Abstract

Expanse of cloud computing, new challenges for developers,
such as rethinking of old monolith architectures, transition to microservices
cloud native development for completely new projects
need for resiliency without interactions with operations department
In this thesis: service mesh as a solution, demo, pros of istio – resiliency and fault injection

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
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

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)

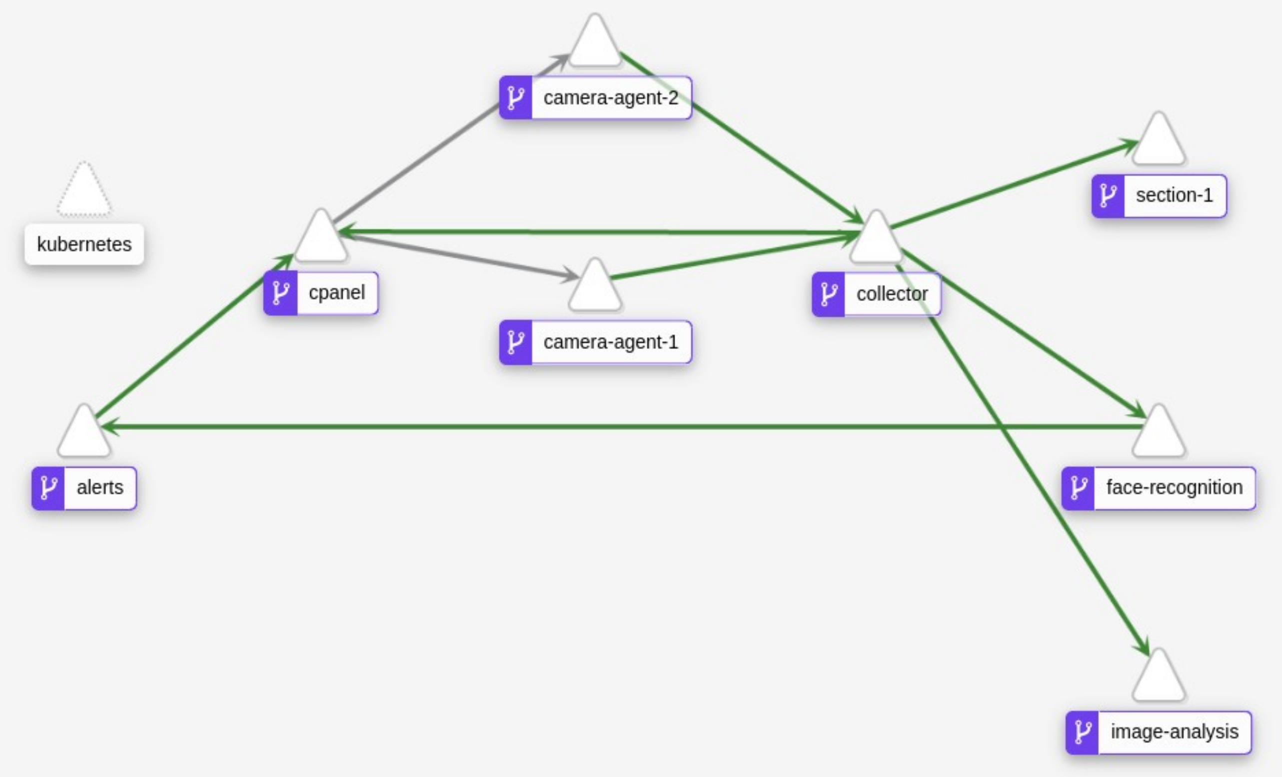
Major idea

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

Deploy with minikube

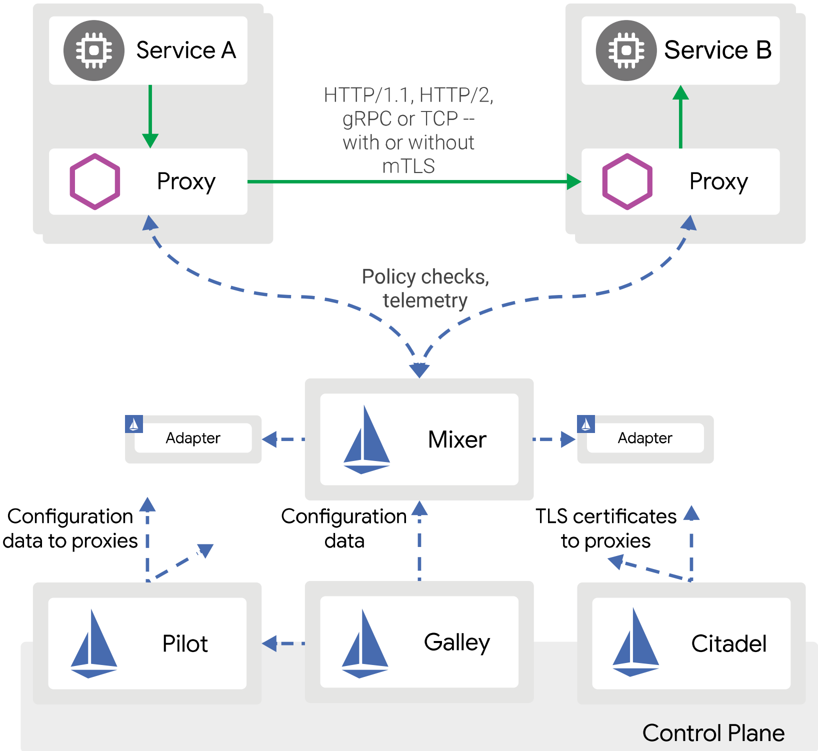
service mesh in general

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

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, faut injection
destination rules – named subsets, circuit breaker, load balancing
gateways – Virtual Service to allow L7 routing, use defaut 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 possibilities:
        - + 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 repo, minikube (specific versions), istio, shell scripts, makefile

Implementation

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

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

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

play around:

Evaluation and Discussion
- 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.

Istio routing mechanism
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.

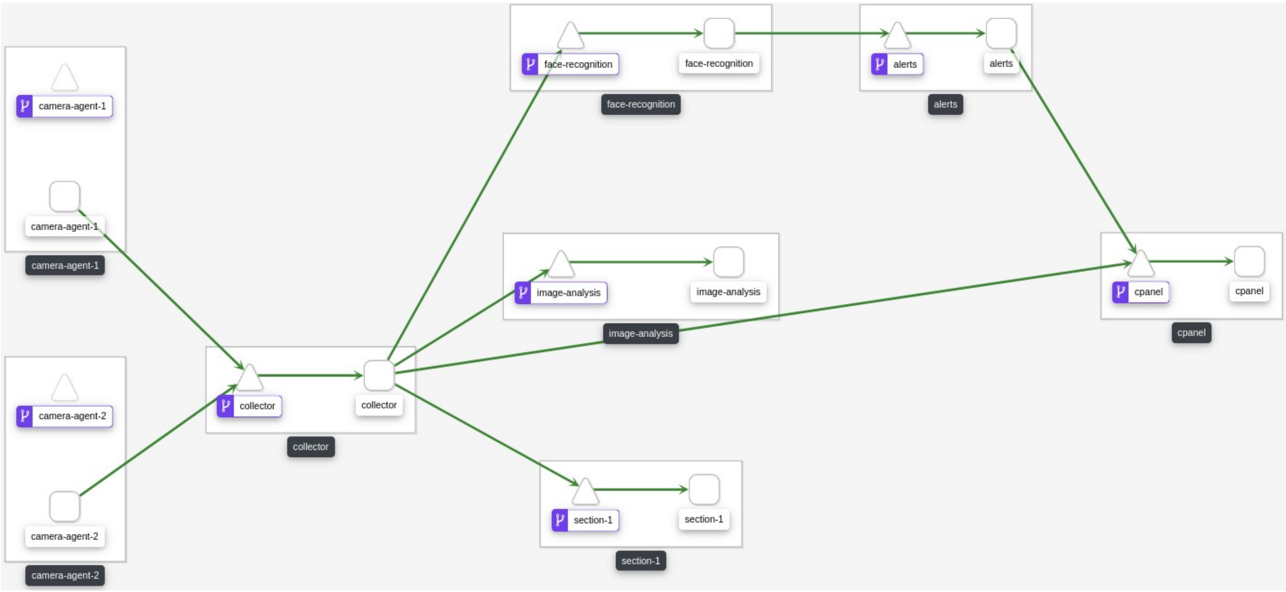
```
$ make deploy-app-default
$ make deploy-istio-default
$ make health
```

```
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
```

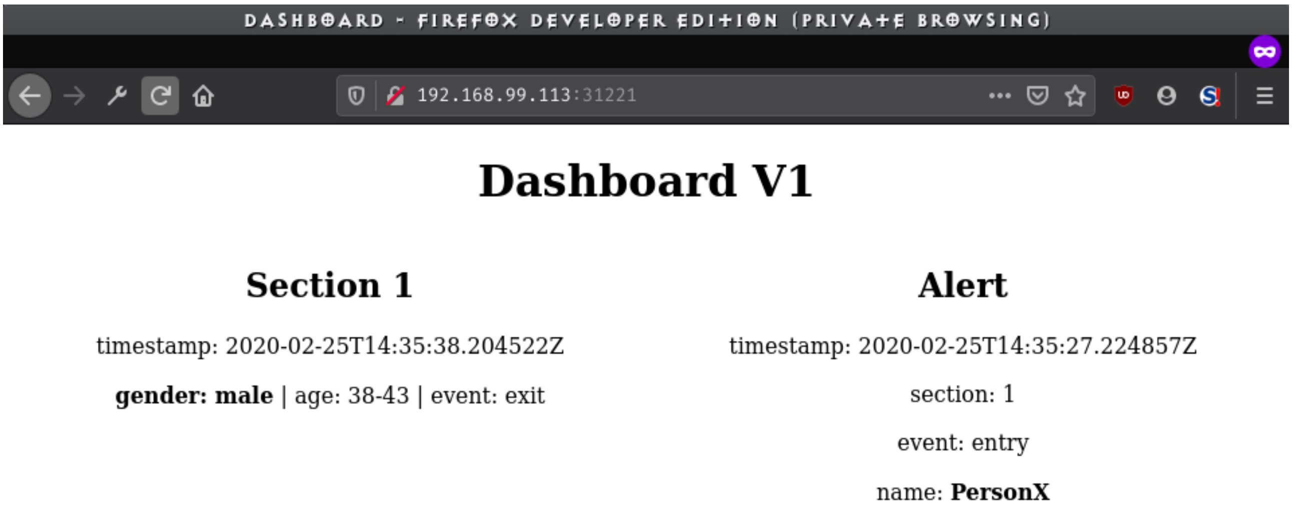
$ make start-cameras
$ make health

```
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
```

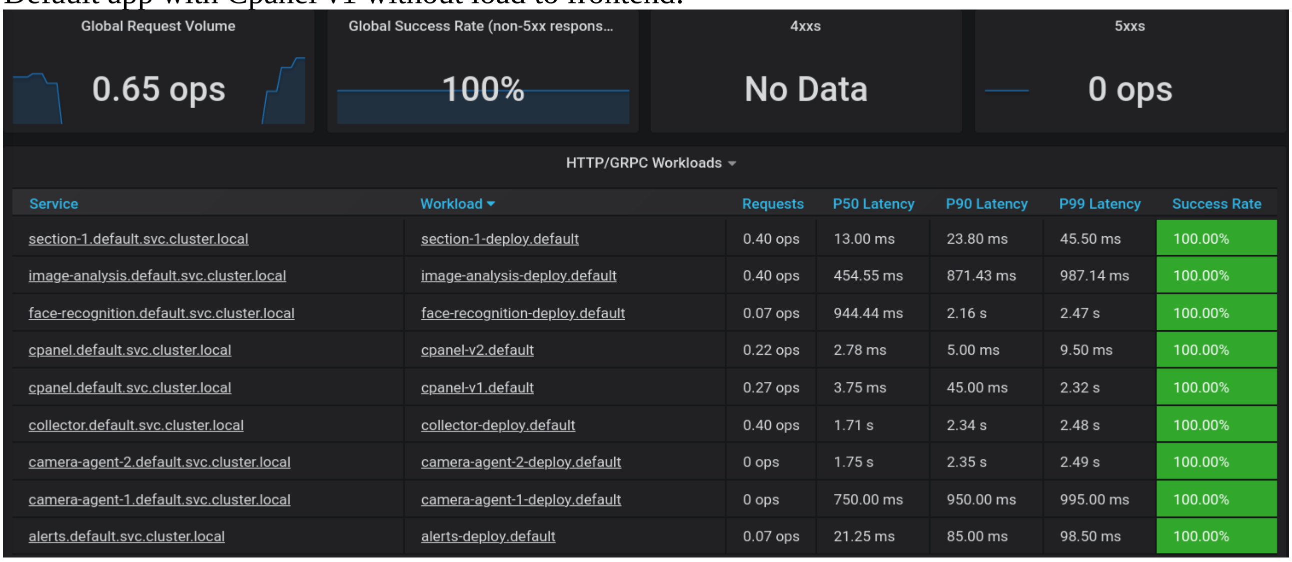
curl http://192.168.99.113:31221/status
CPanel v1 : Online
curl http://192.168.99.113:31221/cameras/1/state
{"streaming":true,"cycle":7,"fps":0,"section":"1","destination":"http://collector.default.svc.cluster.local:8080","event":"exit"}
curl http://192.168.99.113:31221/cameras/2/state
{"streaming":true,"cycle":5,"fps":0,"section":"1","destination":"http://colle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

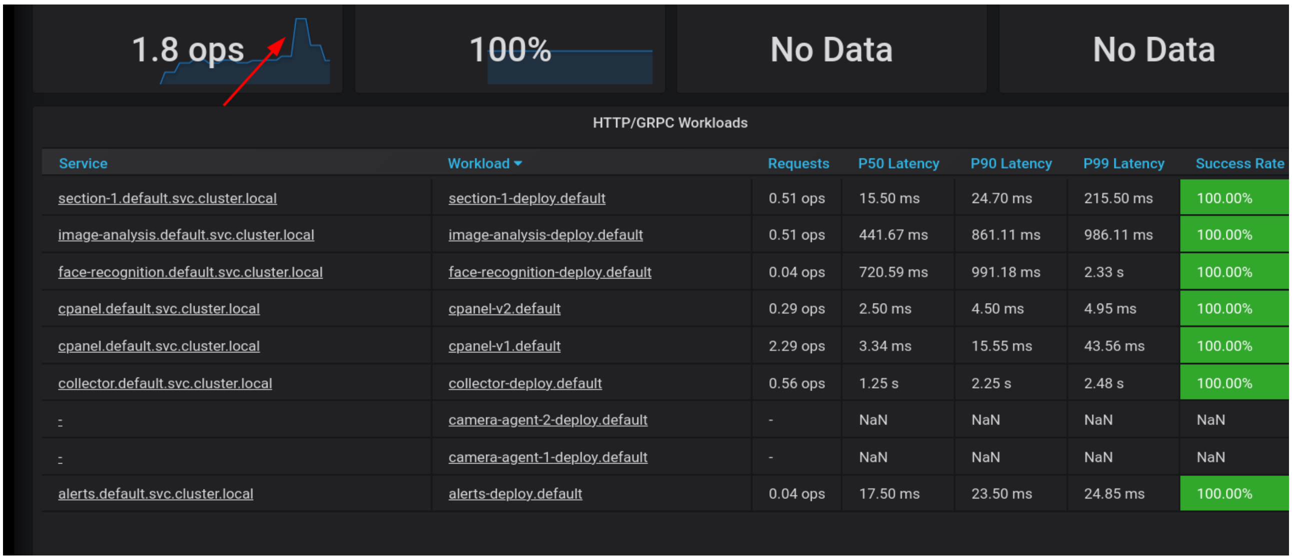


Default app with Cpanel v1 without load to frontend:



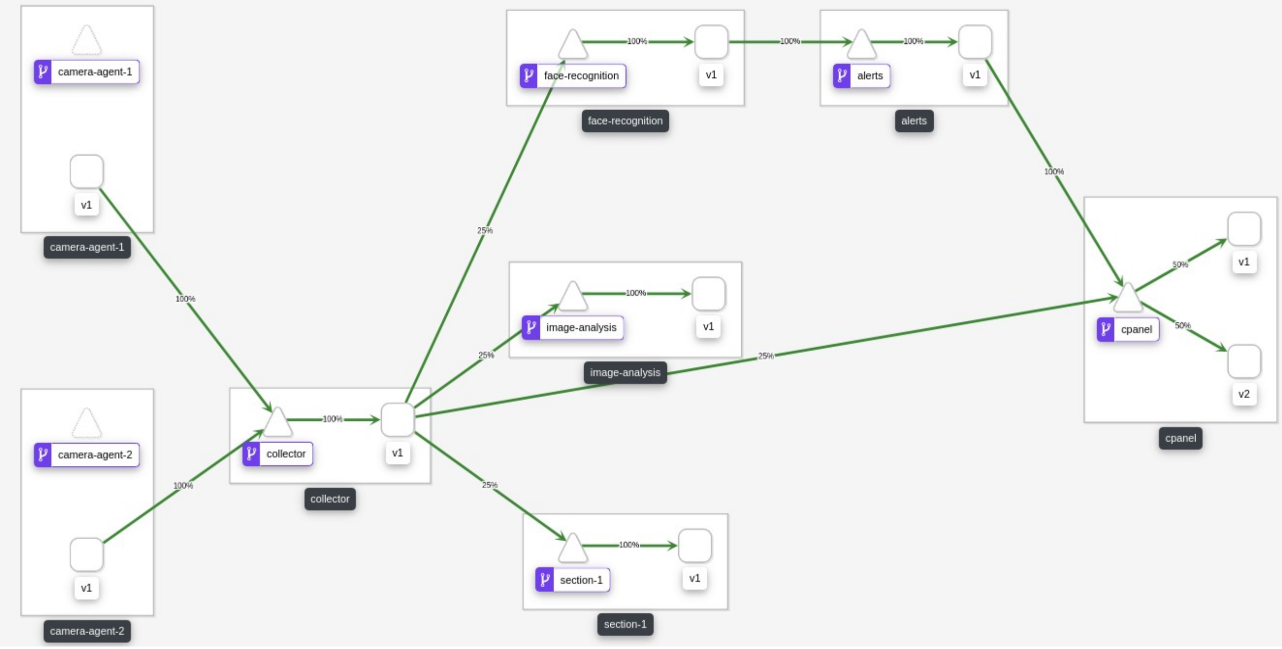
Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default	0.40 ops	13.00 ms	23.80 ms	45.50 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.40 ops	454.55 ms	871.43 ms	987.14 ms	100.00%
face-recognition.default.svc.cluster.local	face-recognition-deploy.default	0.07 ops	944.44 ms	2.16 s	2.47 s	100.00%
cpanel.default.svc.cluster.local	cpanel-v2.default	0.22 ops	2.78 ms	5.00 ms	9.50 ms	100.00%
cpanel.default.svc.cluster.local	cpanel-v1.default	0.27 ops	3.75 ms	45.00 ms	2.32 s	100.00%
collector.default.svc.cluster.local	collector-deploy.default	0.40 ops	1.71 s	2.34 s	2.48 s	100.00%
camera-agent-2.default.svc.cluster.local	camera-agent-2-deploy.default	0 ops	1.75 s	2.35 s	2.49 s	100.00%
camera-agent-1.default.svc.cluster.local	camera-agent-1-deploy.default	0 ops	750.00 ms	950.00 ms	995.00 ms	100.00%
alerts.default.svc.cluster.local	alerts-deploy.default	0.07 ops	21.25 ms	85.00 ms	98.50 ms	100.00%

```
$ 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
…
```

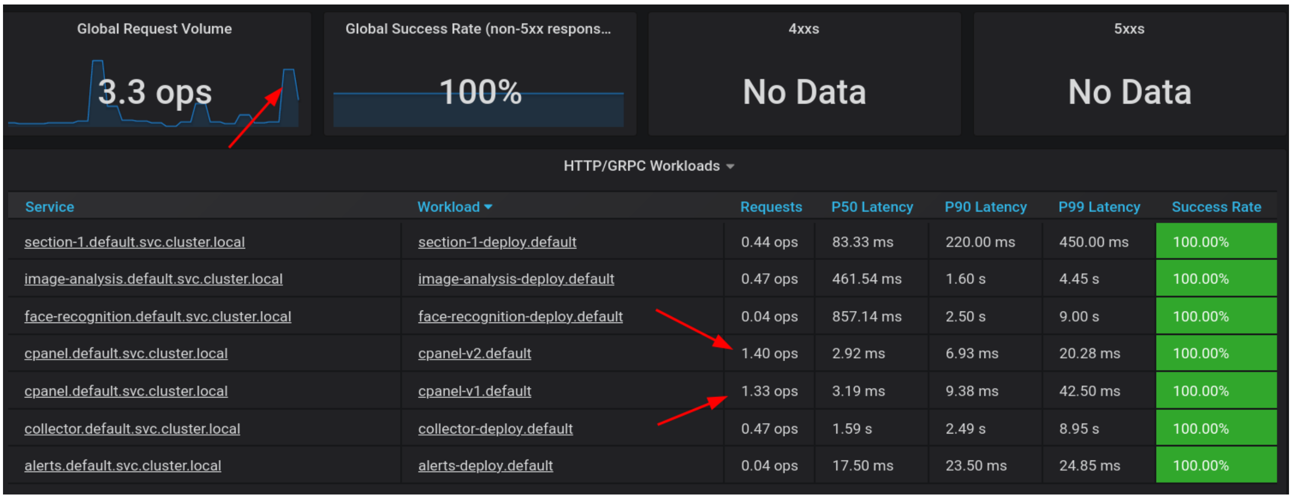


Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default	0.51 ops	15.50 ms	24.70 ms	215.50 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.51 ops	441.67 ms	861.11 ms	986.11 ms	100.00%
face-recognition.default.svc.cluster.local	face-recognition-deploy.default	0.04 ops	720.59 ms	991.18 ms	2.33 s	100.00%
cpanel.default.svc.cluster.local	cpanel-v2.default	0.29 ops	2.50 ms	4.50 ms	4.95 ms	100.00%
cpanel.default.svc.cluster.local	cpanel-v1.default	2.29 ops	3.34 ms	15.55 ms	43.56 ms	100.00%
collector.default.svc.cluster.local	collector-deploy.default	0.56 ops	1.25 s	2.25 s	2.48 s	100.00%
-	camera-agent-2-deploy.default	-	NaN	NaN	NaN	NaN
-	camera-agent-1-deploy.default	-	NaN	NaN	NaN	NaN
alerts.default.svc.cluster.local	alerts-deploy.default	0.04 ops	17.50 ms	23.50 ms	24.85 ms	100.00%

```
$ 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



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%

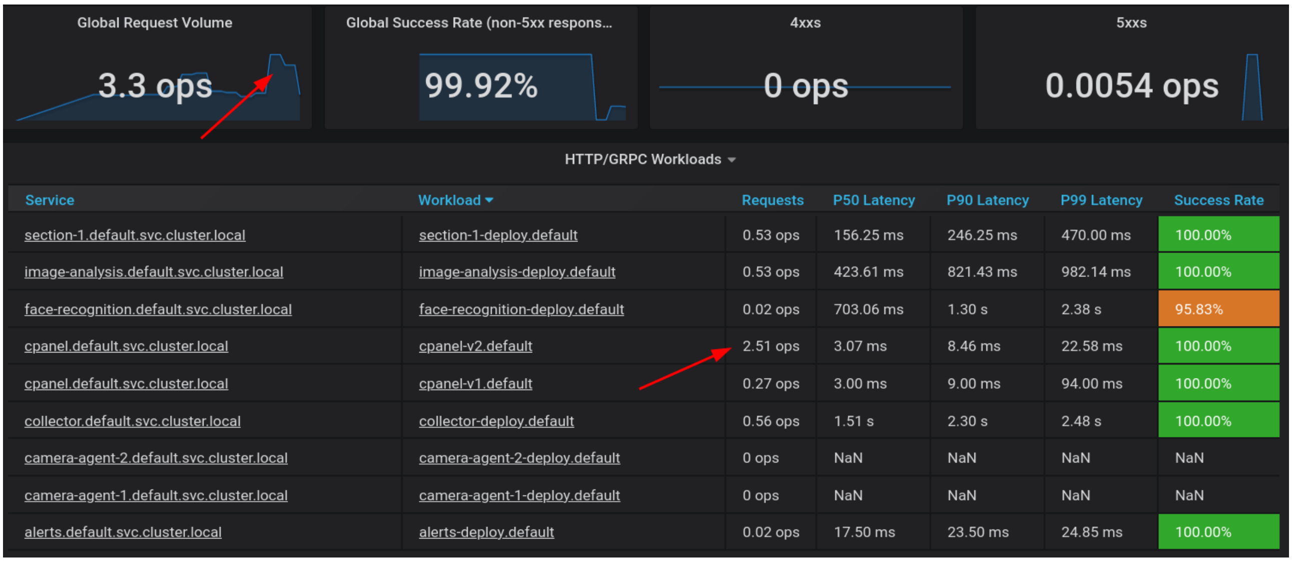
$ make cpanel-v2
./kubectl apply -f istio/virt_svc_v2.yaml
virtualservice.networking.istio.io/cpanel configured
check configuration
$ k get virtualservices cpanel -o yaml

```
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
```


```
$ 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
```



Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default	0.53 ops	156.25 ms	246.25 ms	470.00 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.53 ops	423.61 ms	821.43 ms	982.14 ms	100.00%
face-recognition.default.svc.cluster.local	face-recognition-deploy.default	0.02 ops	703.06 ms	1.30 s	2.38 s	95.83%
cpanel.default.svc.cluster.local	cpanel-v2.default	2.51 ops	3.07 ms	8.46 ms	22.58 ms	100.00%
cpanel.default.svc.cluster.local	cpanel-v1.default	0.27 ops	3.00 ms	9.00 ms	94.00 ms	100.00%
collector.default.svc.cluster.local	collector-deploy.default	0.56 ops	1.51 s	2.30 s	2.48 s	100.00%
camera-agent-2.default.svc.cluster.local	camera-agent-2-deploy.default	0 ops	NaN	NaN	NaN	NaN
camera-agent-1.default.svc.cluster.local	camera-agent-1-deploy.default	0 ops	NaN	NaN	NaN	NaN
alerts.default.svc.cluster.local	alerts-deploy.default	0.02 ops	17.50 ms	23.50 ms	24.85 ms	100.00%

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

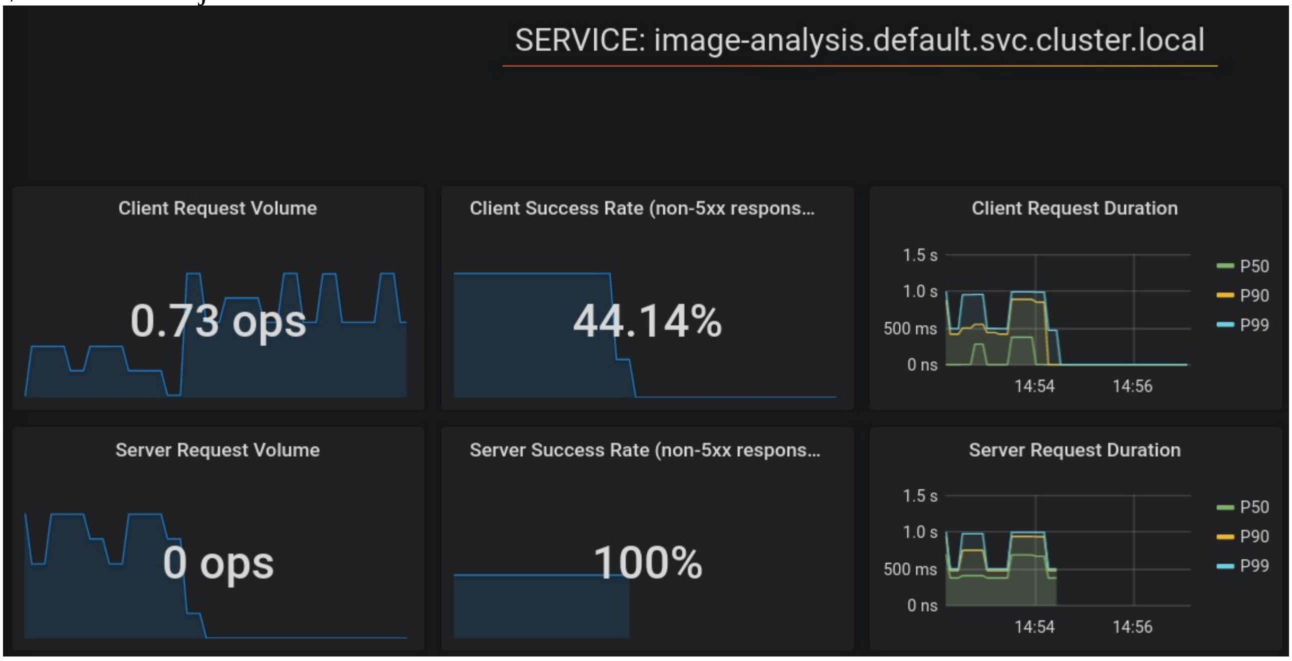
$ make all-reset
./kubectl delete service –all

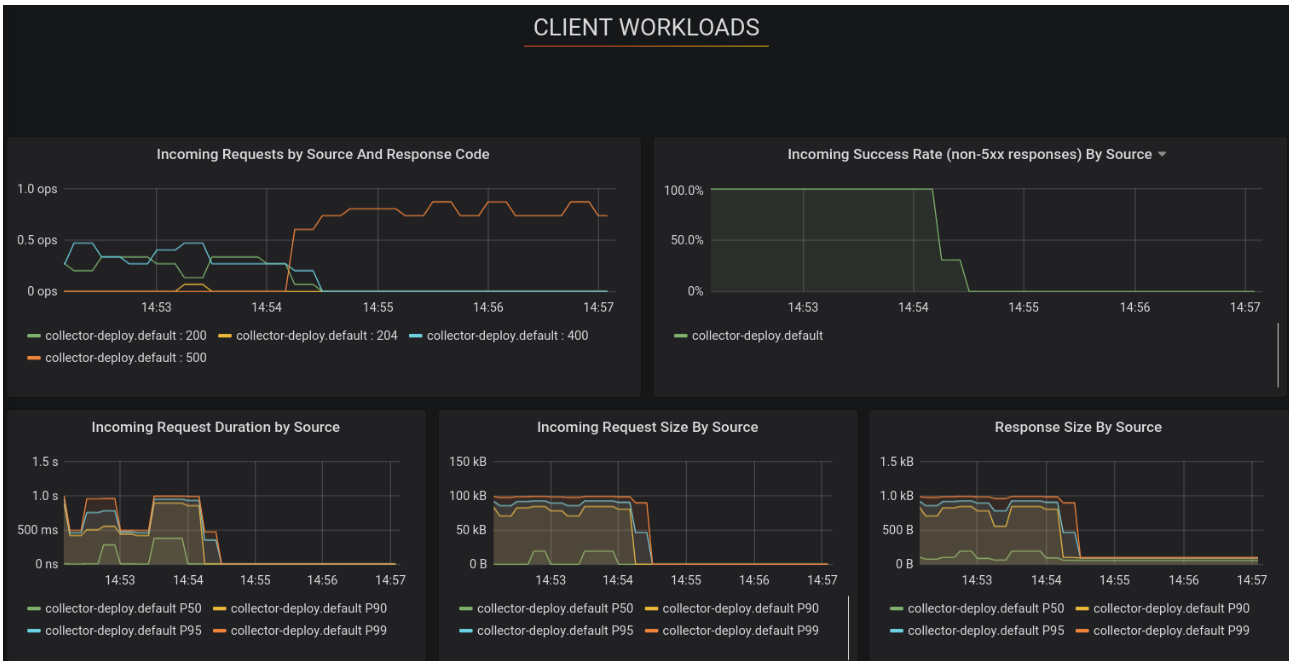
fault injection
Internal istio mechanism for chaos testing. Allows simulating network and service errors without touching the source code of microservice at all. All faults are done by sidecar proxy.

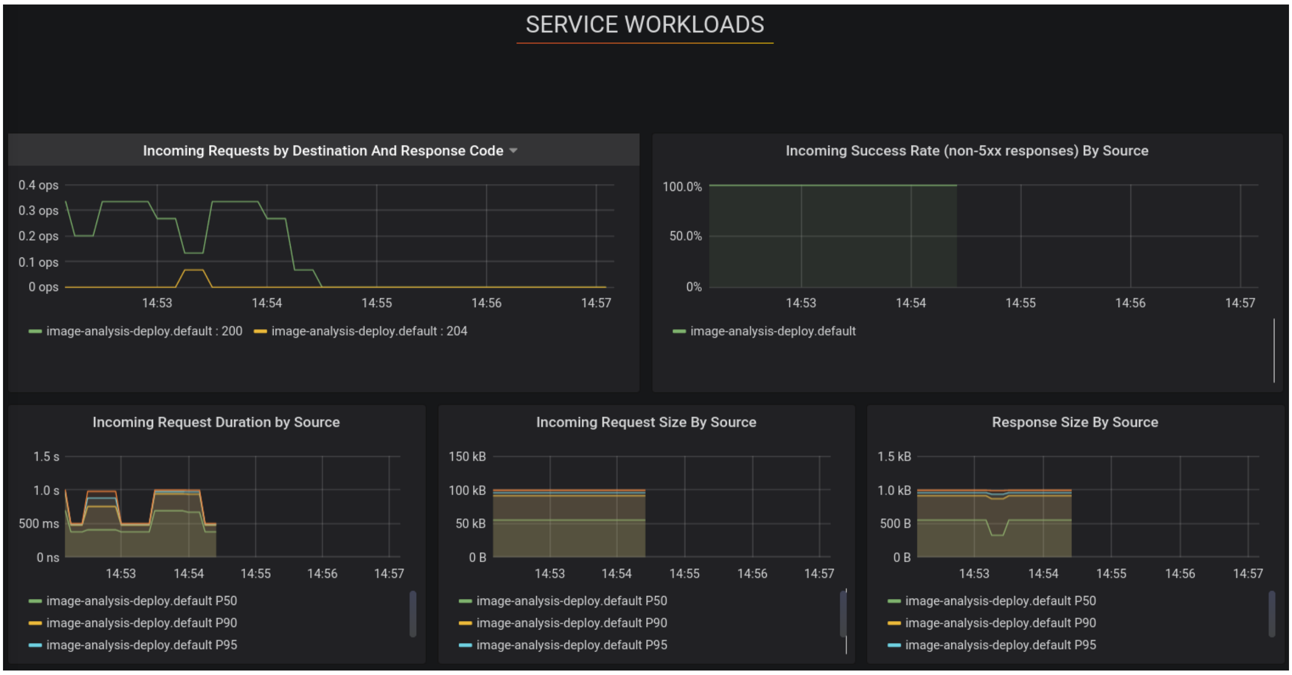
```
http:
- fault:
    abort:
      httpStatus: 503
      percentage:
        value: 50
route:
- destination:
    host: alerts.default.svc.cluster.local
    subset: v1

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

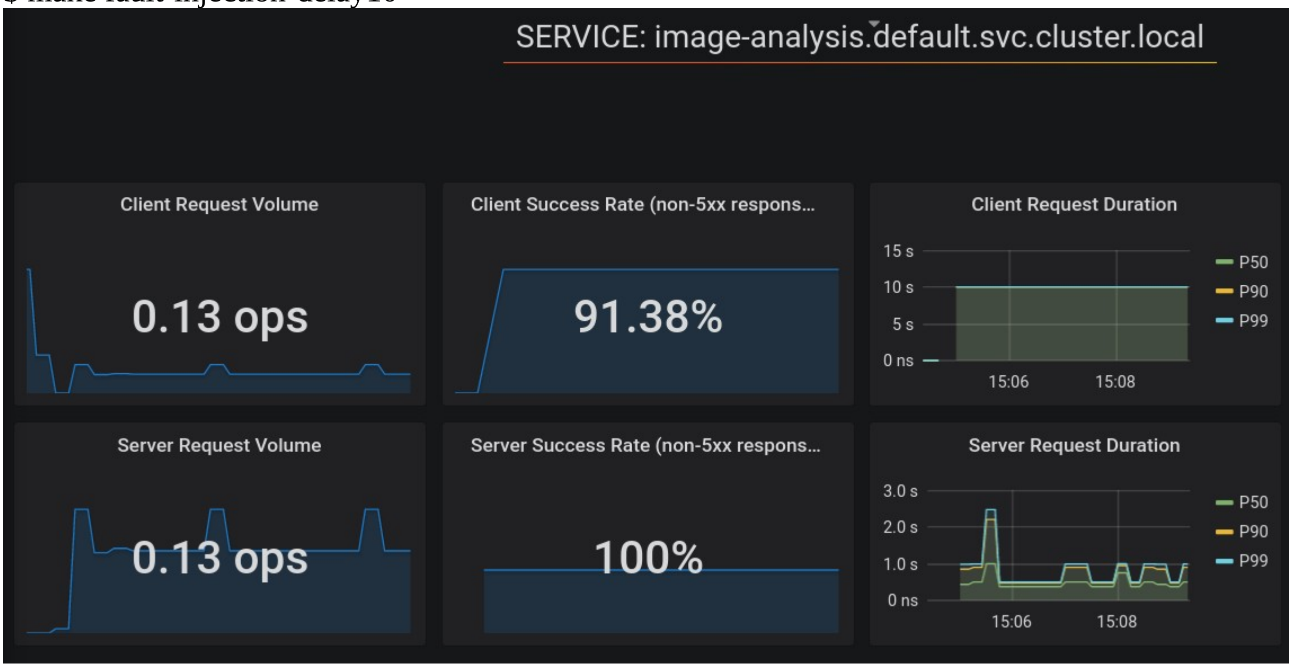
$ make fault-injection-500

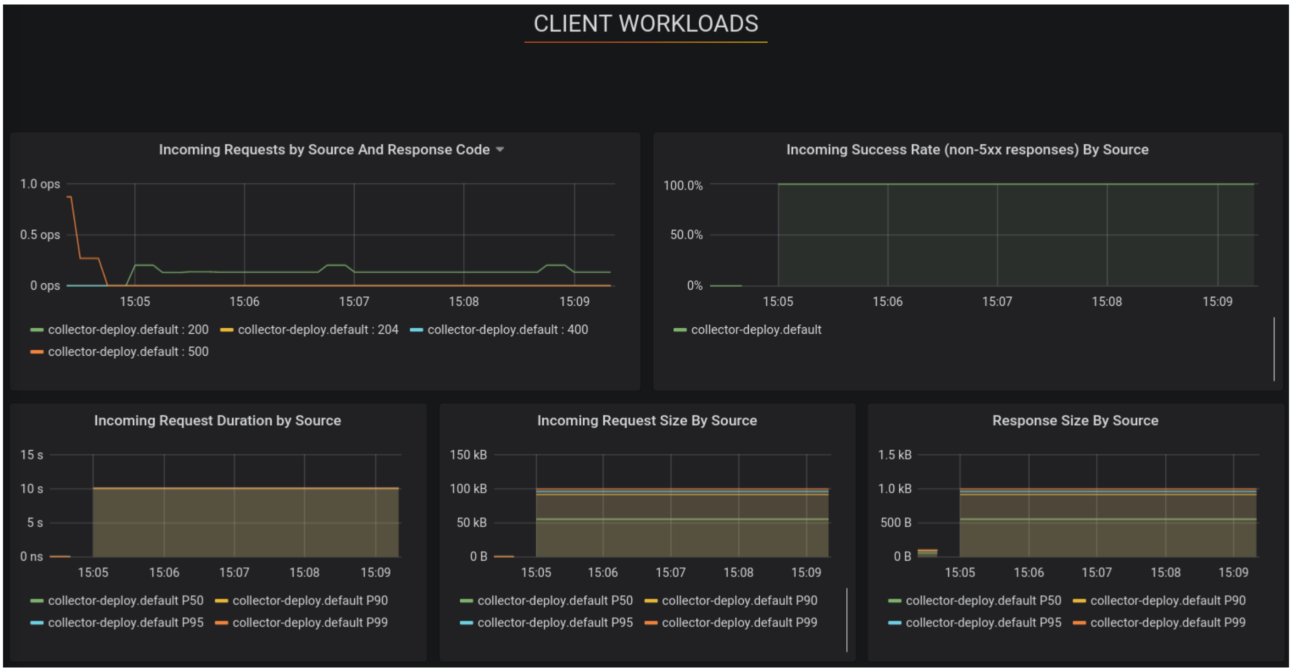


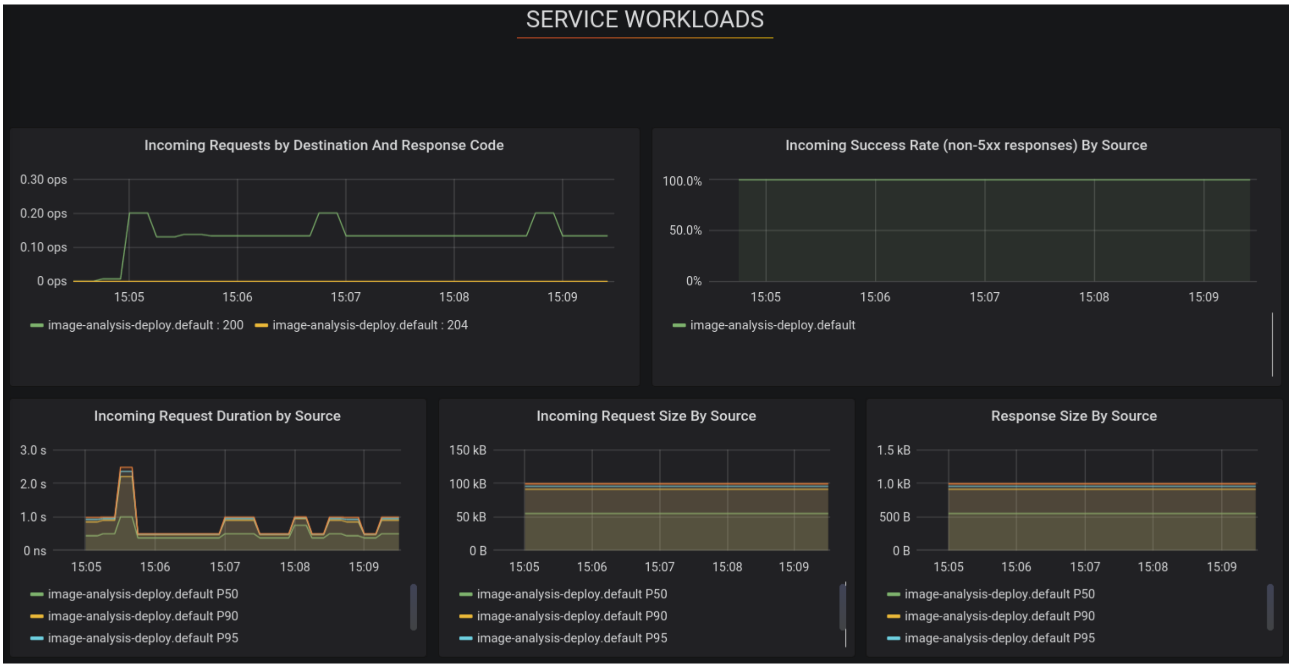




$ make fault-injection-delay10





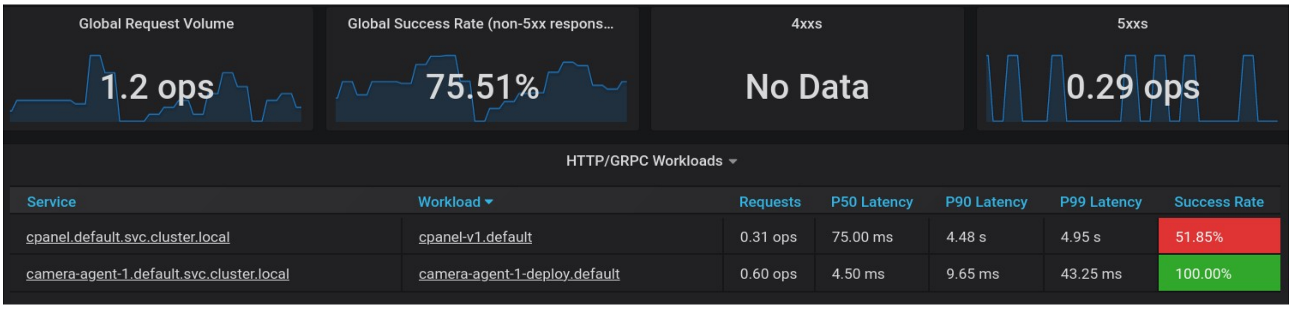


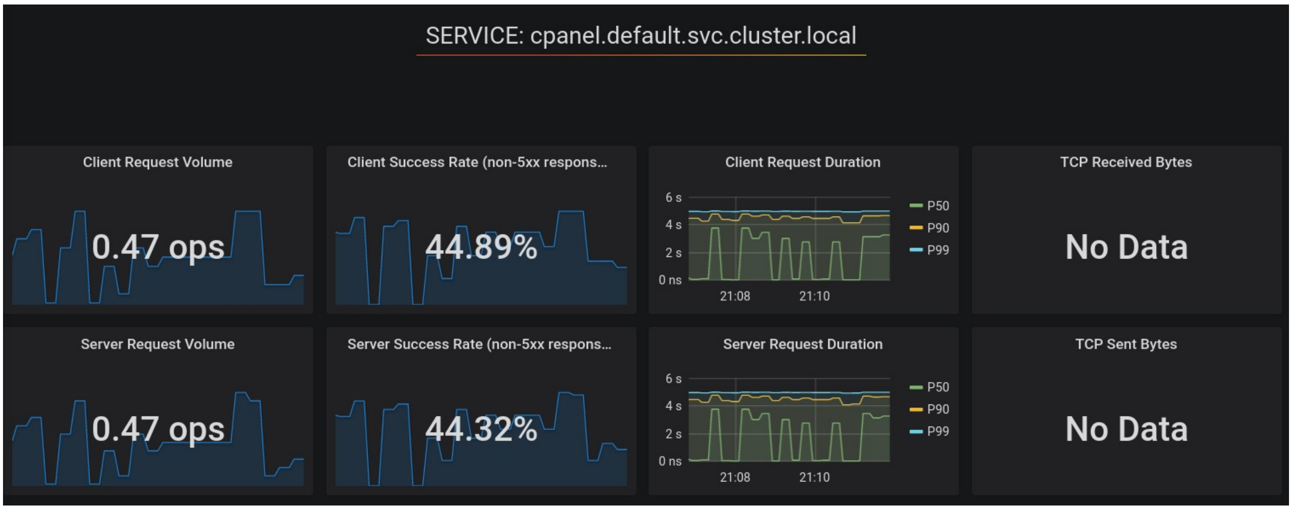
timeout

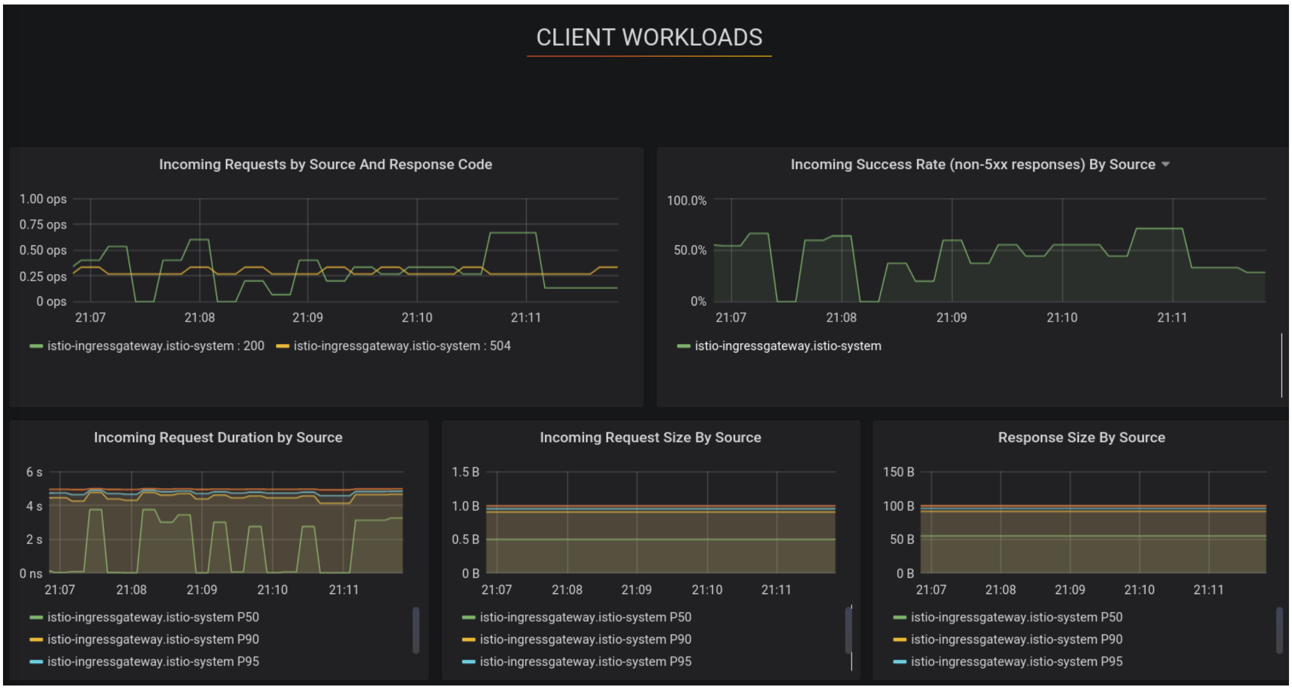
```
$ make timeout
./kubectl apply -f istio/timeout.yaml
virtualservice.networking.istio.io/camera-agent-1 configured
virtualservice.networking.istio.io/cpanel configured
```

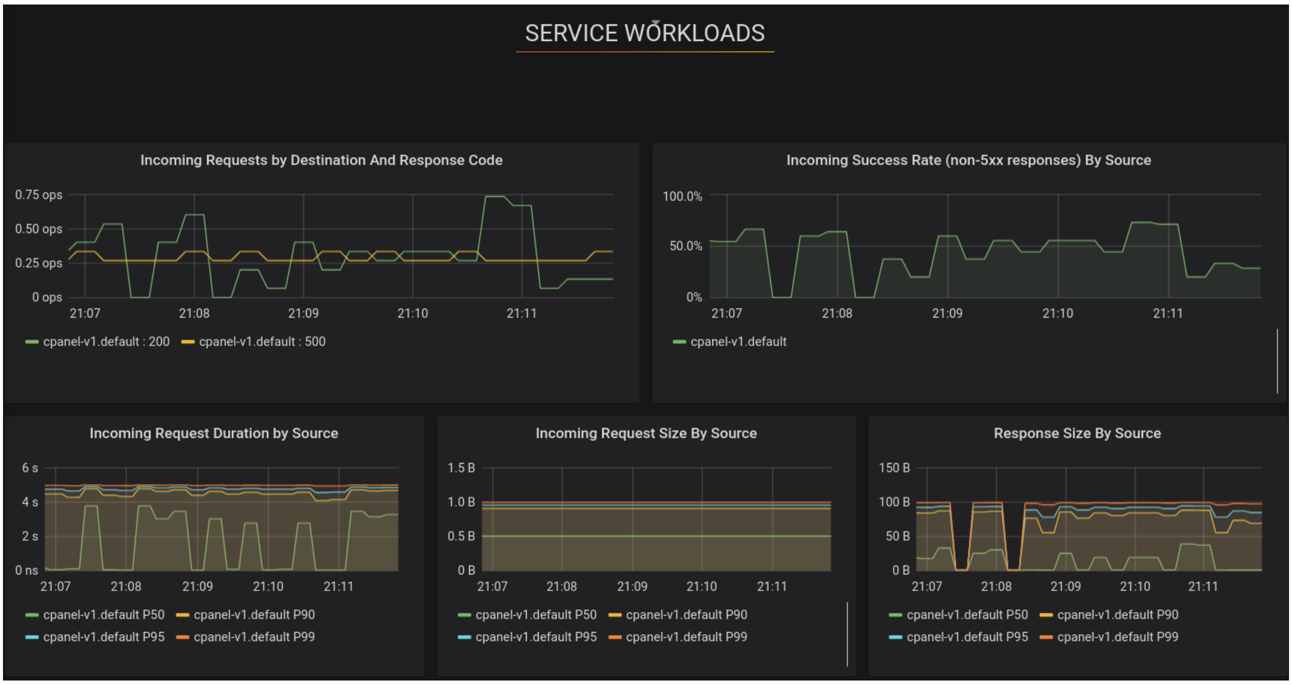
$ make health-timeout
for i in {1..10}; do sleep 0.2; curl http://192.168.99.113:31221/cameras/1/state; printf "\n"; done
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"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

grafana with 1000 requests





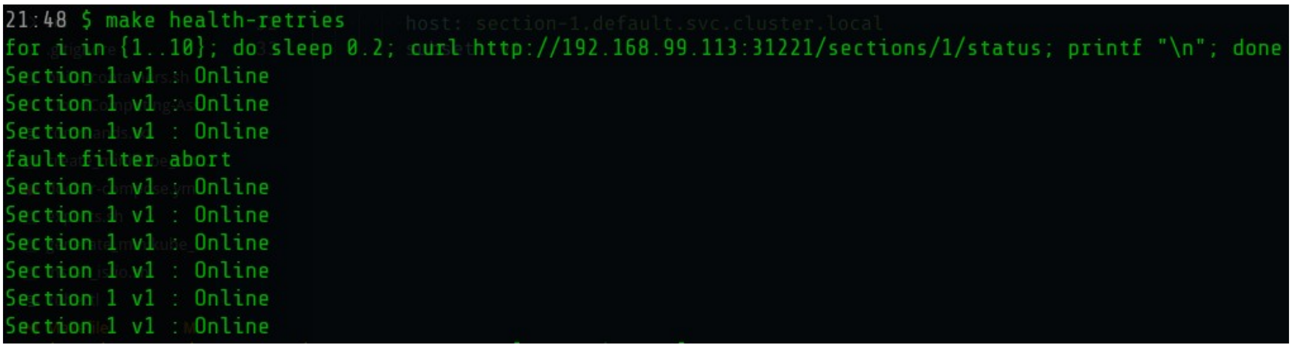


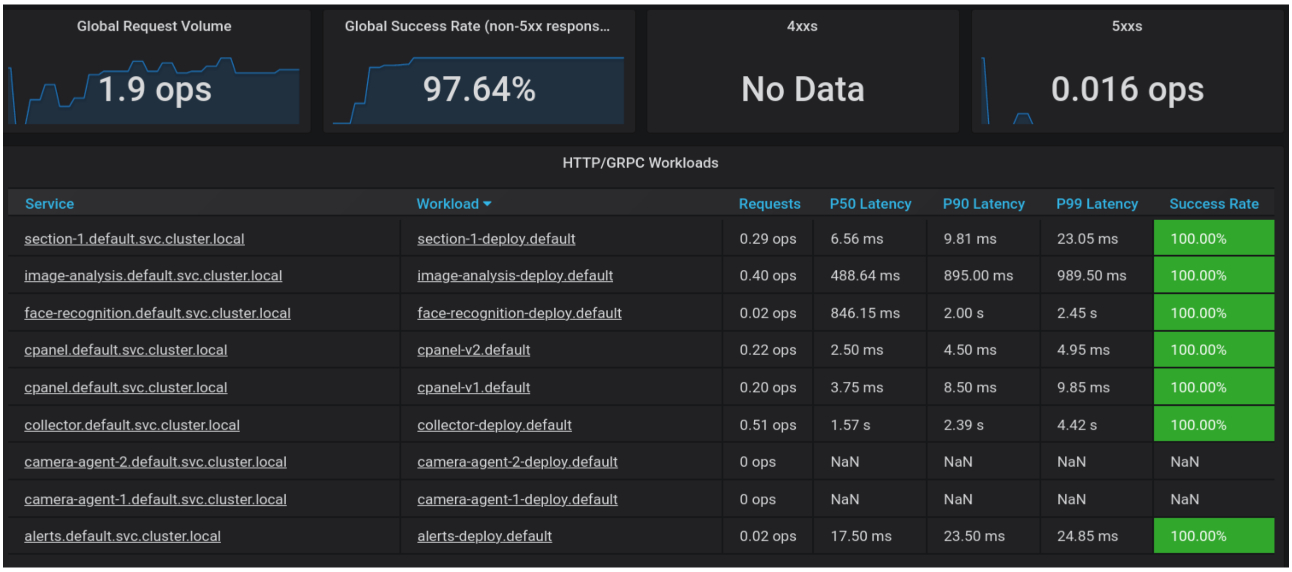


retries

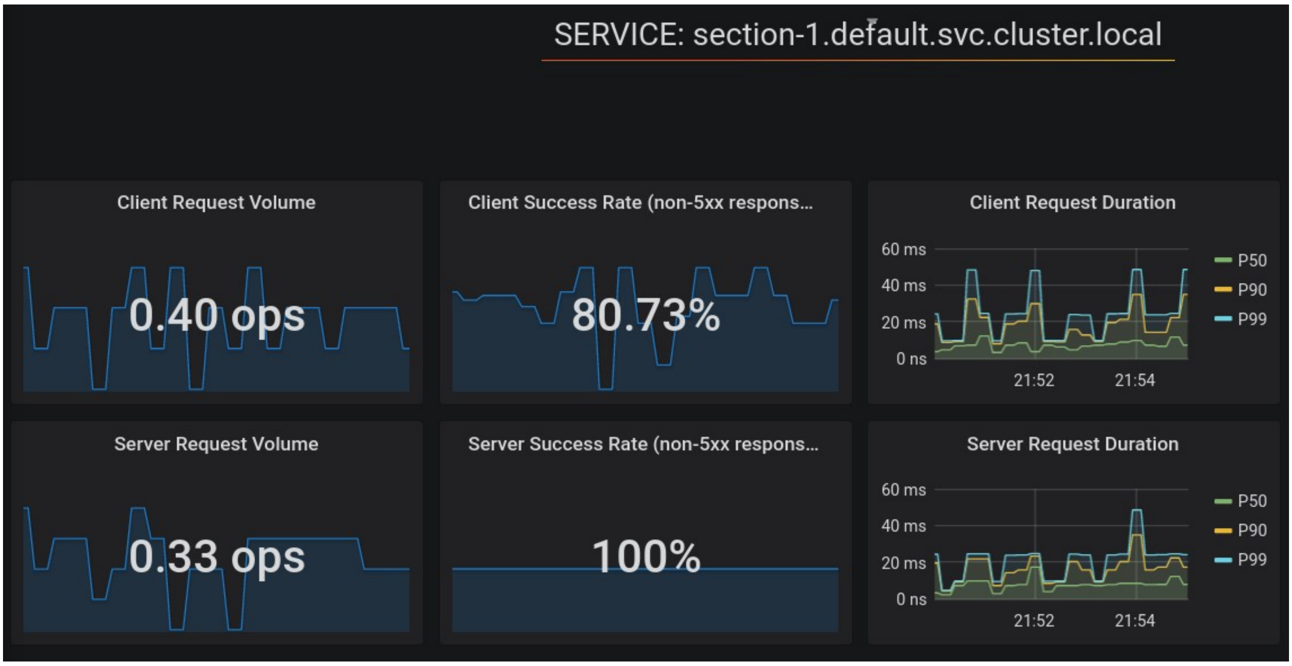
```
$ make retries
./kubectl apply -f istio/retry.yaml
virtualservice.networking.istio.io/collector configured
virtualservice.networking.istio.io/section-1 configured

$ 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
```

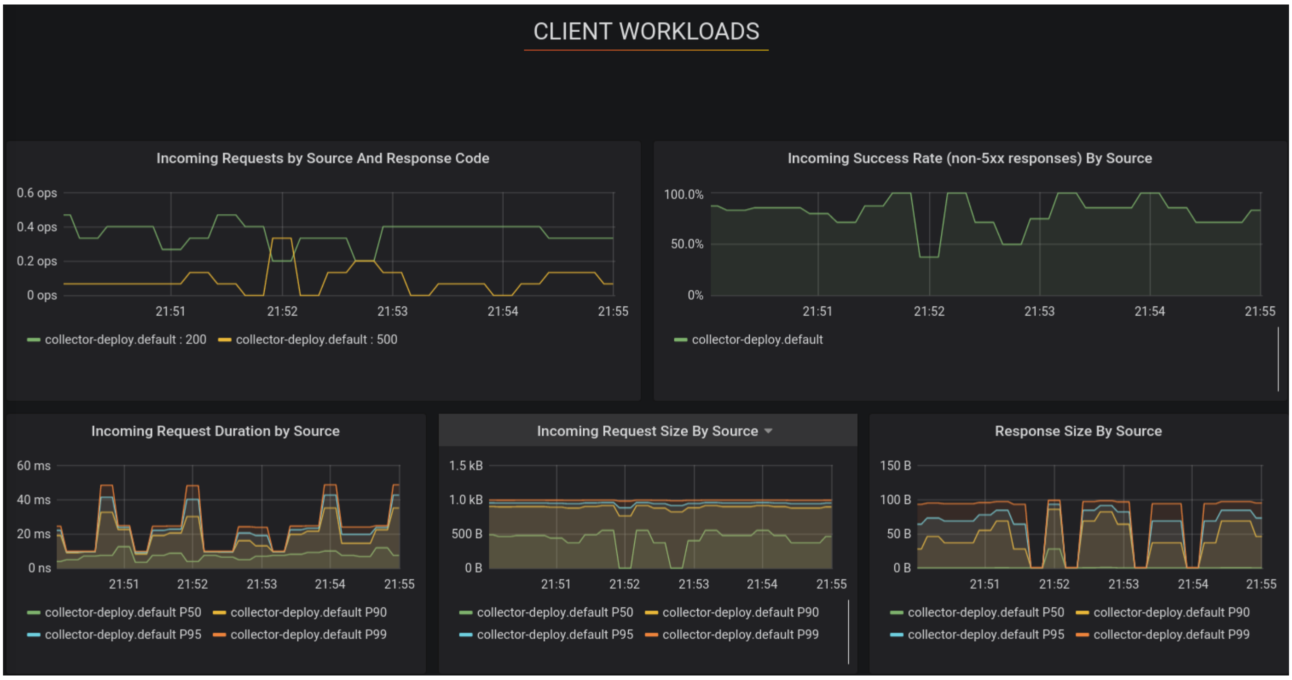
fault filter abort
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online

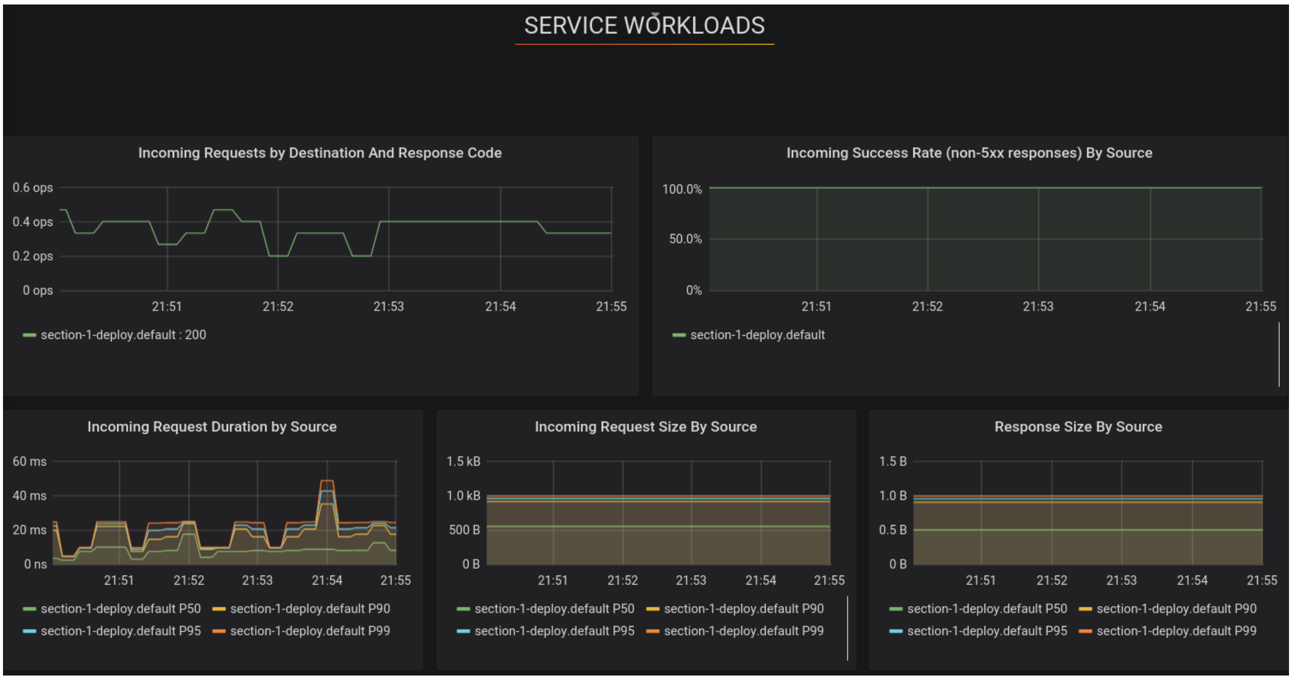




Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default	0.29 ops	6.56 ms	9.81 ms	23.05 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.40 ops	488.64 ms	895.00 ms	989.50 ms	100.00%
face-recognition.default.svc.cluster.local	face-recognition-deploy.default	0.02 ops	846.15 ms	2.00 s	2.45 s	100.00%
cpanel.default.svc.cluster.local	cpanel-v2.default	0.22 ops	2.50 ms	4.50 ms	4.95 ms	100.00%
cpanel.default.svc.cluster.local	cpanel-v1.default	0.20 ops	3.75 ms	8.50 ms	9.85 ms	100.00%
collector.default.svc.cluster.local	collector-deploy.default	0.51 ops	1.57 s	2.39 s	4.42 s	100.00%
camera-agent-2.default.svc.cluster.local	camera-agent-2-deploy.default	0 ops	NaN	NaN	NaN	NaN
camera-agent-1.default.svc.cluster.local	camera-agent-1-deploy.default	0 ops	NaN	NaN	NaN	NaN
alerts.default.svc.cluster.local	alerts-deploy.default	0.02 ops	17.50 ms	23.50 ms	24.85 ms	100.00%







Circuit breaker

Conclusions and Future Work

- 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 $$$

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