Abstract
cloud native, need for resiliency, service meshes as solution, demo, pros of istio – resiliency and fault injection

Motivation
docker, adoption of containers, 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
other service meshes/libraries, pros/cons, trend

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

Major idea
take uni cloud computing project and deploy it with istio, because I wanted to try out istio on something, that I coded myself and not “hello world” projects or some that are given in istio documentation (Bookinfo App).
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

describe **istio**, features, API installed as kubernetes CRD

- service mesh
  - security – who talks whom, trusted communication
  - observability – tracing / metrics
- 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:
        - + Client-side load balancing
        - Timeout – virt svc, default = 15 sec
        - Retry – virt svc, default = NO
        - circuit breaker – dest rule – LB least request
        - Pool ejection
        - + Health checks
        - Outlier Detection
- **demo**: git repo, minikube (specific versions), istio, shell scripts, makefile

Implementation
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
- **k8s**:
    - 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
- **how to play around:**

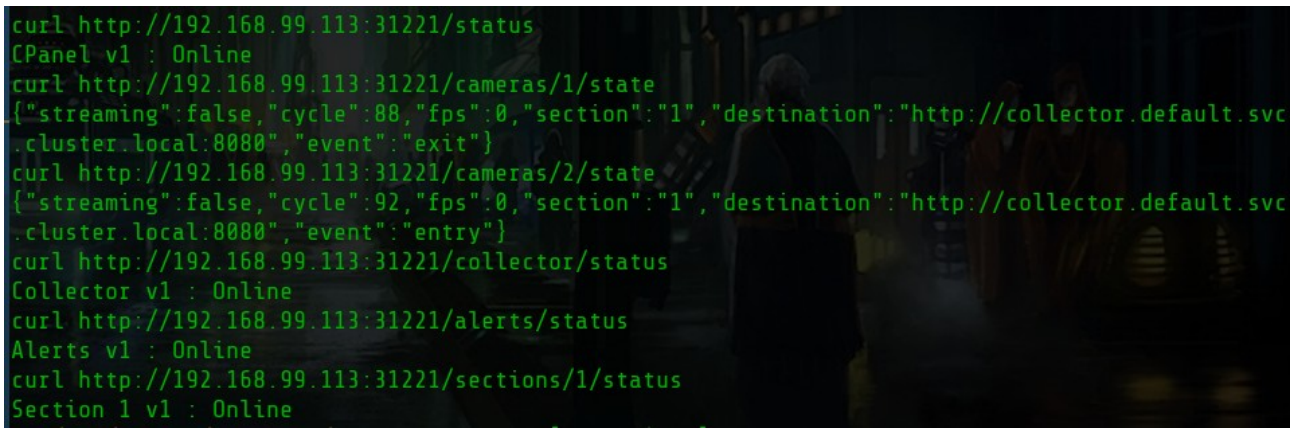
Evaluation and Discussion

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

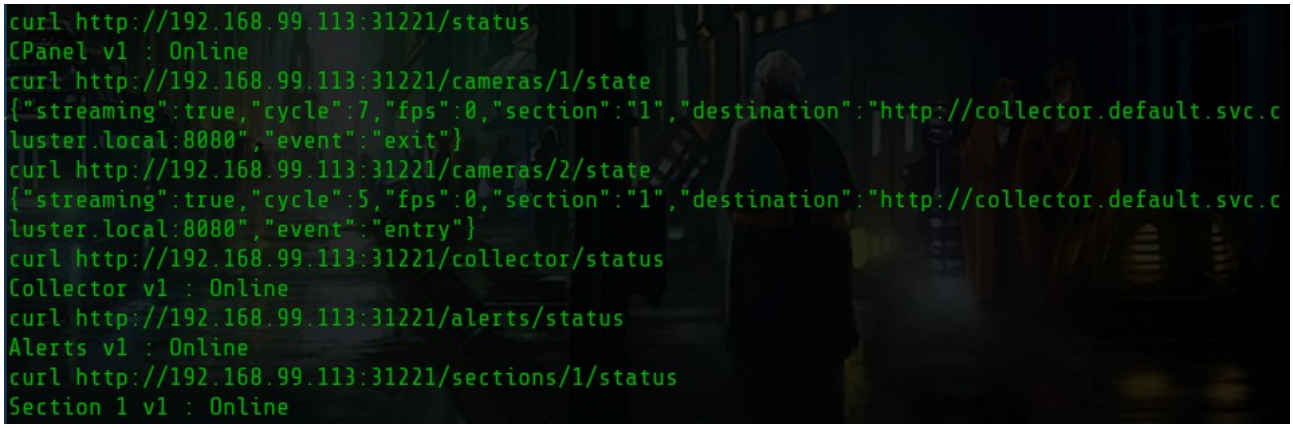
Kubernetes has only round robin load balancing. Istio with the help of destinations rules extends native kubernetes load balancing and presents the following types: random, round robin, weighted least request, ring hash (#istio). In such a case istio can give any microservice replica set it's own load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

Istio routing mechanism

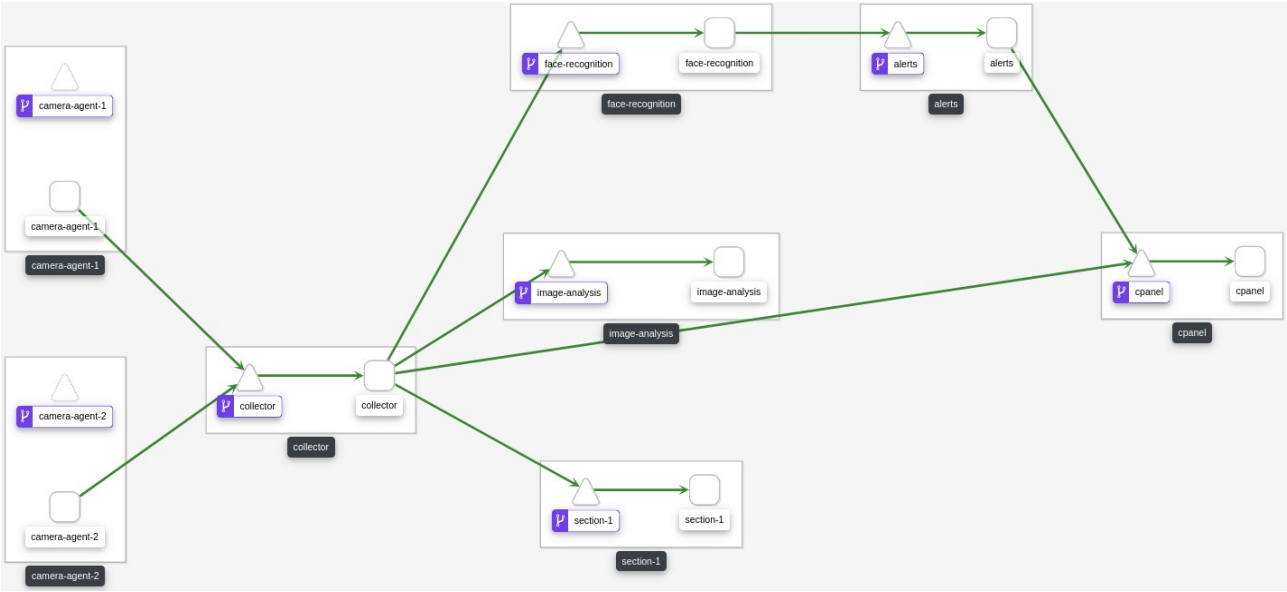
This solution can be used to make canary deployments and also make user experience more resilient - "user resilience". For example, new version of service can be made available only to one group of users (test group). It can be as much as only 1% of of the hole traffic. Users can be filtered by headers 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



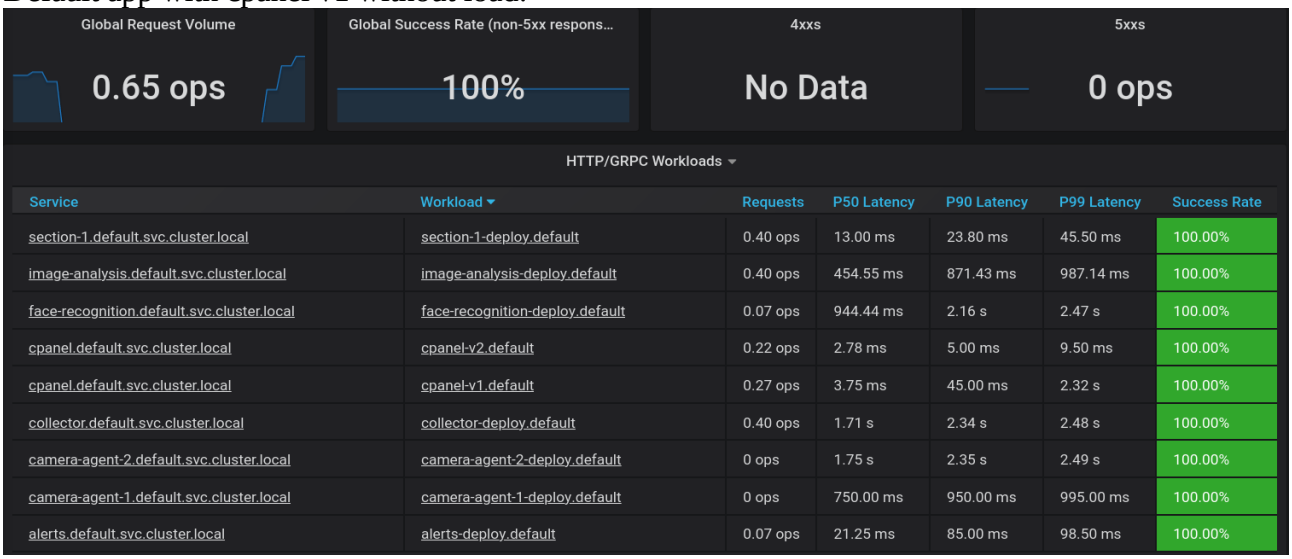
$ make start-cameras
$ make health



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.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



Default app with cpanel v1 without load:



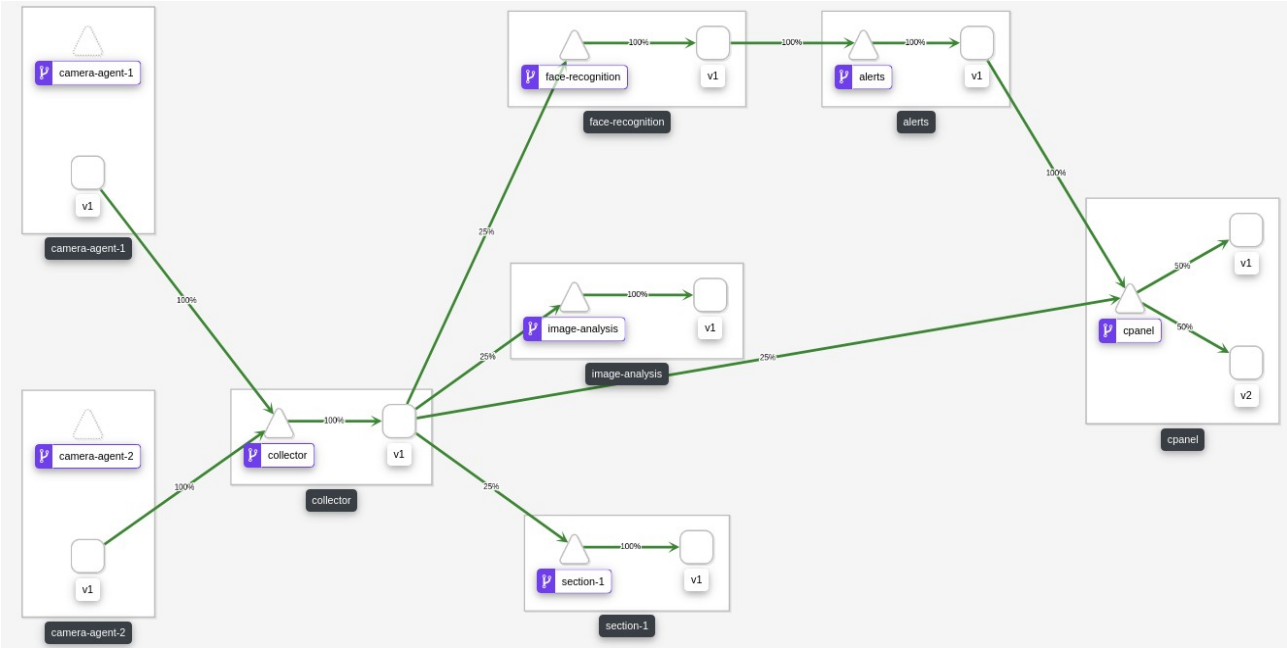
$ make load
for i in {1..100}; do sleep 0.2; curl http://192.168.99.113:31221/status; printf "\n"; done
CPanel v1 : Online
CPanel v1 : Online

CPanel v1 : Online

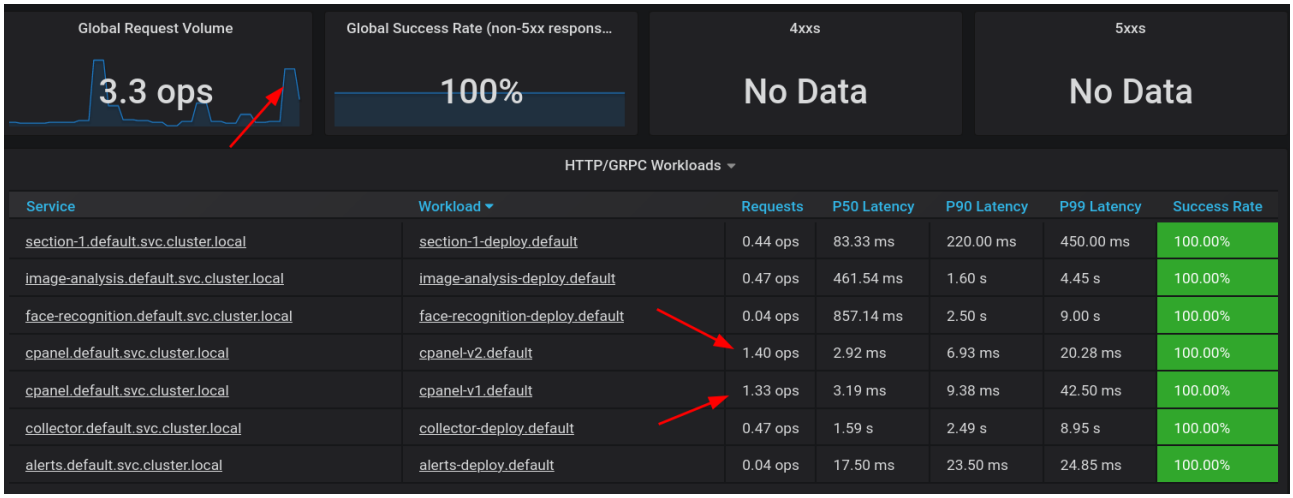
$ make cpanel-50-50
./kubectl apply -f istio/virt_svc_50-50.yaml
virtualservice.networking.istio.io/cpanel configured
check configuration
$ k get virtualservices cpanel -o yaml

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

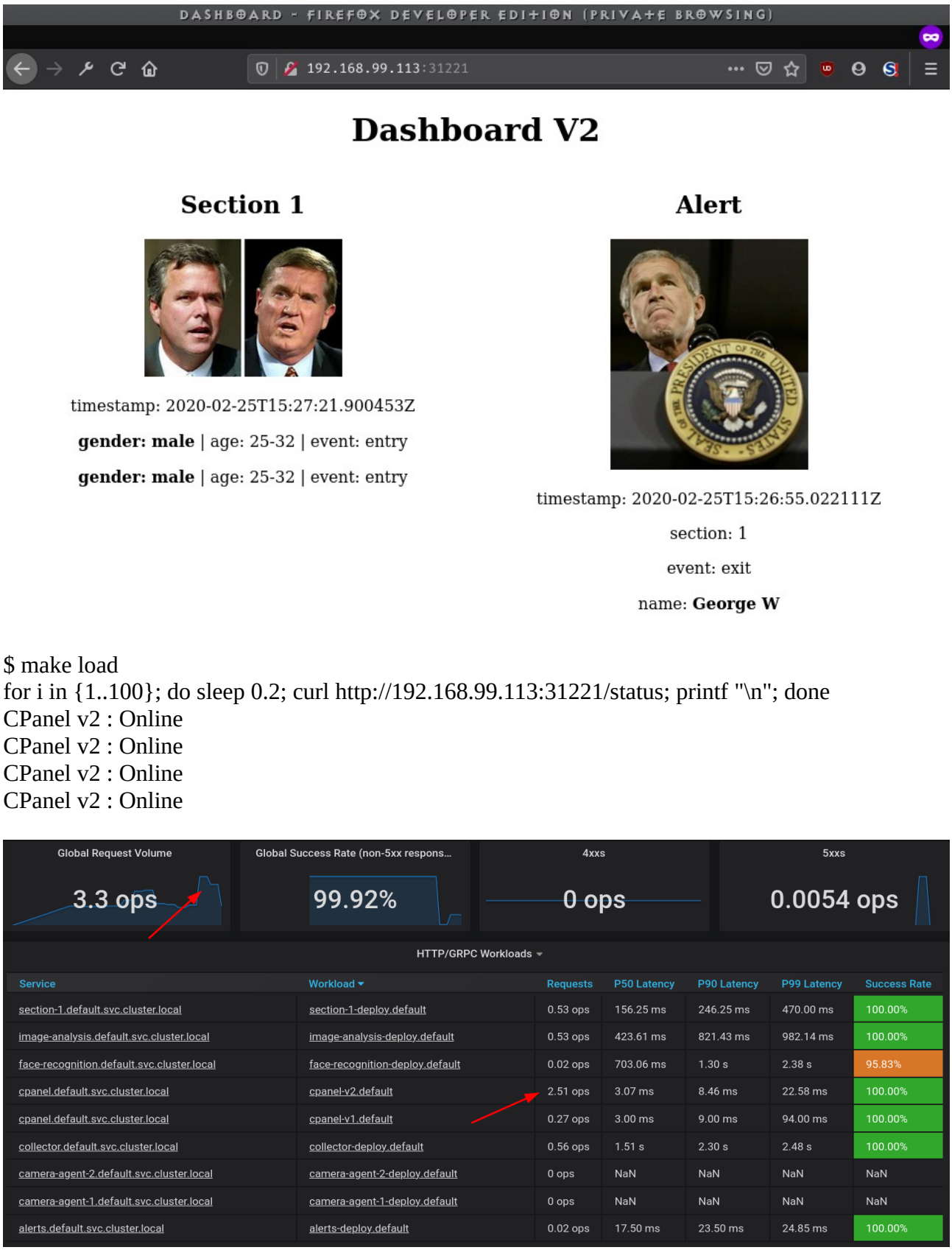


Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default	0.44 ops	83.33 ms	220.00 ms	450.00 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.47 ops	461.54 ms	1.60 s	4.45 s	100.00%
face-recognition.default.svc.cluster.local	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

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

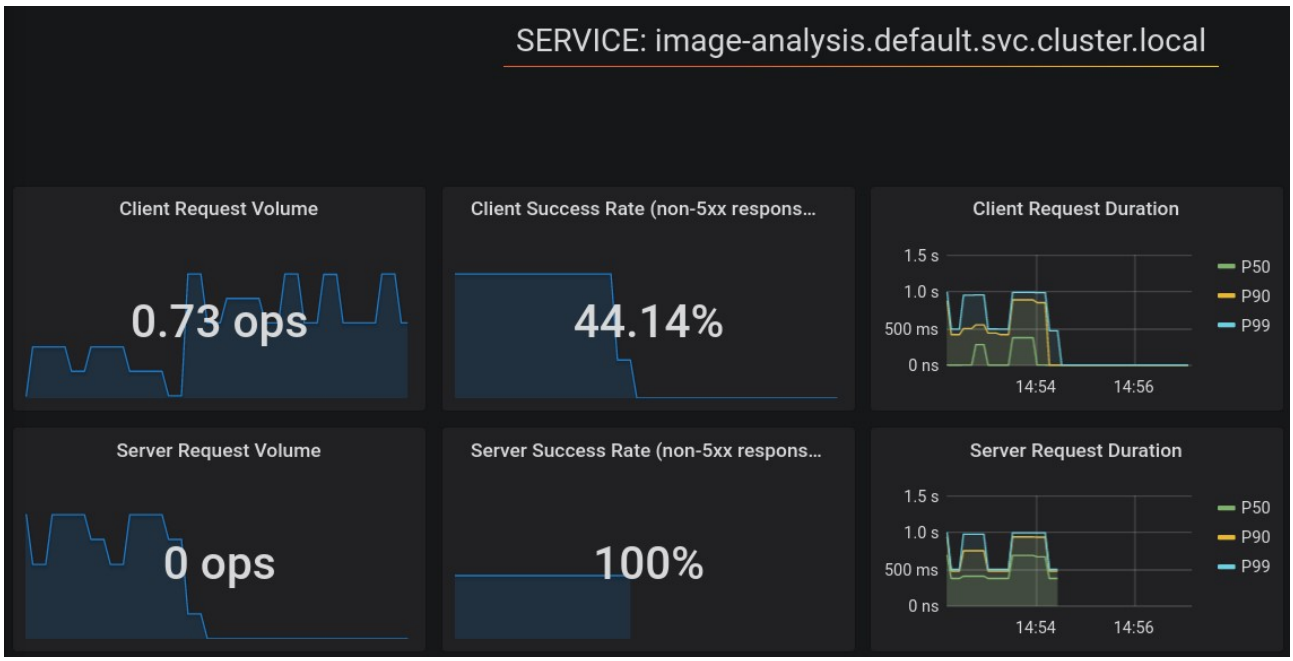
Internal istio mechanism for chaos testing. Allows simulating network and service errors without touching the source code of microservice at all. All faults are done by sidecar proxy.

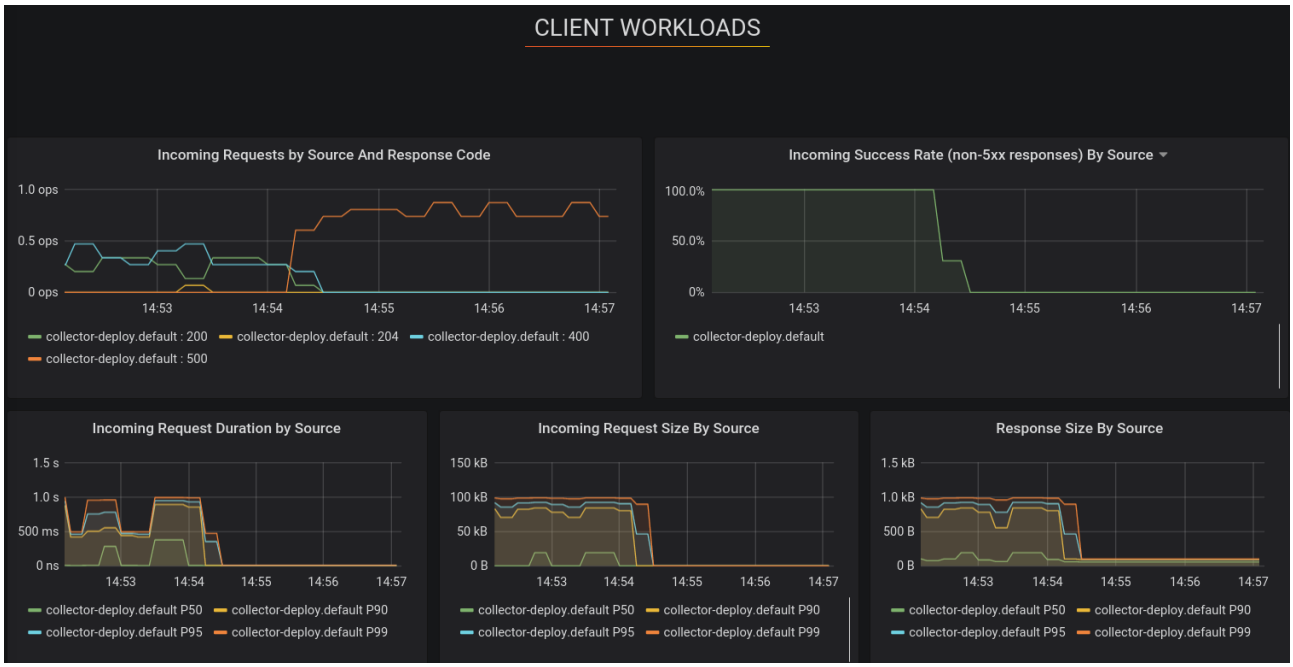
```
http:
- fault:
    abort:
      httpStatus: 503
      percentage:
        value: 50
route:
- destination:
    host: alerts.default.svc.cluster.local
    subset: v1

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
```

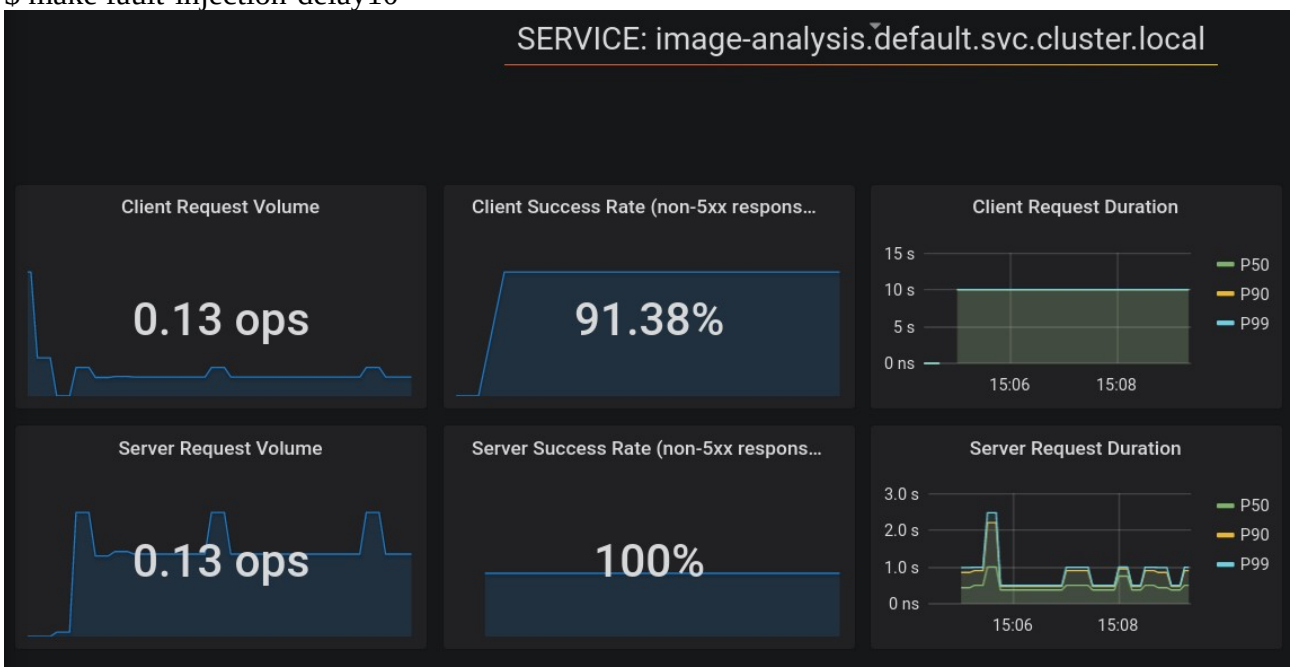
client workloads - workloads that are calling this service
service workloads - workloads that are providing this service

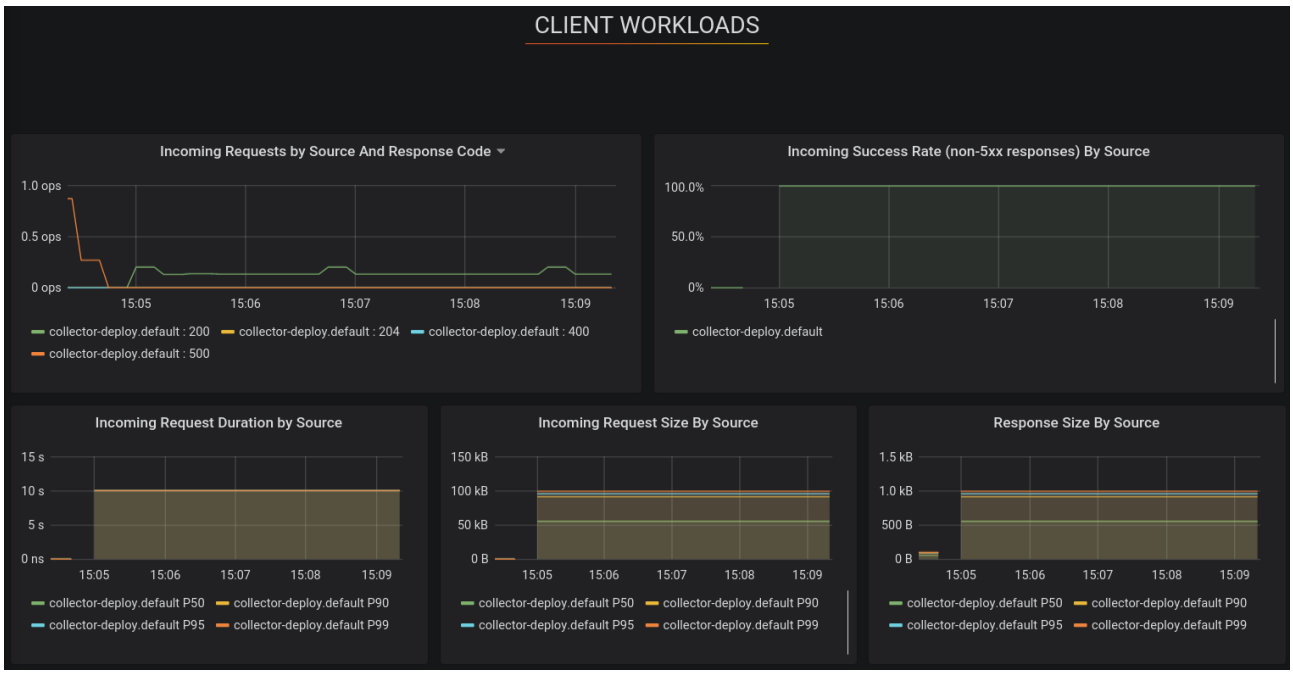
$ make fault-injection-500

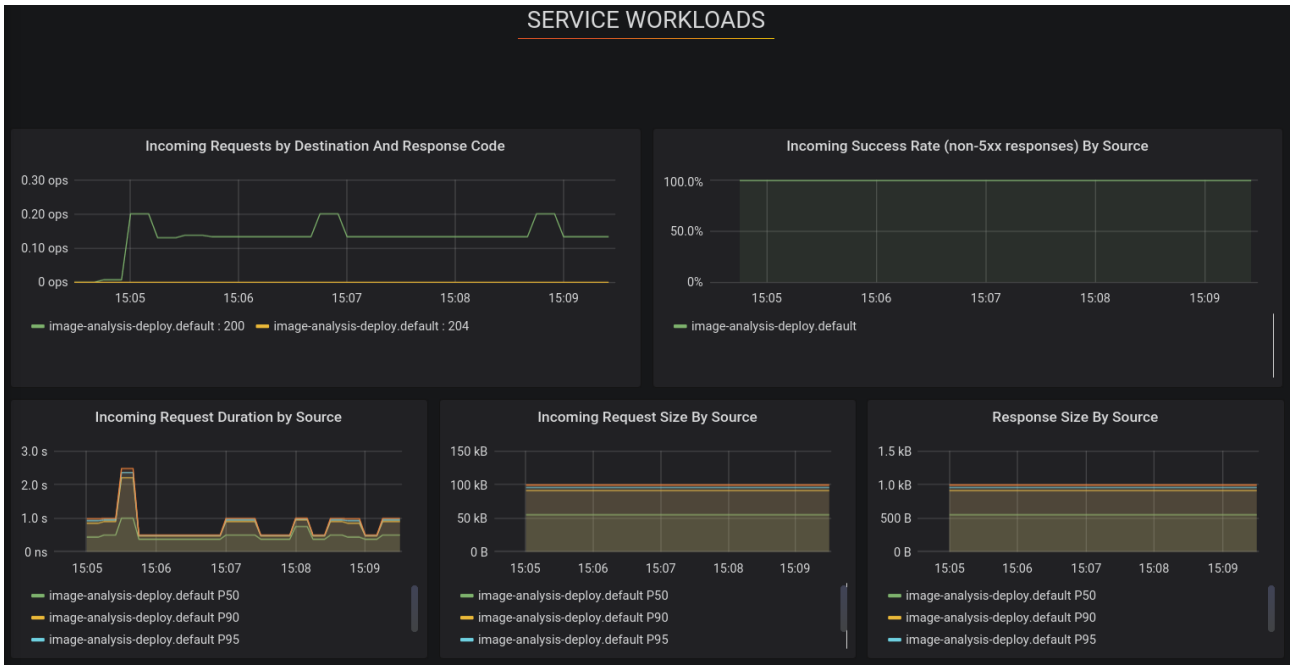

SERVICE: image-analysis.default.svc.cluster.local
Client Request Volume
0.73 ops
Client Success Rate (non-5xx respons...
44.14%
Client Request Duration
1.5 s
1.0 s
500 ms
0 ns
14:54
14:56
P50
P90
P99
Server Request Volume
0 ops
Server Success Rate (non-5xx respons...
100%
Server Request Duration
1.5 s
1.0 s
500 ms
0 ns
14:54
14:56
P50
P90
P99


CLIENT WORKLOADS
Incoming Requests by Source And Response Code
1.0 ops
0.5 ops
0 ops
14:53
14:54
14:55
14:56
14:57
collector-deploy.default : 200
collector-deploy.default : 204
collector-deploy.default : 400
collector-deploy.default : 500
Incoming Success Rate (non-5xx responses) By Source
100.0%
50.0%
0%
14:53
14:54
14:55
14:56
14:57
collector-deploy.default
Incoming Request Duration by Source
1.5 s
1.0 s
500 ms
0 ns
14:53
14:54
14:55
14:56
14:57
collector-deploy.default P50
collector-deploy.default P90
collector-deploy.default P95
collector-deploy.default P99
Incoming Request Size By Source
150 kB
100 kB
50 kB
0 B
14:53
14:54
14:55
14:56
14:57
collector-deploy.default P50
collector-deploy.default P90
collector-deploy.default P95
collector-deploy.default P99
Response Size By Source
1.5 kB
1.0 kB
500 B
0 B
14:53
14:54
14:55
14:56
14:57
collector-deploy.default P50
collector-deploy.default P90
collector-deploy.default P95
collector-deploy.default P99

$ make fault-injection-delay10







timeout

$ make timeout

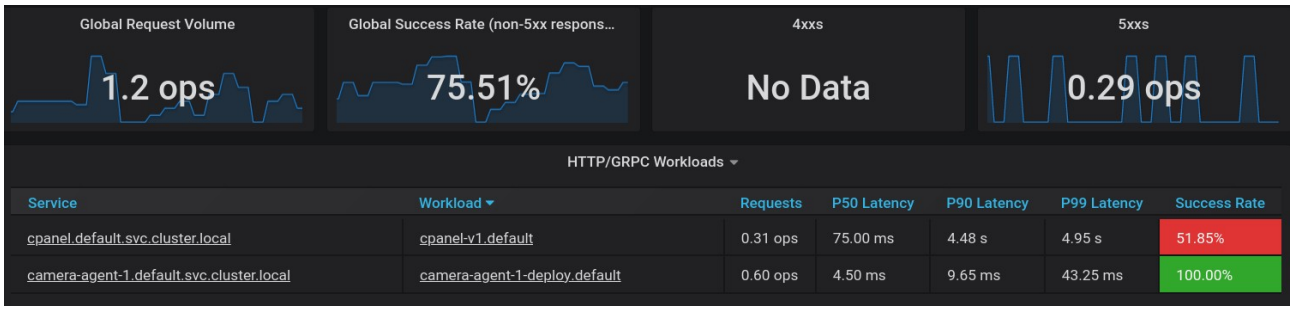
./kubectl apply -f istio/timeout.yaml

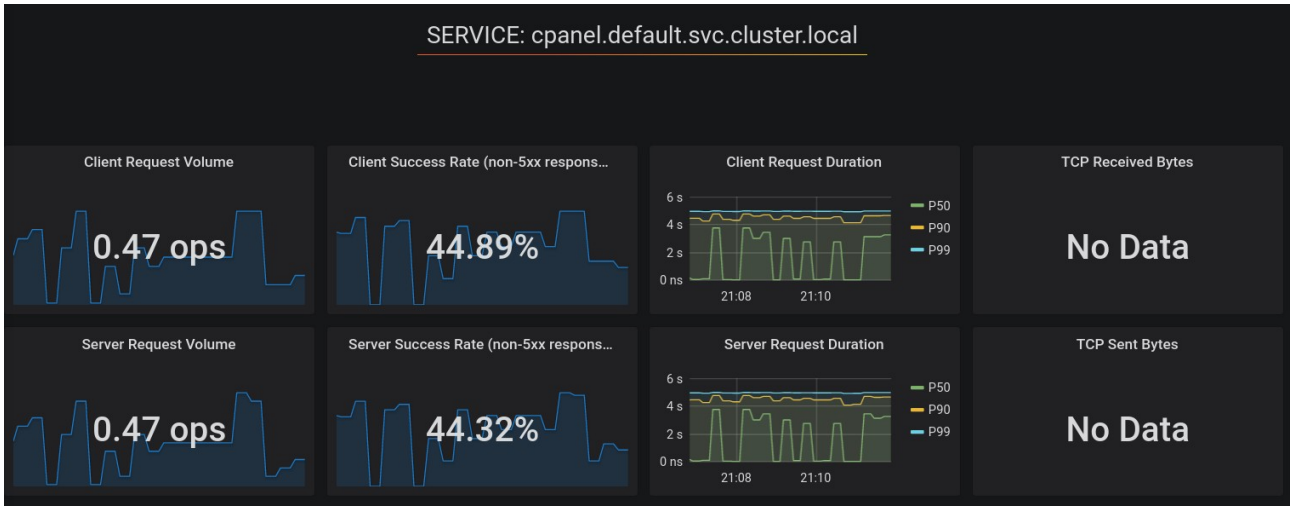
virtualservice.networking.istio.io/camera-agent-1 configured

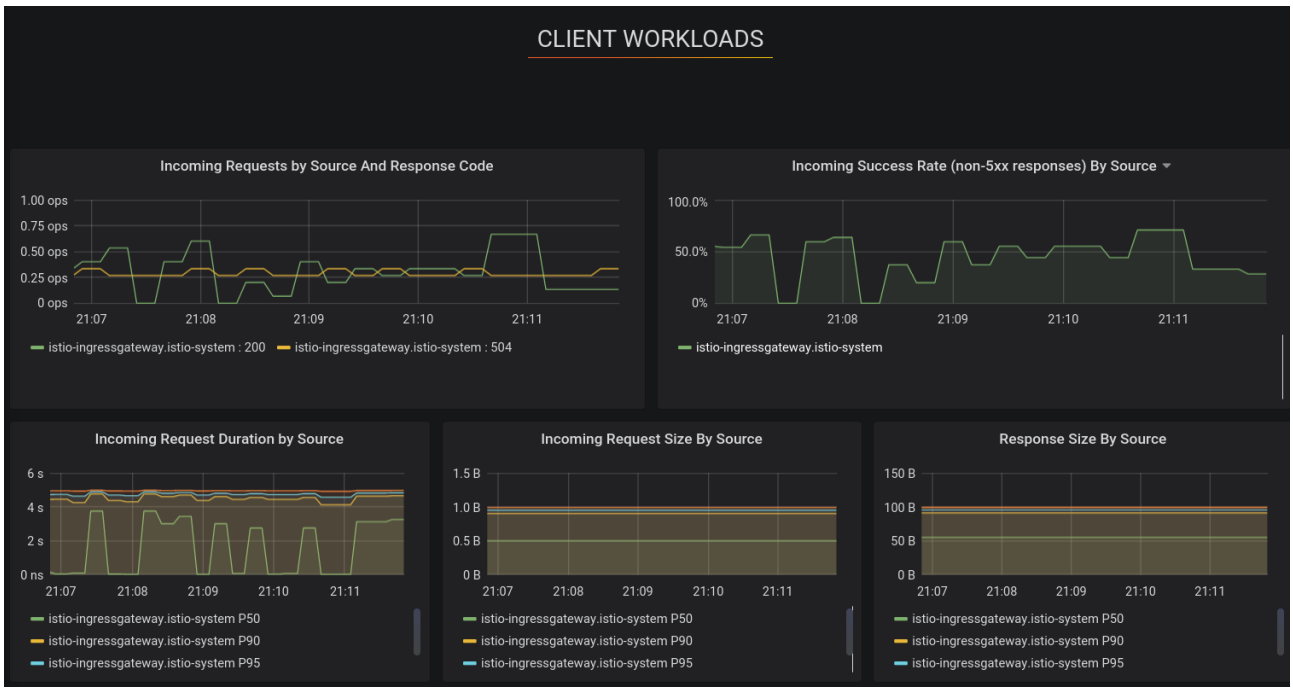
virtualservice.networking.istio.io/cpanel configured

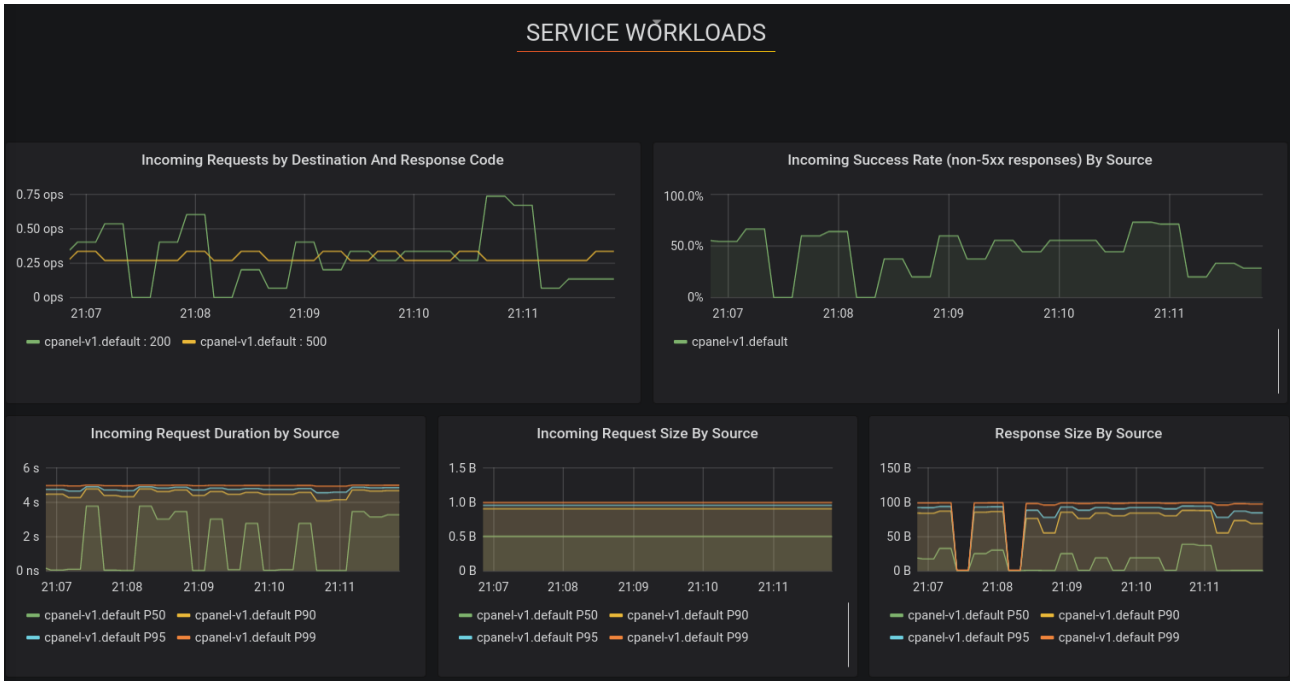
$ make health-timeout
for i in {1..10}; do sleep 0.2; curl http://192.168.99.113:31221/cameras/1/state; printf "\n"; done
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"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

Conclusions and Future Work

- pros of istio resiliency features
- expanse of service meshes
- complexity of operations (# of micro services, agile)
- advice - move to production step by step incremental, complexity of debugging
  - configure log level to error – otherwise too much traffic $$$

References

1. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
2. https://www.martinfowler.com/articles/microservices.html
3. https://snyk.io/blog/10-docker-image-security-best-practices/
4. Cloud computing assignment
5. https://12factor.net/
6. https://kubernetes.io/
7. https://istio.io/
8. https://www.docker.com/

Appendix / Supplemental Material

- cc assignment
- commands