alerts v1 : Online
curl http://192.168.99.113:31221/sections/1/status
Section 1 v1 : Online
```

Version 1 of Cpanel microservice displays information about latest statistic from image analysis and the most recent alert. Both are displayed without showing the photo from camera agent itself. Displaying the photo is made in Version 2 of Cpanel microservice.



Dashboard V1

Section 1

timestamp: 2020-02-25T14:35:38.204522Z

gender: male | age: 38-43 | event: exit

Alert

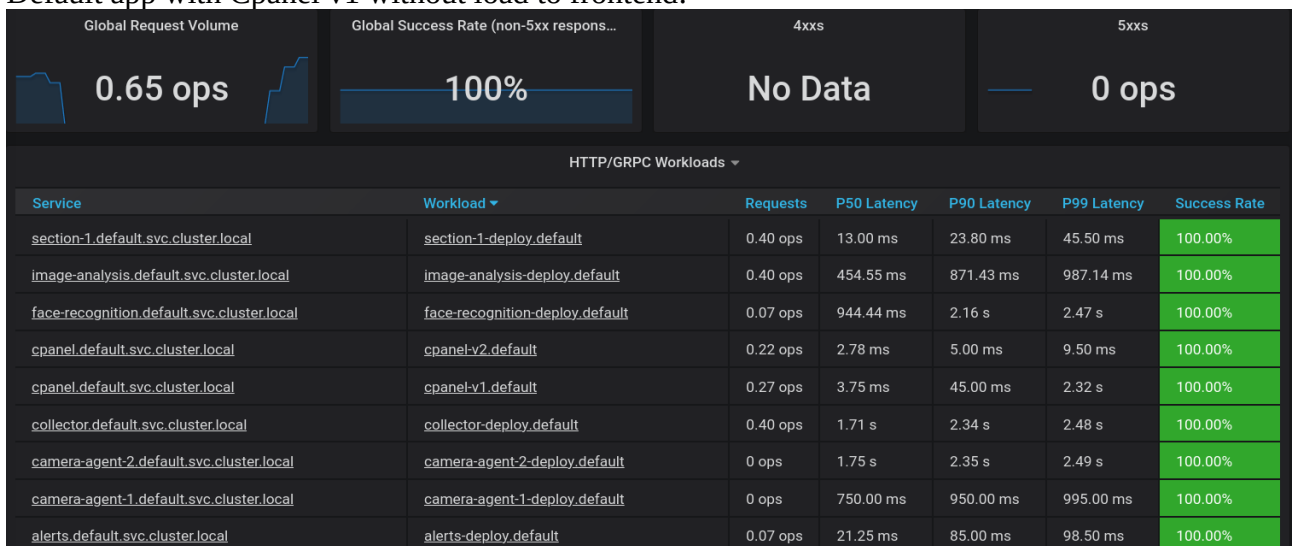
timestamp: 2020-02-25T14:35:27.224857Z

section: 1

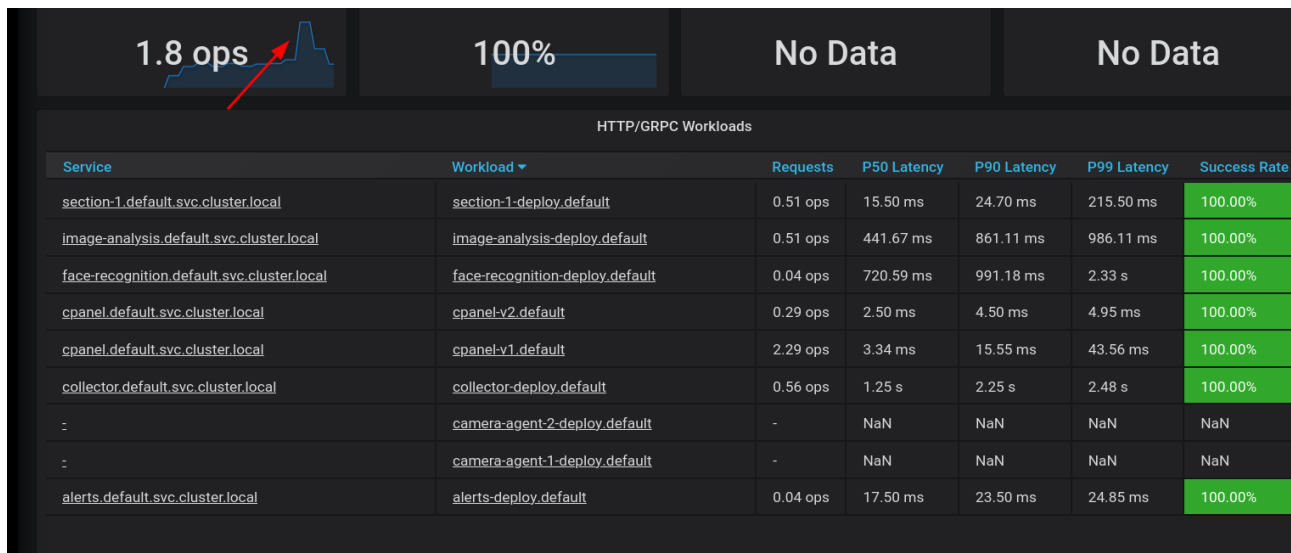
event: entry

name: **PersonX**

Default app with Cpanel v1 without load to frontend:



```
$ make load
for i in {1..100}; do sleep 0.2; curl http://192.168.99.113:31221/status; printf "\n"; done
CPanel v1 : Online
CPanel v1 : Online
CPanel v1 : Online
...
```

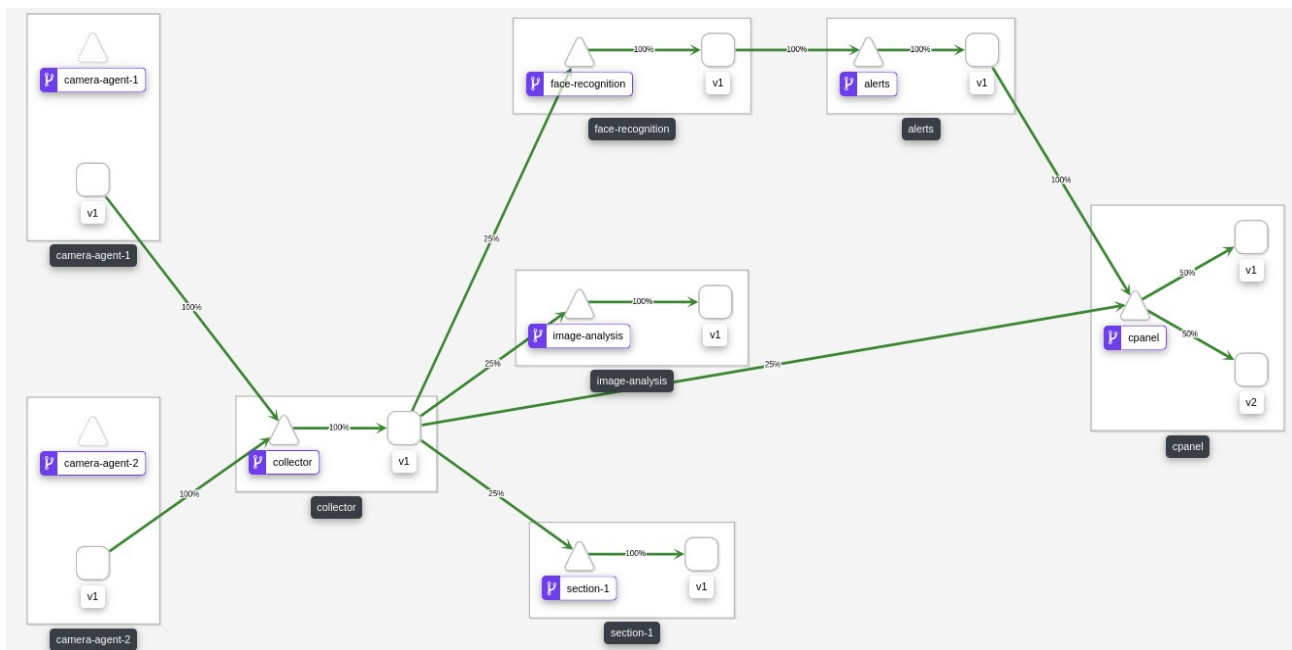


```
$ make cpanel-50-50
./kubectl apply -f istio/virt_svc_50-50.yaml
virtualservice.networking.istio.io/cpanel configured
check configuration
$ k get virtualservices cpanel -o yaml
```

route:

- destination:
 - host: cpanel.default.svc.cluster.local
 - port:
 - number: 8080
 - subset: v1
 - weight: 50
- destination:
 - host: cpanel.default.svc.cluster.local
 - port:
 - number: 8080
 - subset: v2
 - weight: 50

```
$ make start-cameras
```



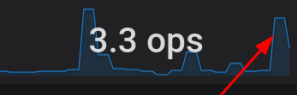
```

$ make load
for i in {1..100}; do sleep 0.2; curl http://192.168.99.113:31221/status; printf "\n"; done
CPanel v1 : Online
CPanel v1 : Online
CPanel v1 : Online
CPanel v2 : Online
CPanel v2 : Online
CPanel v2 : Online

```

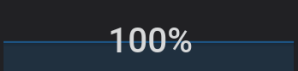
Global Request Volume

3.3 ops



Global Success Rate (non-5xx responses...)

100%



4xxs

No Data

5xxs

No Data

HTTP/GRPC Workloads

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default	0.44 ops	83.33 ms	220.00 ms	450.00 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.47 ops	461.54 ms	1.60 s	4.45 s	100.00%
face-recognition.default.svc.cluster.local	face-recognition-deploy.default	0.04 ops	857.14 ms	2.50 s	9.00 s	100.00%
cpanel.default.svc.cluster.local	cpanel-v2.default	1.40 ops	2.92 ms	6.93 ms	20.28 ms	100.00%
cpanel.default.svc.cluster.local	cpanel-v1.default	1.33 ops	3.19 ms	9.38 ms	42.50 ms	100.00%
collector.default.svc.cluster.local	collector-deploy.default	0.47 ops	1.59 s	2.49 s	8.95 s	100.00%
alerts.default.svc.cluster.local	alerts-deploy.default	0.04 ops	17.50 ms	23.50 ms	24.85 ms	100.00%

```

$ make cpanel-v2
./kubectl apply -f istio/virt_svc_v2.yaml
virtualservice.networking.istio.io/cpanel configured
check configuration
$ k get virtualservices cpanel -o yaml

```

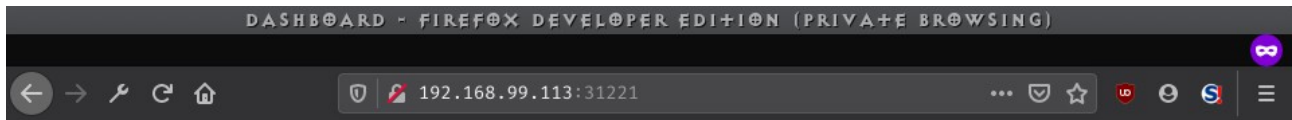
```

route:
- destination:
    host: cpanel.default.svc.cluster.local
    port:

```

```
    number: 8080
    subset: v1
weight: 0
- destination:
    host: cpanel.default.svc.cluster.local
    port:
        number: 8080
        subset: v2
weight: 100
```

```
$ make start-cameras
```

Dashboard V2

Section 1



timestamp: 2020-02-25T15:27:21.900453Z

gender: male | age: 25-32 | event: entry

gender: male | age: 25-32 | event: entry

Alert



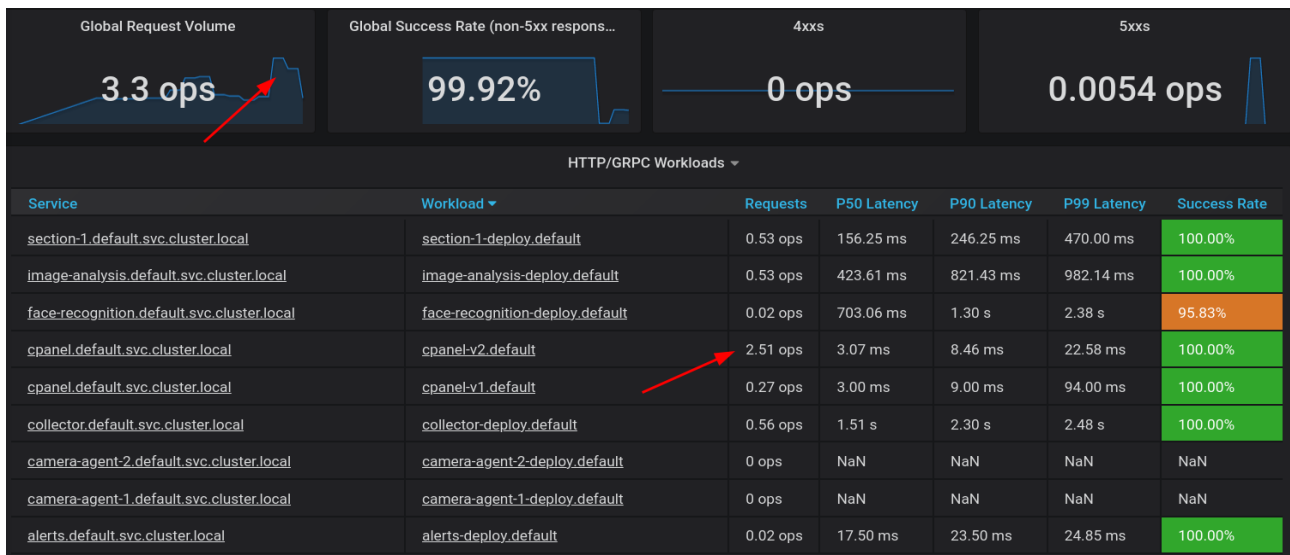
timestamp: 2020-02-25T15:26:55.022111Z

section: 1

event: exit

name: **George W**

```
$ make load
for i in {1..100}; do sleep 0.2; curl http://192.168.99.113:31221/status; printf "\n"; done
CPanel v2 : Online
CPanel v2 : Online
CPanel v2 : Online
CPanel v2 : Online
```



Load balancing

Default round robin between v1 and v2 cpanel (should be 1:3)

```
$ make scale_v2_x3
```

```
kubectl scale deployment cpanel-v2 --replicas=3
```

```
collector-deploy-558dd7dd45-8rlwq      2/2   Running 3    9h
cpanel-v1-8446d9dd45-wx6mz             2/2   Running 2    9h
cpanel-v2-8445ff5964-lgj84             1/2   Running 0    6s
cpanel-v2-8445ff5964-qdhk8             0/2   Running 0    6s
cpanel-v2-8445ff5964-r4r2d             2/2   Running 3    9h
face-recognition-deploy-7b954c454-fdphg 2/2   Running 3    9h
```

Here we can see how kubernetes scales our service.

```
$ make load_balancing
```

```
./kubectl apply -f istio/round_robin.yaml
```

route:

- destination:

host: cpanel.default.svc.cluster.local

port:

number: 8080

```
$ make load
```

```
for i in {1..100}; do sleep 0.2; curl http://192.168.99.113:31221/status; printf "\n"; done
```

CPanel v2 : Online

CPanel v2 : Online

CPanel v2 : Online

CPanel v2 : Online

CPanel v2 : Online

CPanel v1 : Online

CPanel v2 : Online

CPanel v1 : Online

```
$ make random
./kubectl apply -f istio/random_lb.yaml
destinationrule.networking.istio.io/cpanel configured
$ k get destinationrules cpanel -o yaml
```

```
$ make load
for i in {1..100}; do sleep 0.2; curl http://192.168.99.113:31221/status; printf "\n"; done
CPanel v2 : Online
CPanel v2 : Online
CPanel v1 : Online
CPanel v2 : Online
CPanel v1 : Online
CPanel v2 : Online
CPanel v1 : Online
CPanel v2 : Online
```

```
$ make all-reset
./kubectl delete service --all
```

Fault injection

Internal istio mechanism for chaos testing. Allows simulating network and service errors without touching the source code of microservice at all. All faults are done by sidecar Envoy proxy.

```
http:
- fault:
  abort:
    httpStatus: 503
    percentage:
      value: 50
route:
- destination:
  host: alerts.default.svc.cluster.local
  subset: v1

route:
- destination:
  host: cpanel.default.svc.cluster.local
  port:
    number: 8080
  subset: v1
  weight: 50
  fault:
    delay:
      fixedDelay: 10s
      percentage:
        value: 50
- destination:
  host: cpanel.default.svc.cluster.local
  port:
    number: 8080
  subset: v2
```

weight: 50

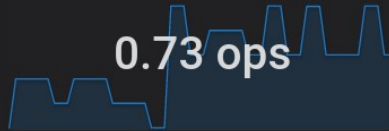
client workloads - workloads that are calling this service

service workloads - workloads that are providing this service

\$ make fault-injection-500

SERVICE: image-analysis.default.svc.cluster.local

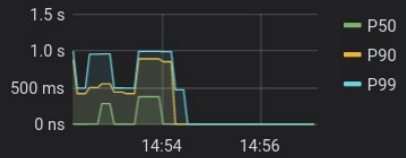
Client Request Volume



Client Success Rate (non-5xx responses...)



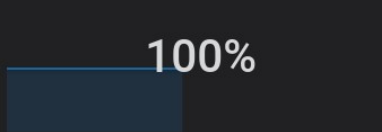
Client Request Duration



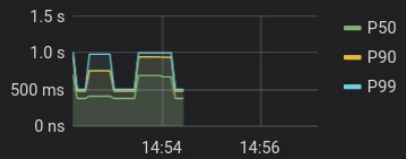
Server Request Volume



Server Success Rate (non-5xx responses...)

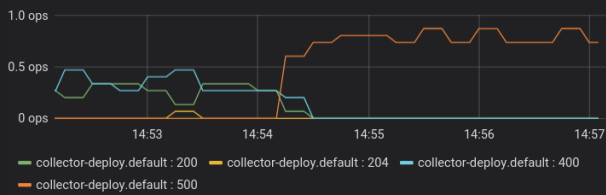


Server Request Duration

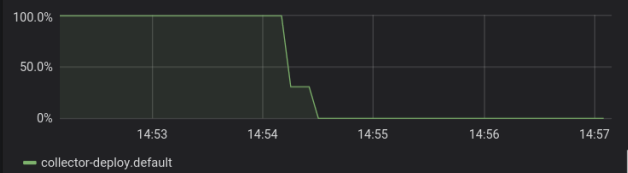


CLIENT WORKLOADS

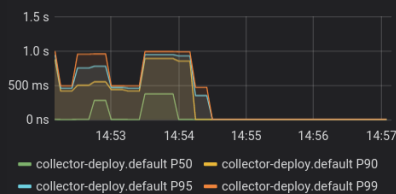
Incoming Requests by Source And Response Code



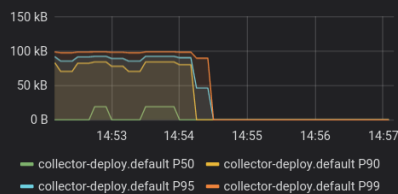
Incoming Success Rate (non-5xx responses) By Source



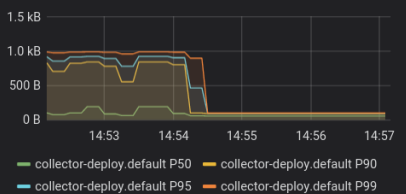
Incoming Request Duration by Source



Incoming Request Size By Source

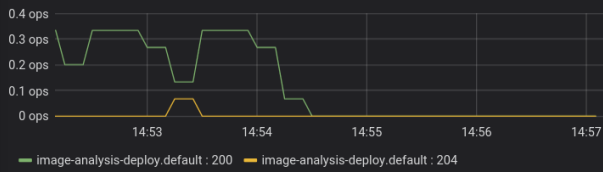


Response Size By Source



SERVICE WORKLOADS

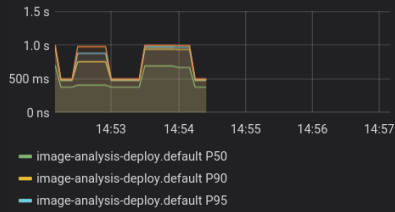
Incoming Requests by Destination And Response Code



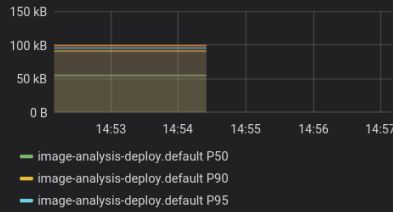
Incoming Success Rate (non-5xx responses) By Source



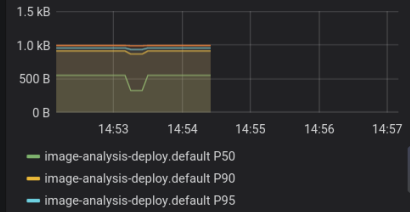
Incoming Request Duration by Source



Incoming Request Size By Source



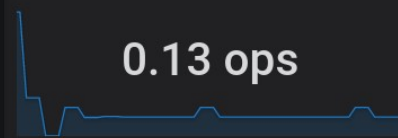
Response Size By Source



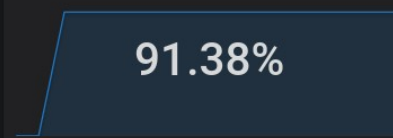
\$ make fault-injection-delay10

SERVICE: image-analysis.default.svc.cluster.local

Client Request Volume



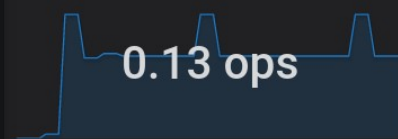
Client Success Rate (non-5xx respons...



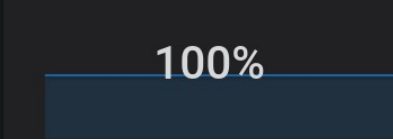
Client Request Duration



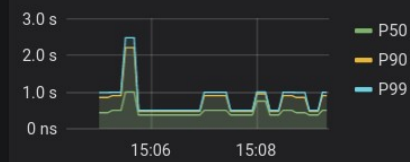
Server Request Volume



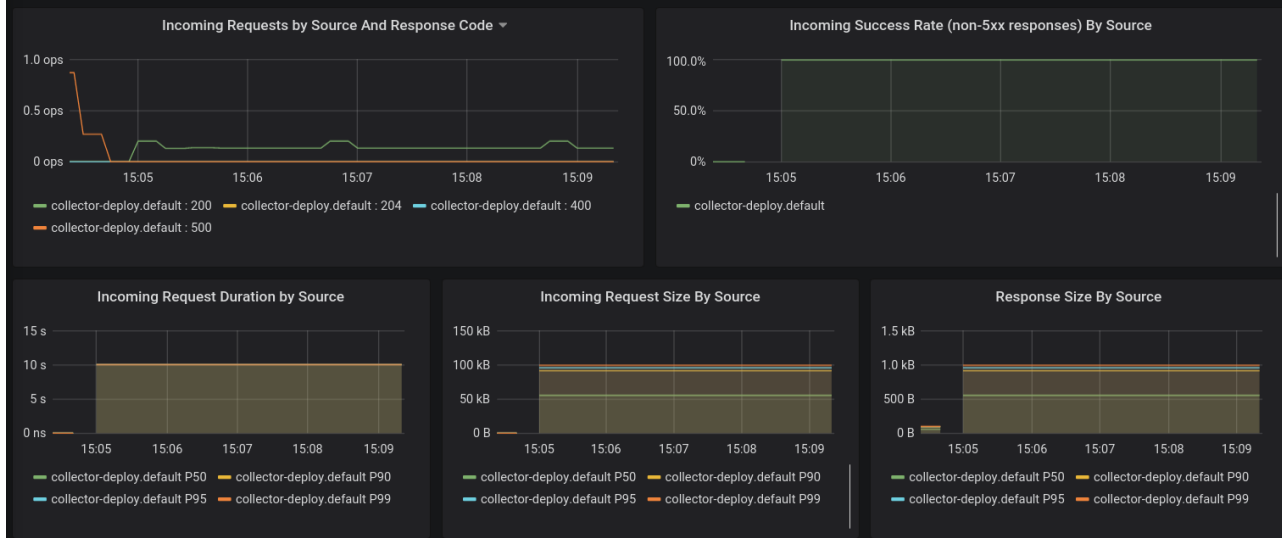
Server Success Rate (non-5xx respons...



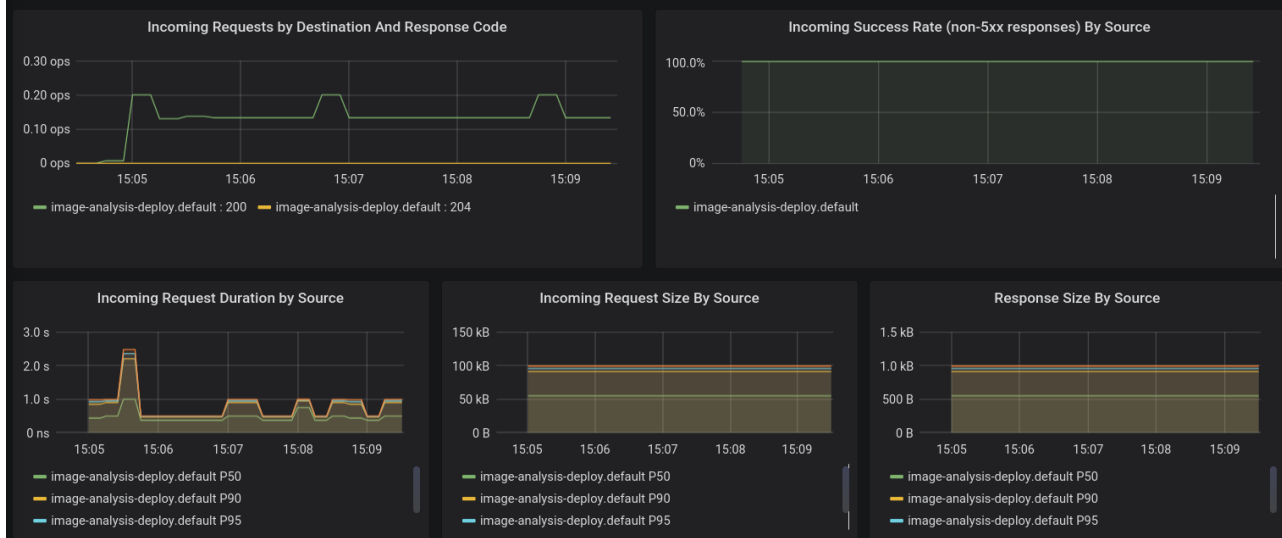
Server Request Duration



CLIENT WORKLOADS



SERVICE WORKLOADS



Timeout

\$ make timeout

./kubectrl apply -f istio/timeout.yaml

virtualservice.networking.istio.io/camera-agent-1 configured

virtualservice.networking.istio.io/cpanel configured

```

20:47 $ make health-timeout
for i in {1..10}; do sleep 0.2; curl http://192.168.99.113:31221/cameras/1/state; printf "\n"; done
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
upstream request timeout
upstream request timeout
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
upstream request timeout
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
upstream request timeout
upstream request timeout
upstream request timeout
upstream request timeout

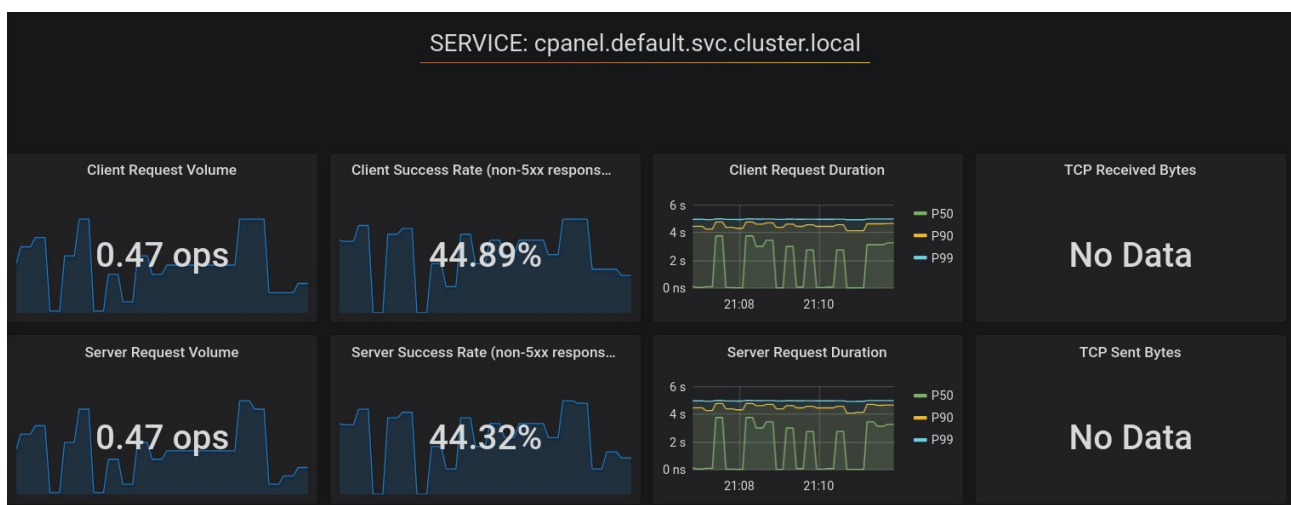
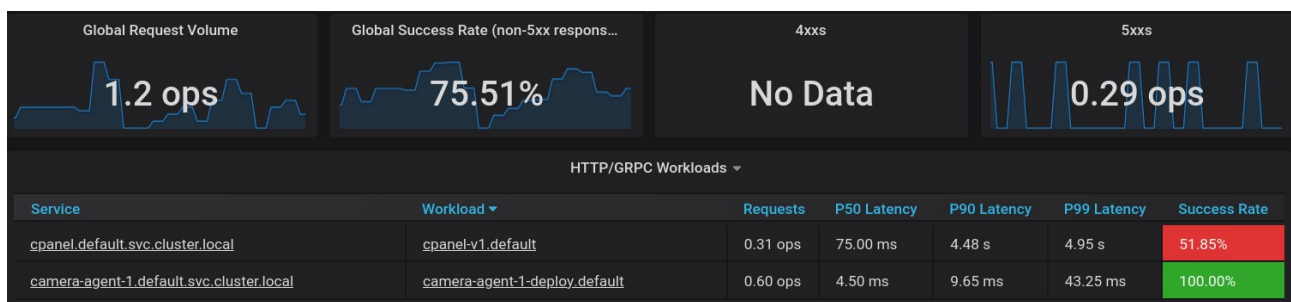
```

```

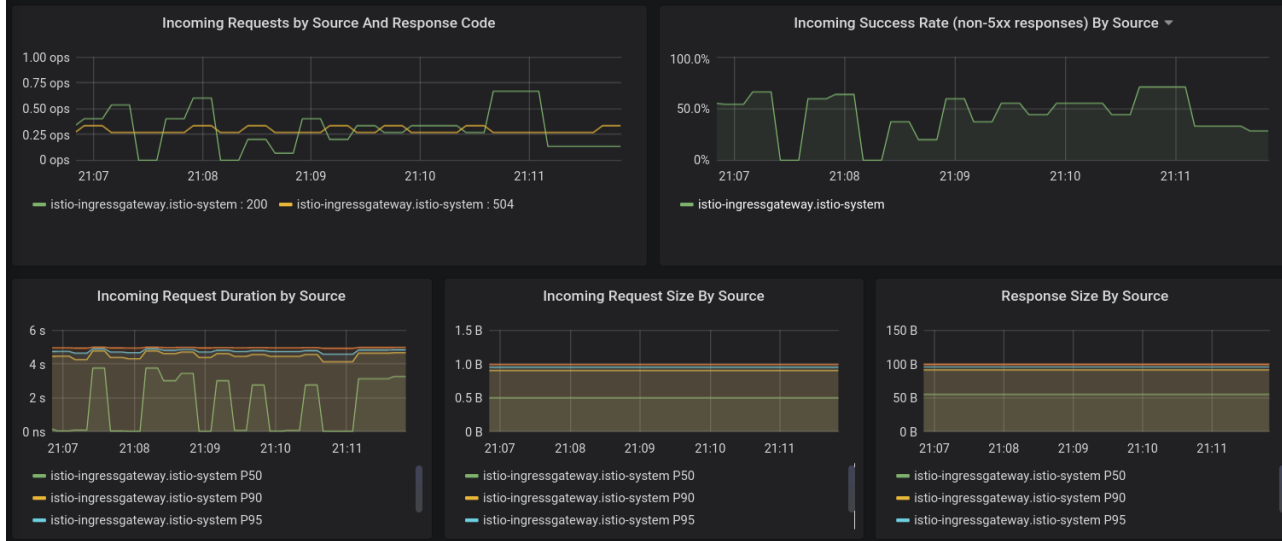
$ make health-timeout
for i in {1..10}; do sleep 0.2; curl http://192.168.99.113:31221/cameras/1/state; printf "\n"; done
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
upstream request timeout
upstream request timeout
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
upstream request timeout
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
upstream request timeout
upstream request timeout
upstream request timeout

```

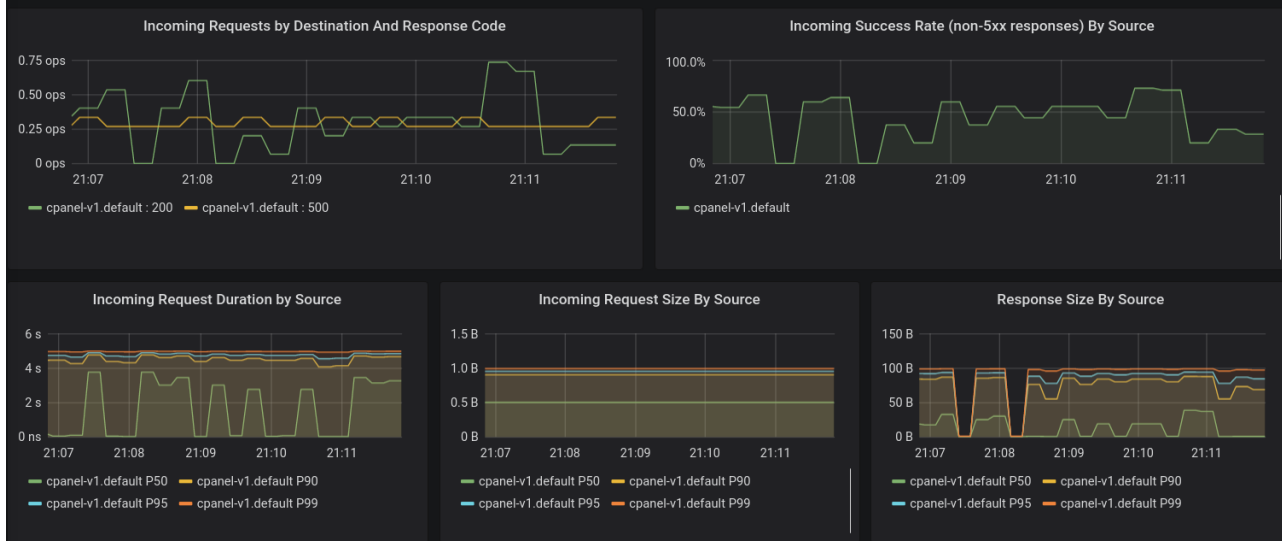
grafana with 1000 requests



CLIENT WORKLOADS



SERVICE WORKLOADS



Retries

\$ make retries

./kubectly apply -f istio/retry.yaml

virtualservice.networking.istio.io/collector configured

virtualservice.networking.istio.io/section-1 configured

\$ make health-retries

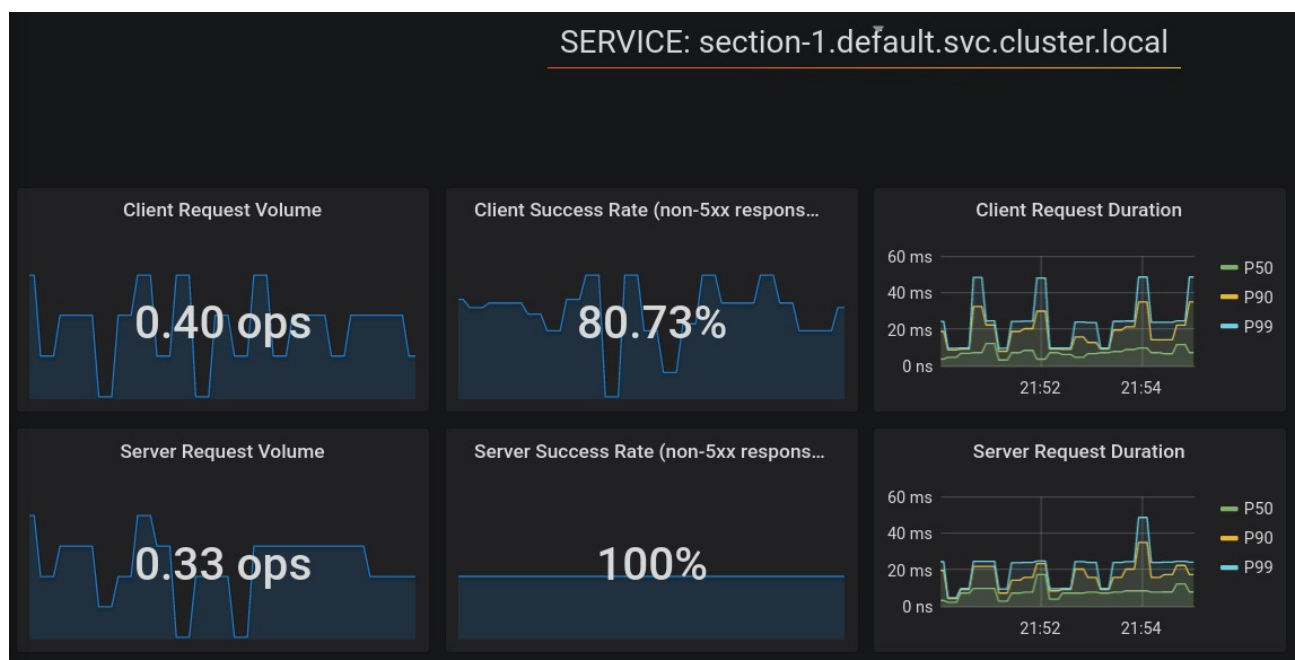
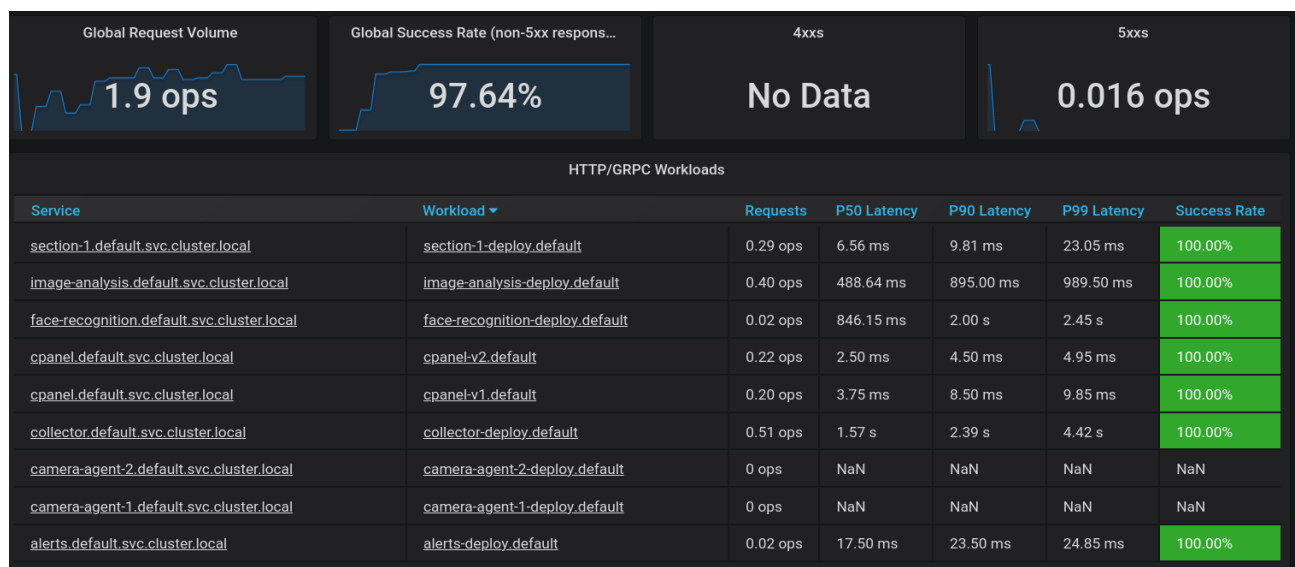
for i in {1..10}; do sleep 0.2; curl http://192.168.99.113:31221/sections/1/status; printf "\n"; done

Section 1 v1 : Online

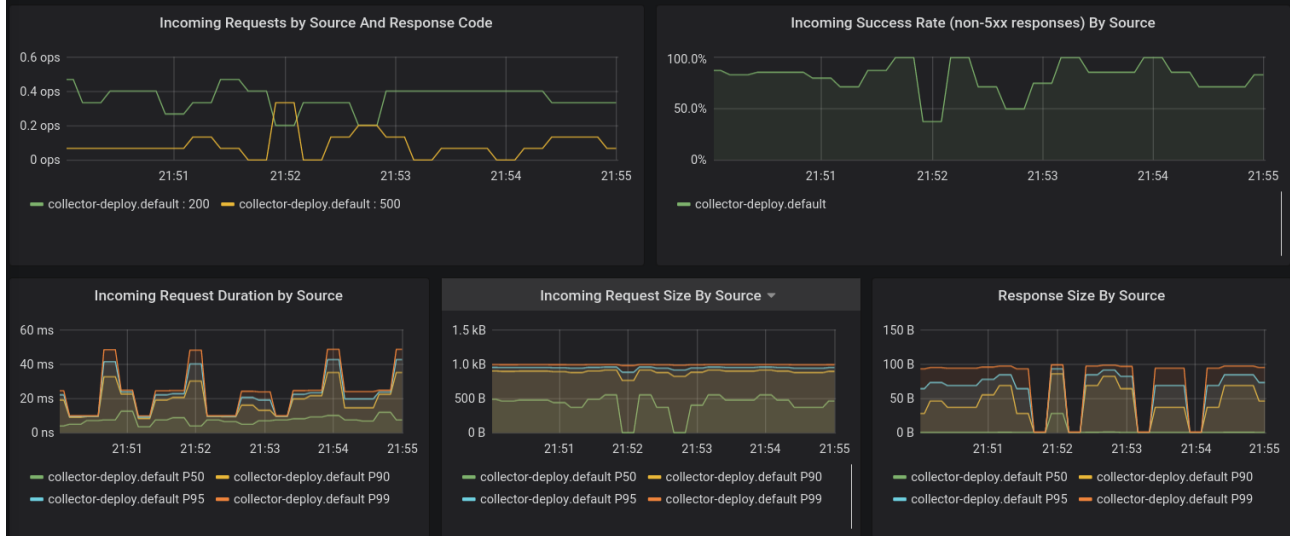
Section 1 v1 : Online

Section 1 v1 : Online
fault filter abort
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online

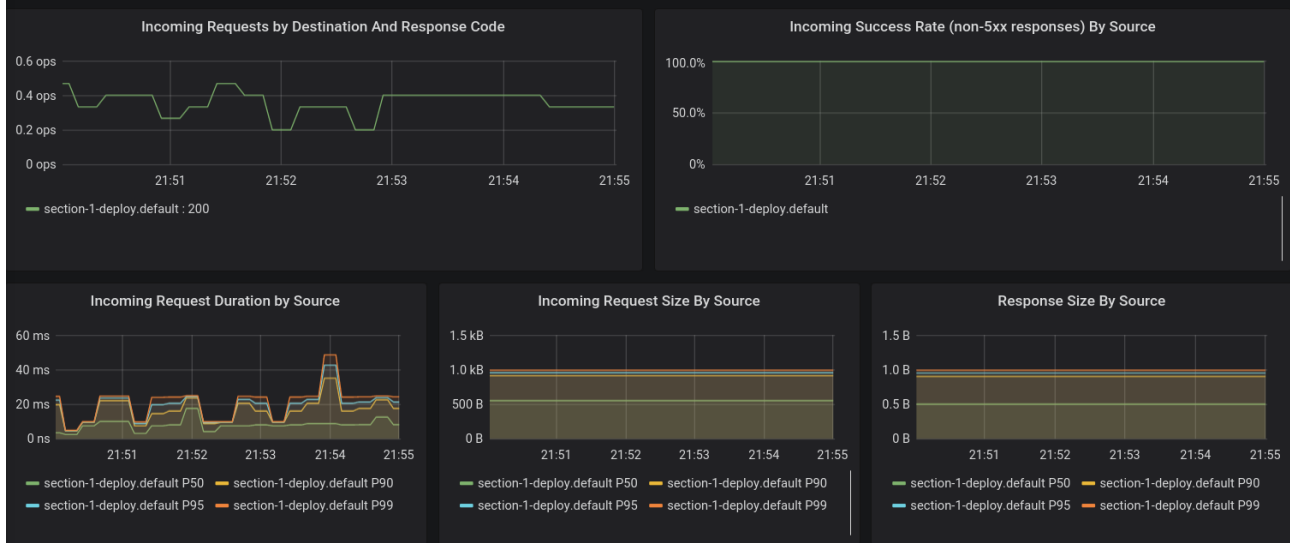
```
21:48 $ make health-retries host: section-1.default.svc.cluster.local
for i in $(seq 1 10); do sleep 0.2; curl http://192.168.99.113:31221/sections/1/status; printf "\n"; done
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
fault filter abort
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
```



CLIENT WORKLOADS



SERVICE WORKLOADS



Circuit breaker

Discussion

Conclusion

- pros of istio resiliency features
- expanse of service meshes
- complexity of operations (# of micro services, agile)
- advices

- move to production step by step incremental, complexity of debugging
- adopt istio only if you have a use case that can be solved through it
- configure log level to error – otherwise too much traffic \$\$\$

Future Work

References

1. (rest)Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
2. (fowler_msvc)<https://www.martinfowler.com/articles/microservices.html>
3. (images)<https://snyk.io/blog/10-docker-image-security-best-practices/>
4. (cc)Cloud computing assignment
5. (twelve)<https://12factor.net/>
6. (k8s)<https://kubernetes.io/>
7. (istio)<https://istio.io/>
8. (docker)<https://www.docker.com/>
9. (alt)<https://aspenmesh.io/service-mesh-architectures/>
10. (tele)<https://www.telepresence.io/>

Supplemental Material

- cc assignment
- commands