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

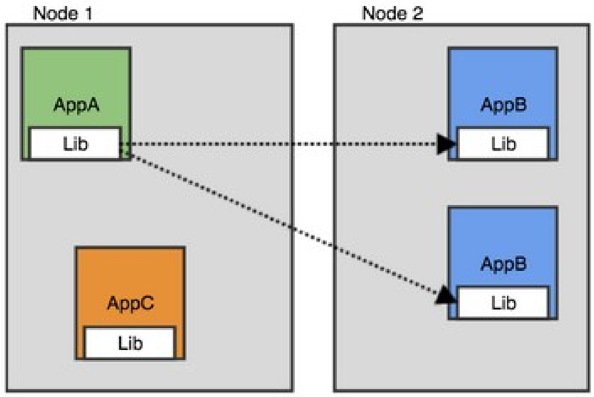
Motivation

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

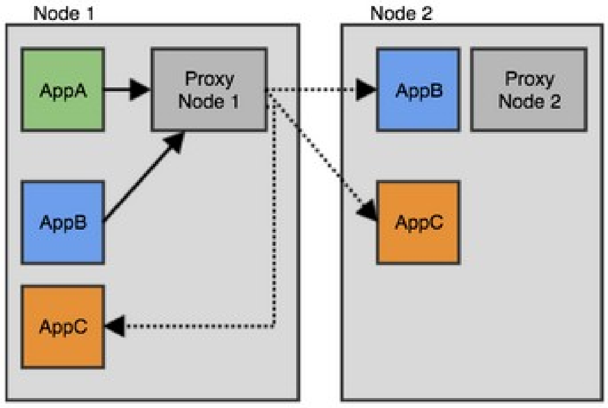
Related work

Need for service meshes
compare API gateway and service mesh
other service meshes/libraries, pros/cons, trend, pictures [alt]

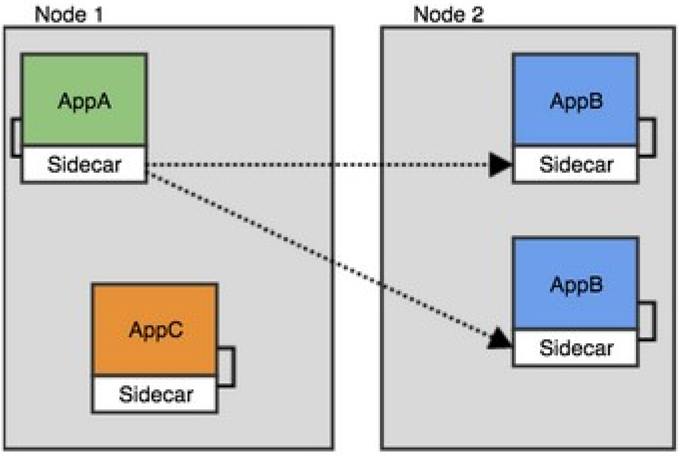
- libs: cons - code change (hystrix, ribbon)



- node agent (linkerd)



- sidecar (istio, linkerd2, consul)



Major idea

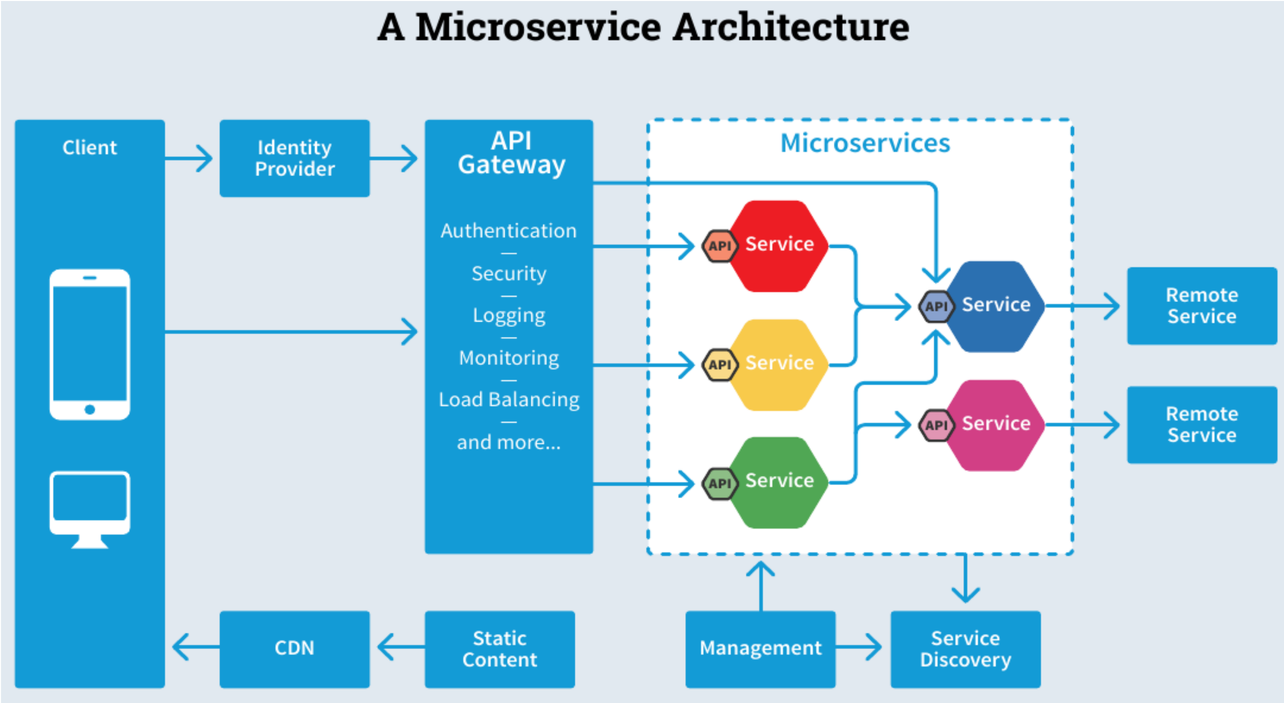
There are plenty of tutorials online that utilize a sample application from istio web site (“Bookinfo” application) to show typical service mesh and specific istio features. The idea of this thesis is to take the already implemented project, adopt it a little bit and provide a working demo of istio resiliency features. The project itself is a part of cloud computing course. The text of the original assignment can be found in supplemental material.
Trying to make focus on operational part of software engineering and not to focus on developing from scratch other possibility was to take a ready open source project from Github and deploy it with istio. After researching and looking into some of such projects the decision to take the application developed by myself in cloud computing course was made.

Microservices

Motivation

Virtualization was not meant to speed application development.

Microservices are small, independently scaled and managed services. Each service has its own unique and well-defined role, runs in its own process and communicates via HTTP APIs or messaging [native].



[native]

services with task in mind, no shared libraries and dependencies, separation of stateless and stateful services,

pros

- granularity - small
- rapid development
- polyglot – language, development teams, domains
- scaling
- suitable for infrastructure as code – CI/CD, canary
- separate data storages
- easier code maintenance

cons

- increased operational complexity – deployment and monitoring [towards]
- communication security
- communication issues
- resiliency issues – cascade failures in distributed systems
- complicated transition from monolith to microservice [towards]
- service discovery

Service mesh

Motivation

out of the box plenty of features that are now implemented in different ways: libraries for logging, API gateways for routing, certificates rotation for secure communication.

monitoring of metrics, tracing of requests, network communication logging, resiliency, security, routing canary, mirroring, green/blue

architecture

service mesh focuses on networking between microservices rather than business logic

data plane and control plane

types

sidecar proxies

pros/cons

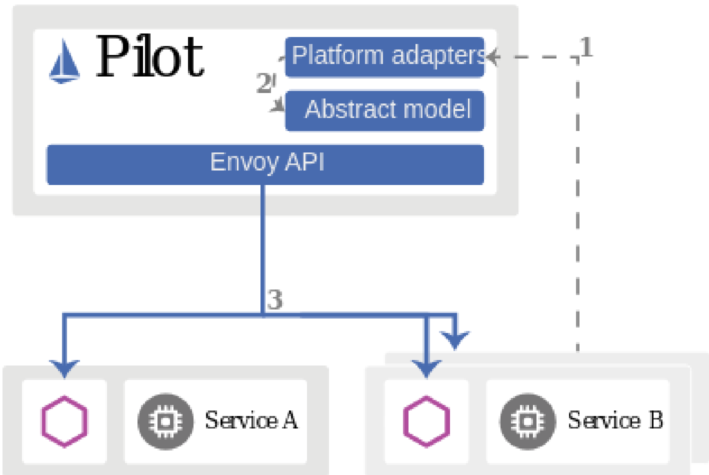
Istio

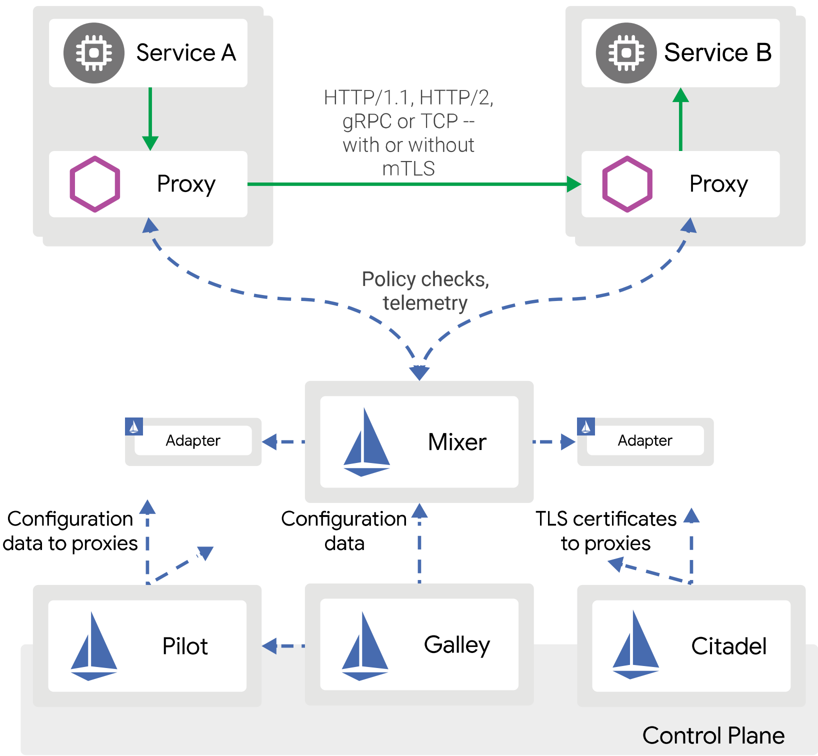
features, API installed as kubernetes CRD

- 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

sidecar proxy - envoy

Observability
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

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 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,

pros:

- all in one solution
- language independent

cons:

- high complexity
- higher latency

- resource hungry – x2 containers
- young technology

Resiliency

In distributed microservices architecture one service can not await that all other services function without errors or that there are no network failures at all. Taking in consideration these aspects resiliency can be defined as the ability of distributed system continue to respond to client though there are network and service errors.

Here you can find resiliency features of istio service mesh.

Health checks
There are two types: liveness and readiness probes [k8s]. They are crucial for system resiliency because the traffic should be forwarded only to healthy pods. Liveness probes help to determine if application started and run correctly. Readiness probes check if application is ready to receive traffic for example after all configurations finished successful [action].
Though these are mechanisms belong are kubernetes native thay are still worth to mention because istio proxies allow these health checks to work seamlessly. Only Http health checks work only with mTLS enabled so need some configuration on the side istio system namespace.
Exec and tcp health checks work straight forward without any changes in kubernetes manifests.

Load balancing – more sophisticated then native kubernetes solution (round robin). Can be configured in destination rules.

- round robin (default)
- Random - random pods are taken for requests from load balancing pool
- Least requests - least overloaded pod get new requests

```
loadBalancer:
      simple: RANDOM
```

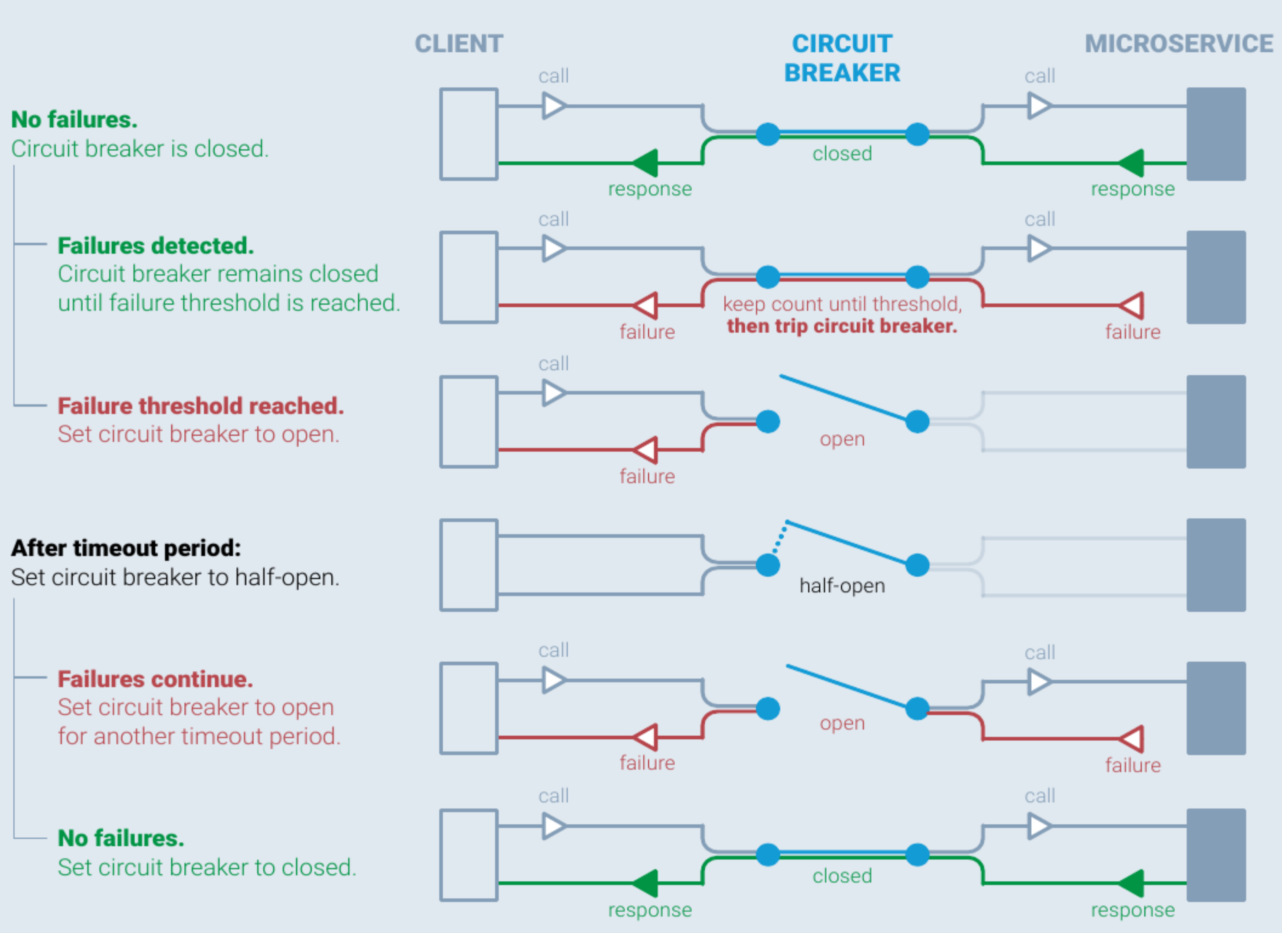
Timeout – virt svc, default = 15 sec.
Helps to deliver fast responses to client without waiting for response from slow service. For user experience it is better to fail fast then to function with delays. Define a proper timeout for calls depends on application and microservice. Too small – not enough time to process request from client, too big – may lead to general slow system responses. Alone waiting for slow responses need much infrastructure resources (CPU, RAM). That is why timeouts are very important and it is very easy to configure them for service with Istio. The main challenge here will be to proper define the length of timeout. So infrastructure engineer need to understand how the microservices application work or need to communicate with developers direct.

```
        percent: 100
        fixedDelay: 2s
```

Retry – virt svc, default = NO.
Retries repeat the failed request in order to get the response faster then return error to client and initialize a completely request. Normally developers take care of it in application code, but istio has built-in retry policies to configure and to make calls more resilient. Of course with repeated retries the load on service will be higher. This should be taken in consideration and could also be protected with circuit breaker for example.

```
     attempts: 3
      perTryTimeout: 2s
```

circuit breaker is configured in istio destination rule manifests.



[native]
General explanation - ...
We can see to types of this pattern in istio.
The first one functions at the connection pool level and protects microservice from overloading. It stops sending traffic to service if requests reach some limit defined in destination rule for this microservice.

```
http:
  http1MaxPendingRequests: 1
  maxRequestsPerConnection: 1
tcp:
  maxConnections: 1
```

The second type is outlier detection. If there are many replicas of microservice one of them can start returning errors (eg 50x). In this case istio will eject the problem pod from the load balancing pool for some time.
Following settings can be configured:

```
consecutiveErrors: 7
interval: 5m
baseEjectionTime: 15m
maxEjectionPercent: 100
```

Demo

The main result of this thesis will be a fully working demo to show the main resilience possibilities of Istio service mesh. The focus is made on all-in-one solution. Project written in cloud computing course is used as a microservice application. Git repository will all necessary scripts is provided to easy start using istio in devepment environment.

With the help of this demo you will learn basics of distributed applications and microservices, the concepts of modern application packaging, deployment and orchestration. Docker files and kubernetes manifests contain best practices from production deployments.

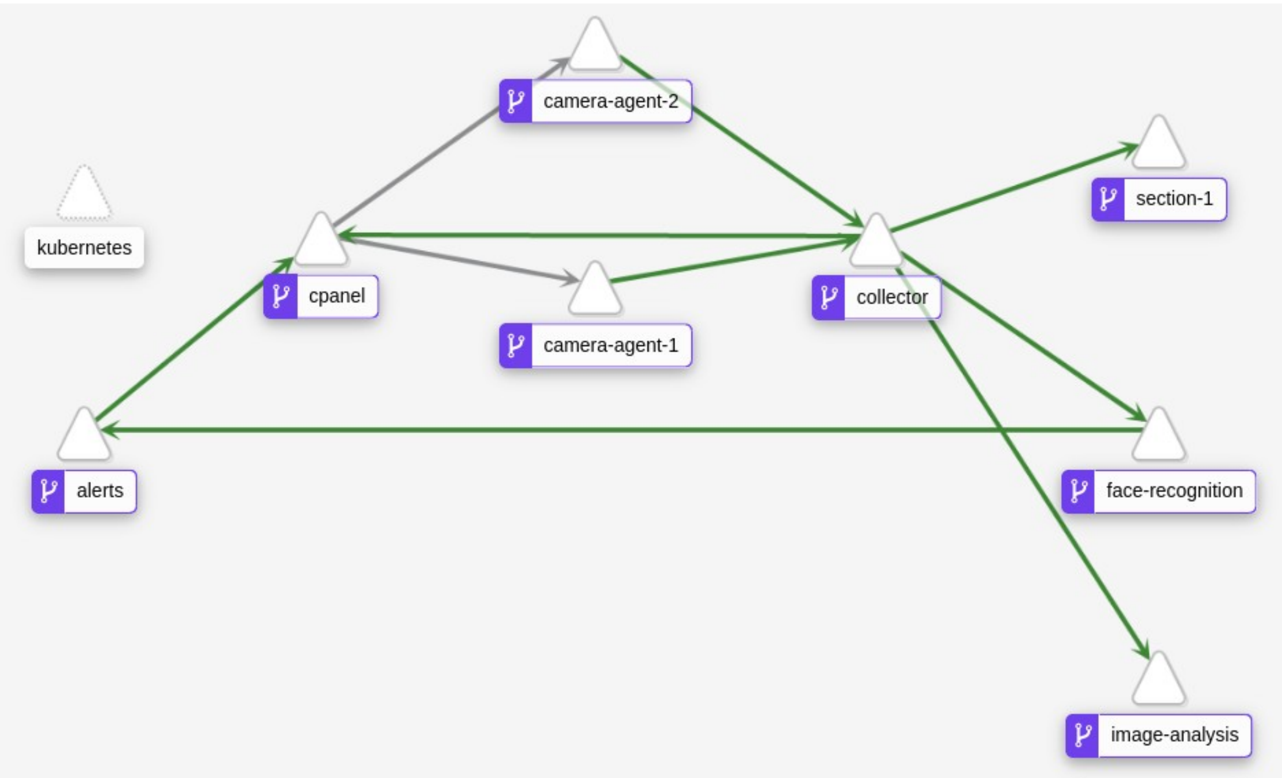
Implementation

The Twelve Factors App

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:
  - alpine, root, 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

git, virtualbox, curl, docker, shell scripts, yaml, minikube with kubectl, istio, Makefile, resiliency try out
install requirements (ram, cpu)
dirty tricks during installation and configuration test environment:

- sharing containers host/guest minikube
- telepresence for debugging and fast response to changes

Evaluation

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.

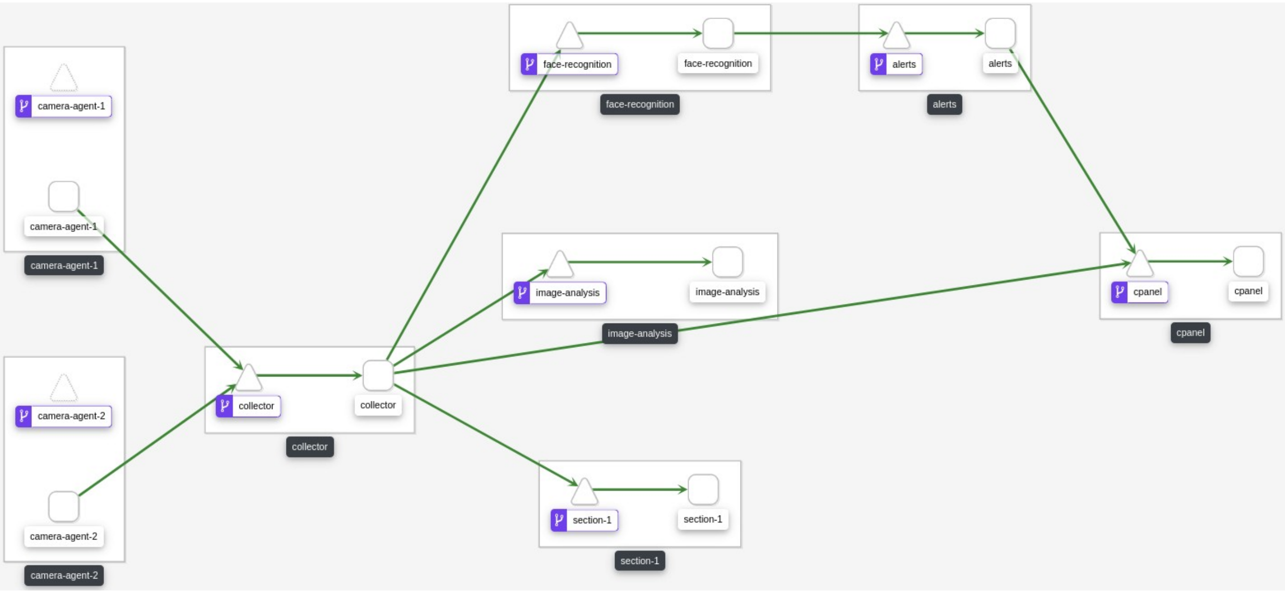
$ make deploy-app-default
$ make deploy-istio-default
$ make health

```
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

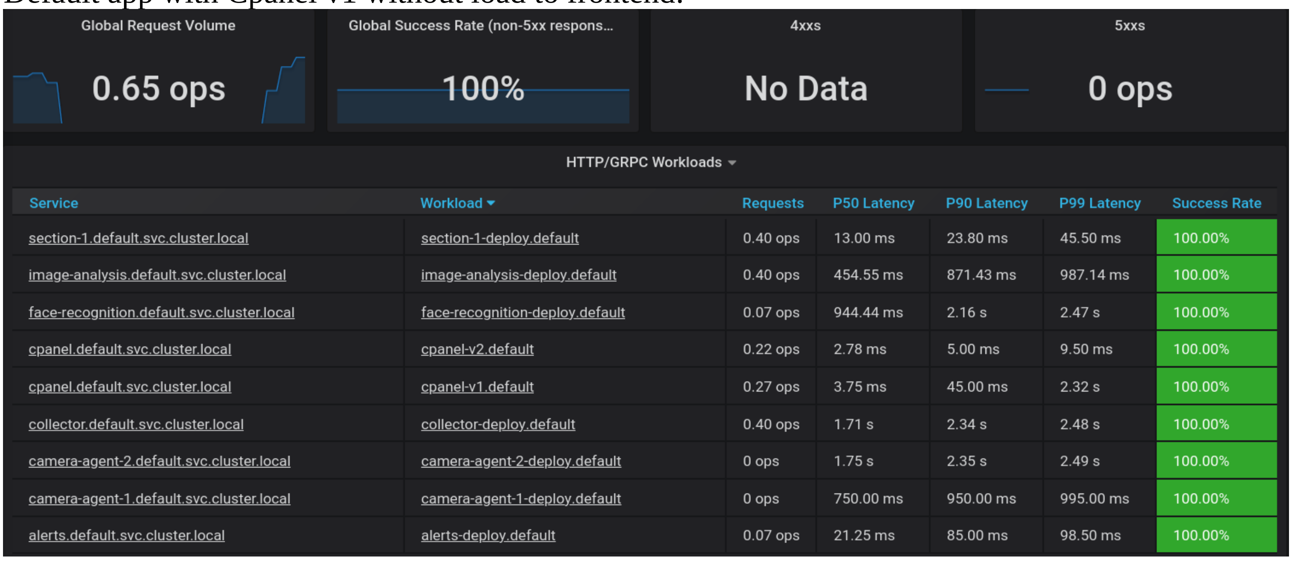
```
curl http://192.168.99.113:31221/status
CPanel v1 : Online
curl http://192.168.99.113:31221/cameras/1/state
{"streaming":true,"cycle":7,"fps":0,"section":"1","destination":"http://collector.default.svc.c
luster.local:8080","event":"exit"}
curl http://192.168.99.113:31221/cameras/2/state
{"streaming":true,"cycle":5,"fps":0,"section":"1","destination":"http://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



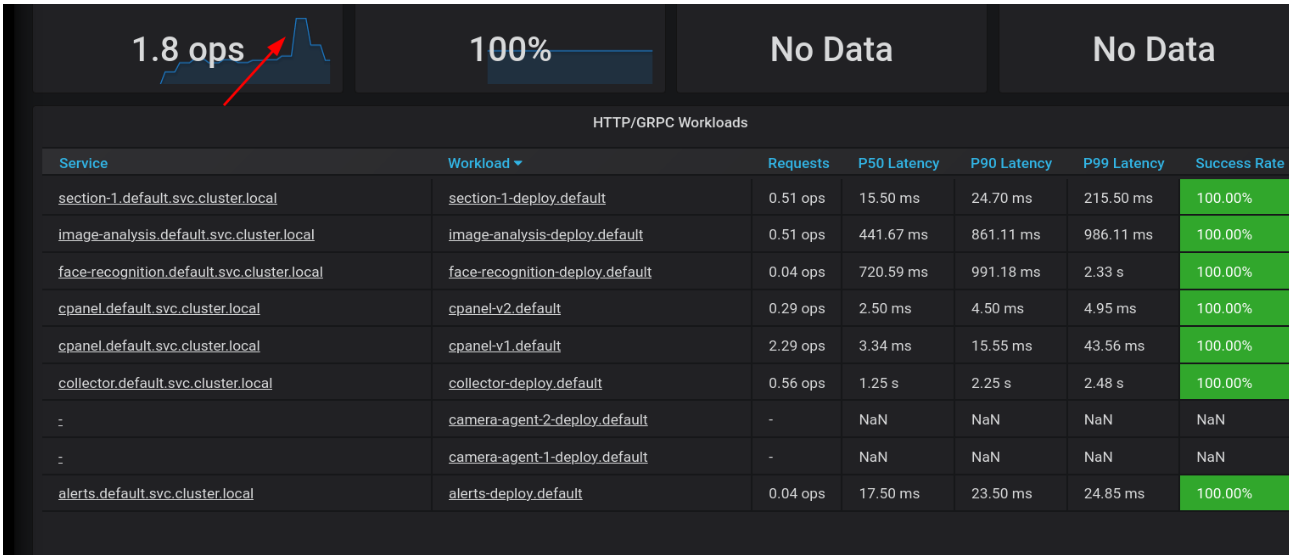
Version 1 of Cpanel microservice displays information about latest statistic from image analysis and the most recent alert. Both are displayed without showing the photo from camera agent itself. Displaying the photo is made in Version 2 of Cpanel microservice.

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
…

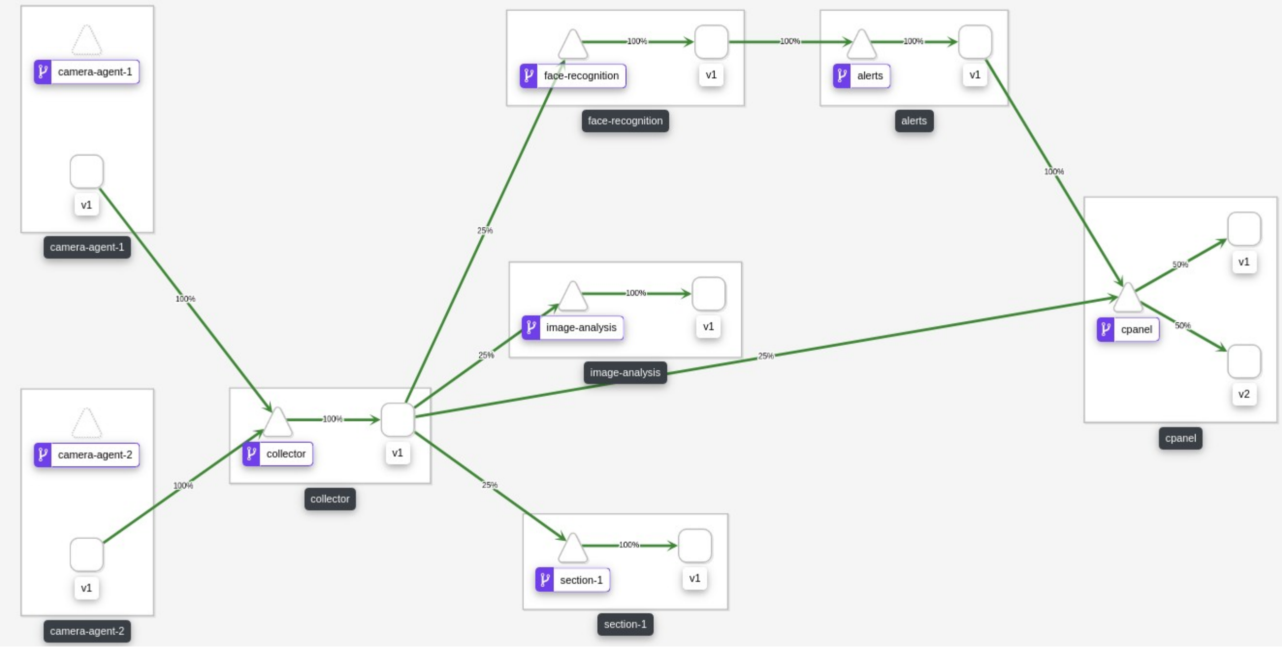


Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default	0.51 ops	15.50 ms	24.70 ms	215.50 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.51 ops	441.67 ms	861.11 ms	986.11 ms	100.00%
face-recognition.default.svc.cluster.local	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%

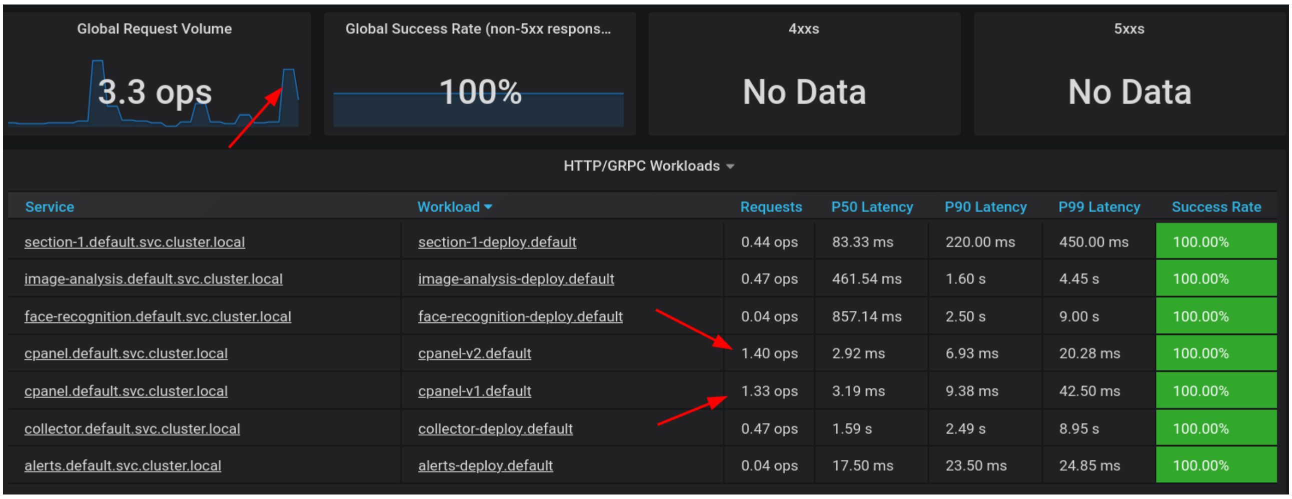
$ make cpanel-50-50
./kubectl apply -f istio/virt_svc_50-50.yaml
virtualservice.networking.istio.io/cpanel configured
check configuration
$ k get virtualservices cpanel -o yaml

```
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



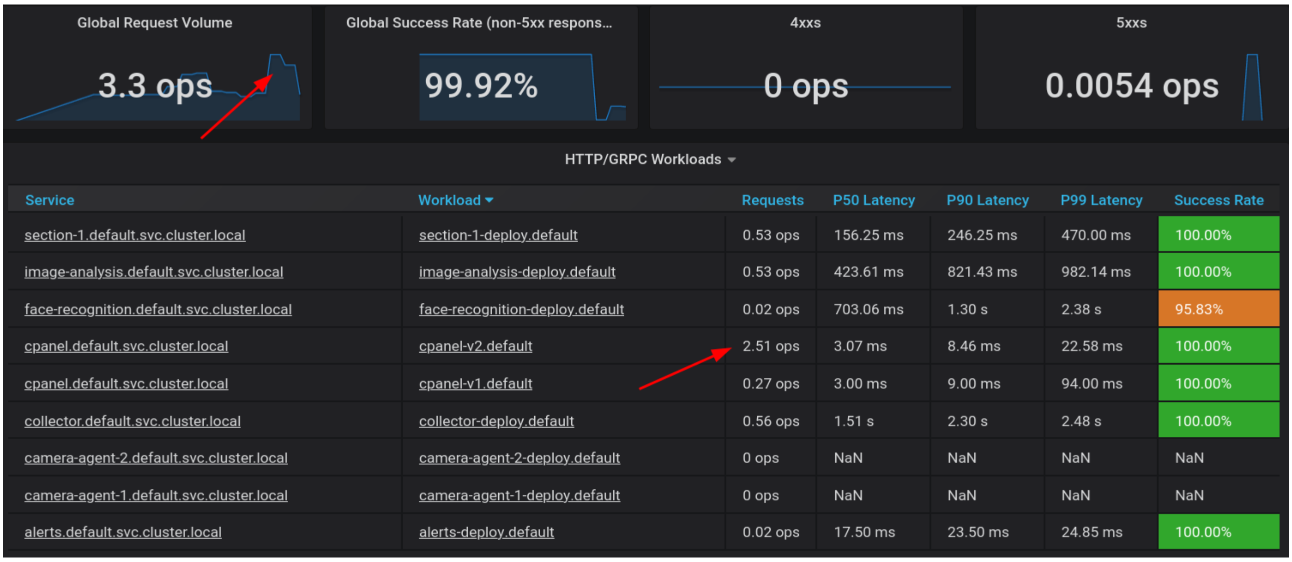
$ make cpanel-v2
./kubectl apply -f istio/virt_svc_v2.yaml
virtualservice.networking.istio.io/cpanel configured
check configuration
$ k get virtualservices cpanel -o yaml

```
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
```

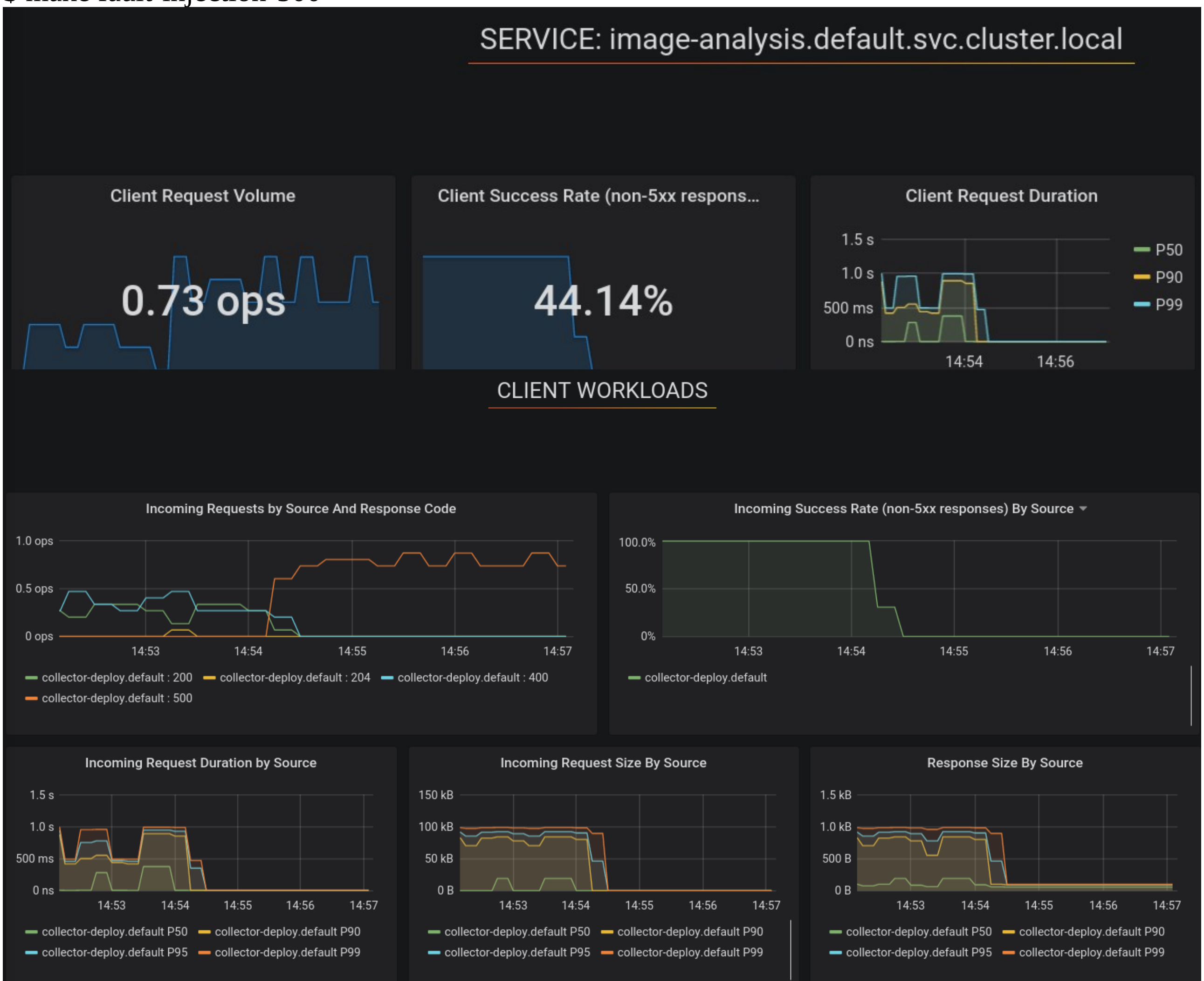
```
$ make random
./kubectl apply -f istio/random_lb.yaml
destinationrule.networking.istio.io/cpanel configured
$ k get destinationrules cpanel -o yaml

$ 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
```

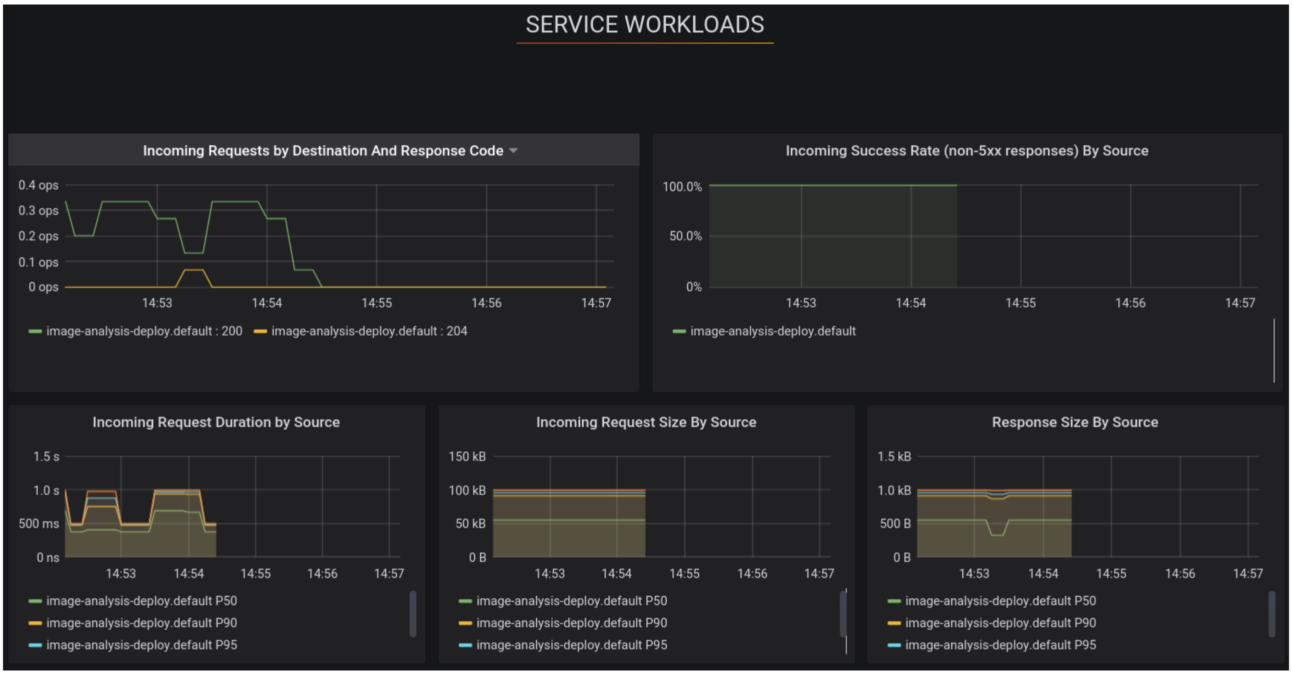
Fault injection

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 Envoy proxy.

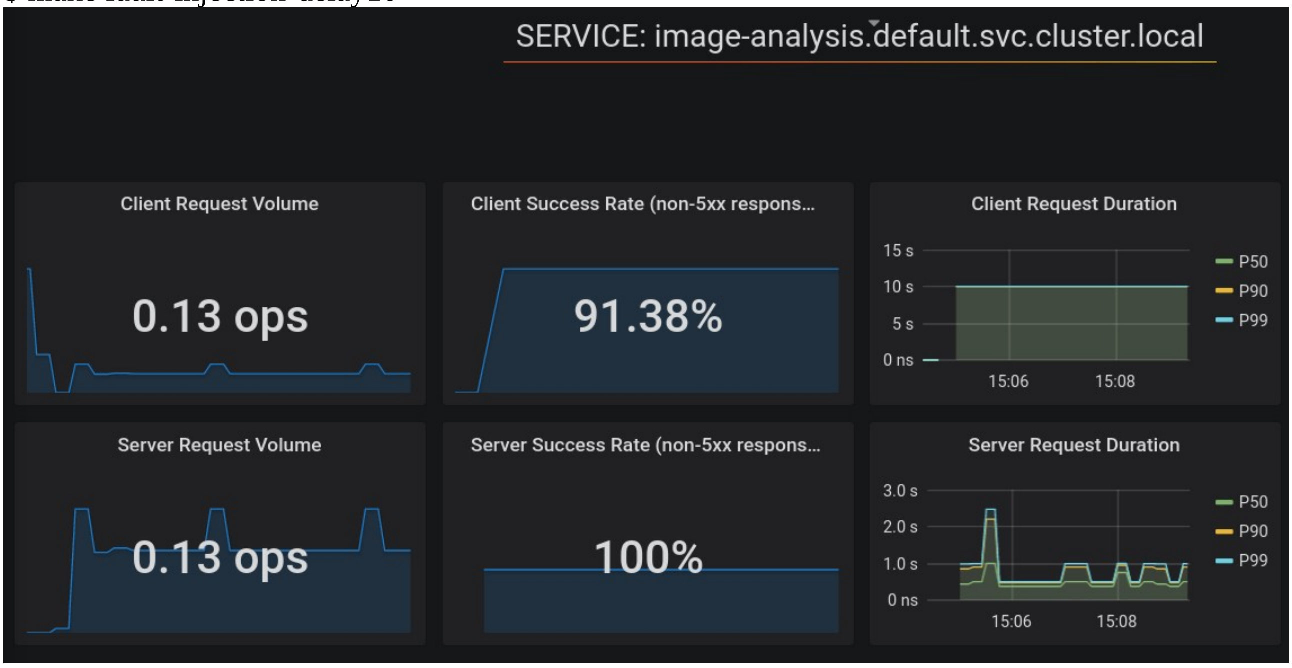
client workloads - workloads that are calling this service
service workloads - workloads that are providing this service

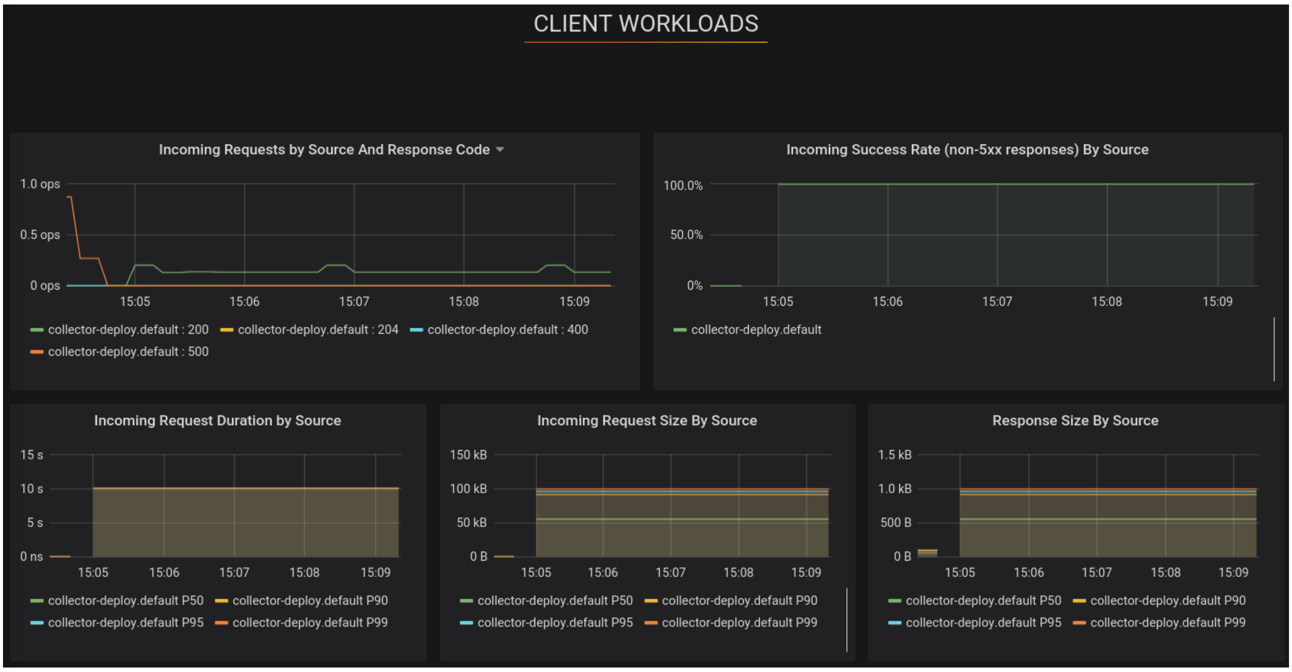
```
$ make fault-injection-500
```

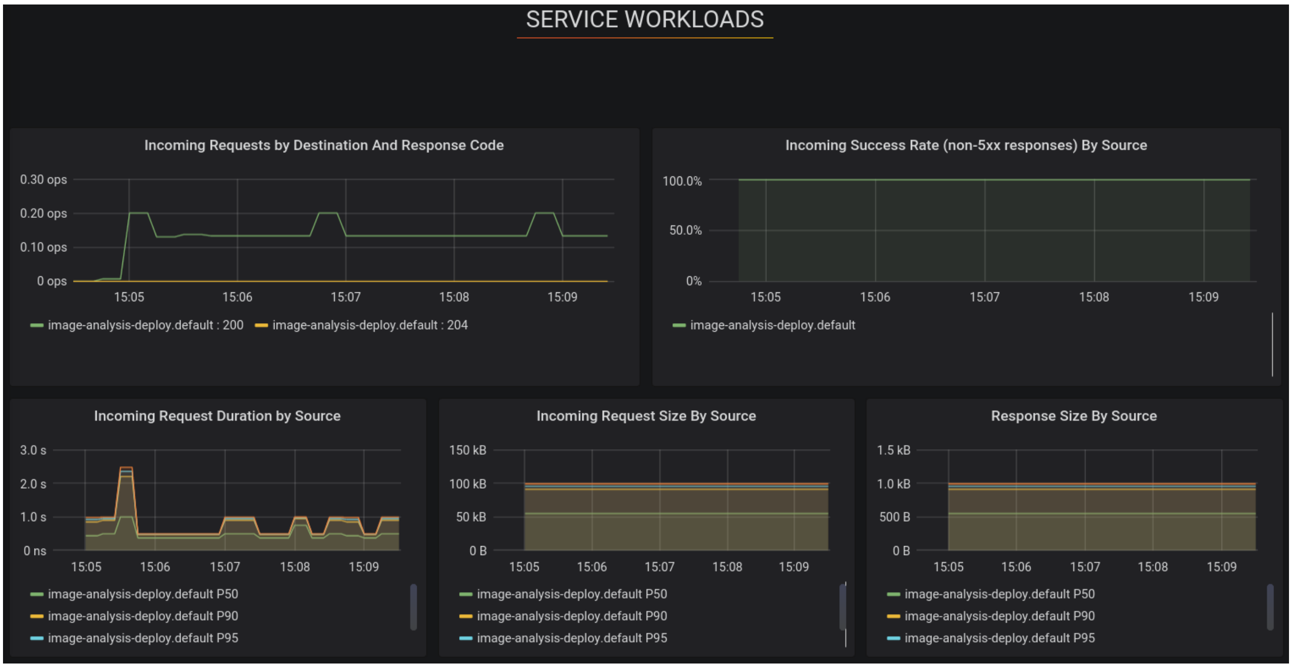




$ make fault-injection-delay10





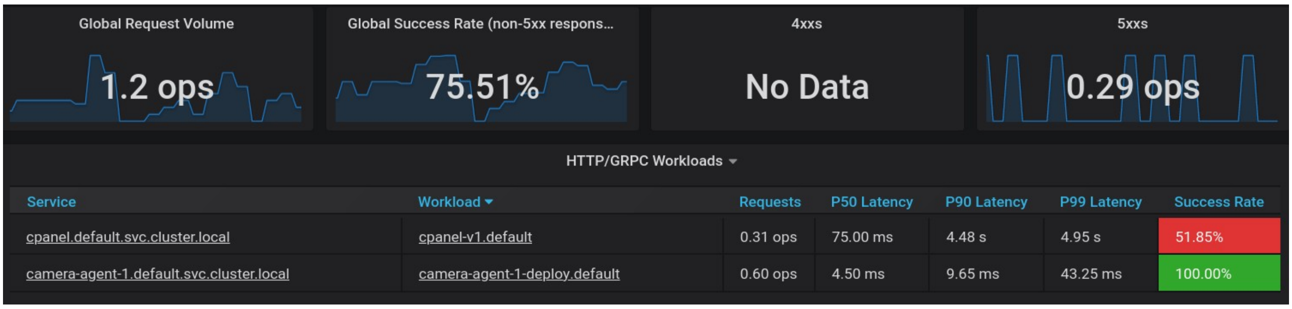


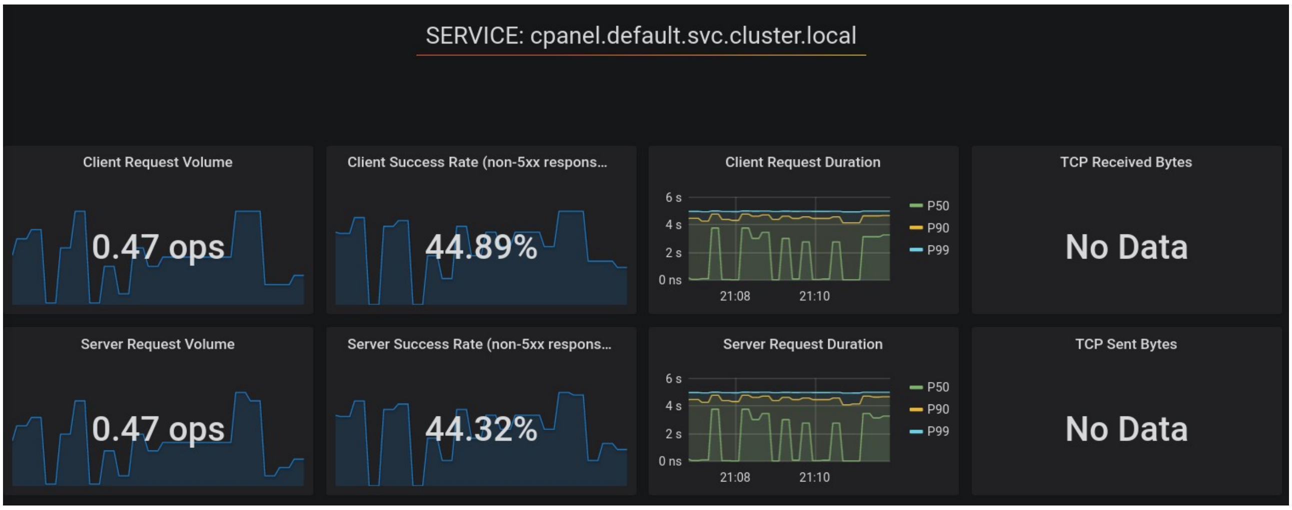
Timeout

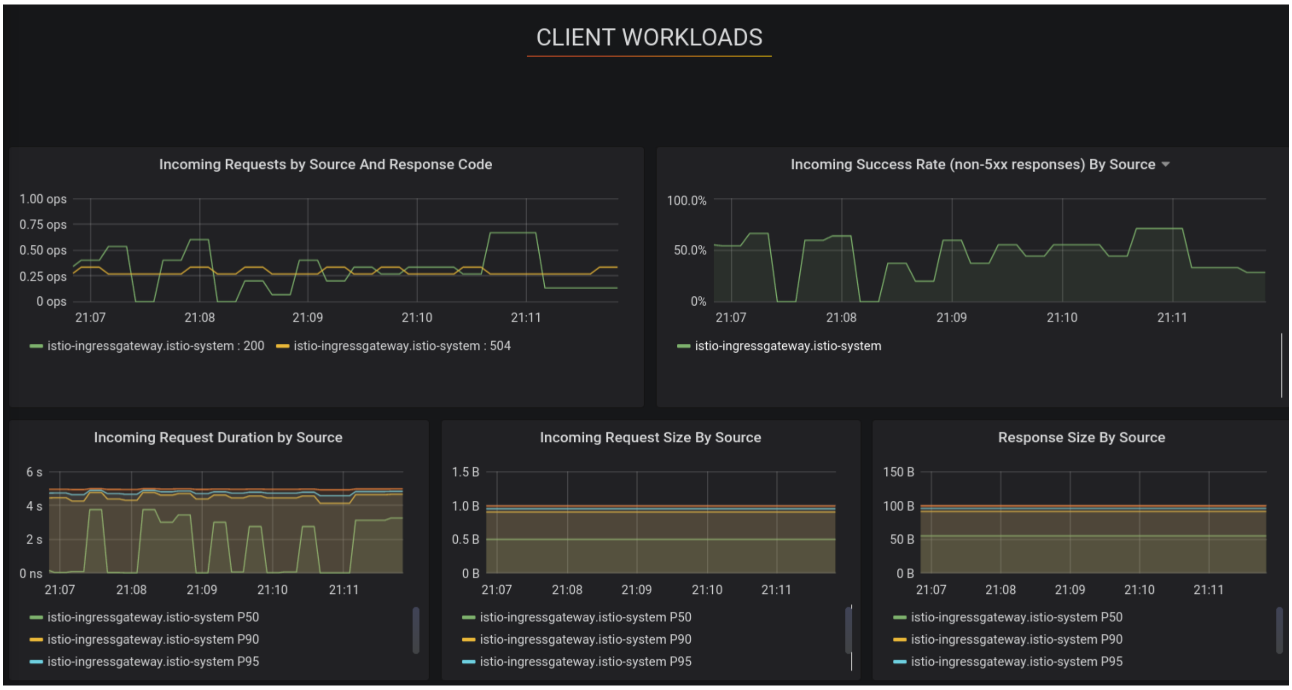
```
$ make timeout
./kubectl apply -f istio/timeout.yaml
virtualservice.networking.istio.io/camera-agent-1 configured
virtualservice.networking.istio.io/cpanel configured
```

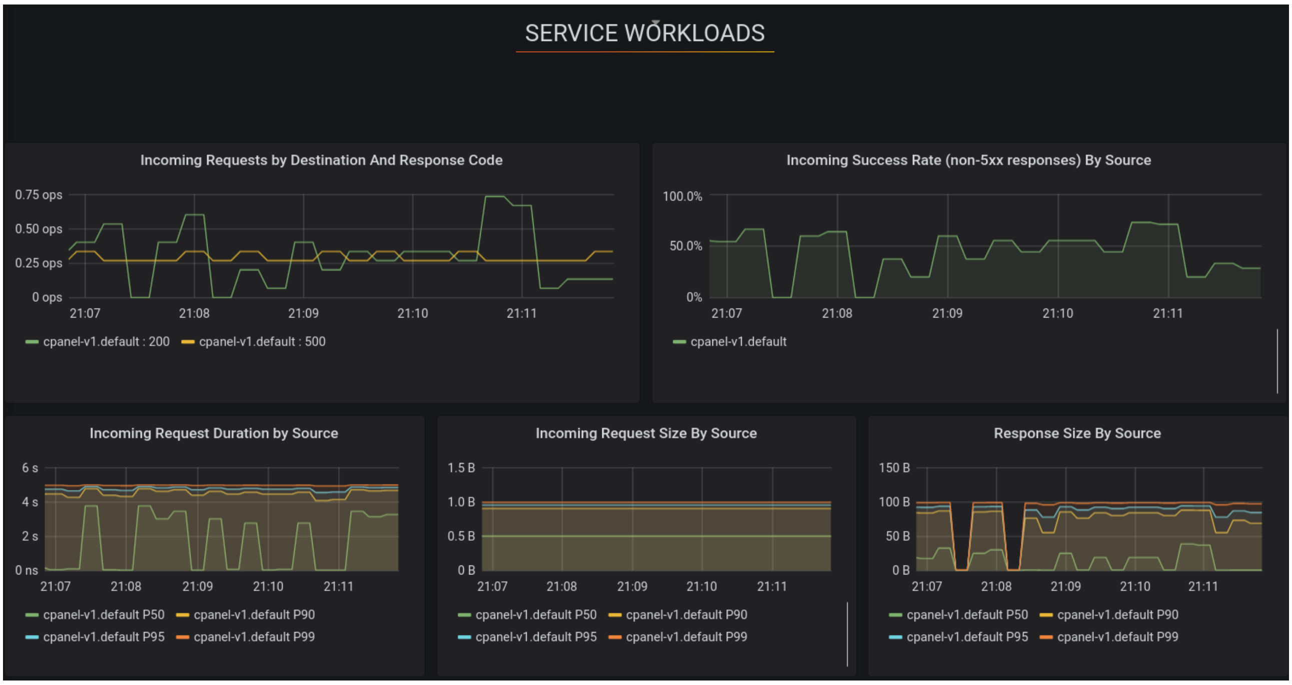

```
$ make health-timeout
for i in {1..10}; do sleep 0.2; curl http://192.168.99.113:31221/cameras/1/state; printf "\n"; done
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":nu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







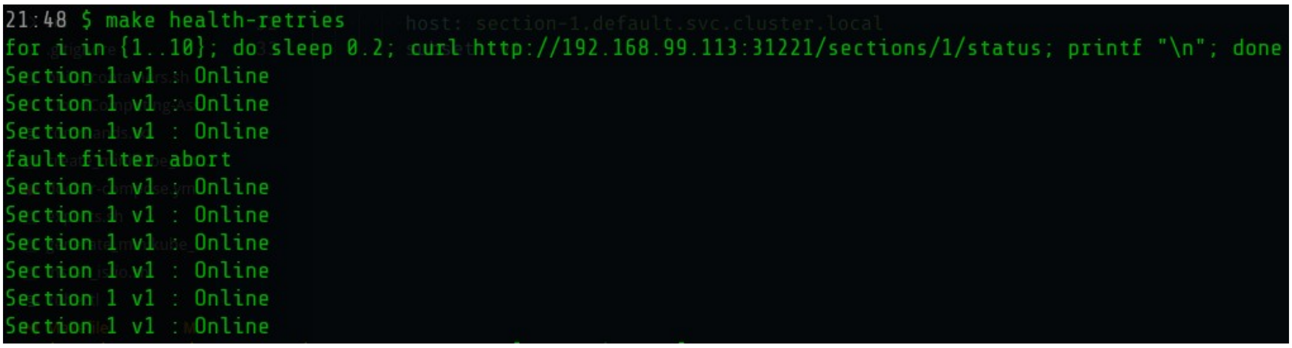


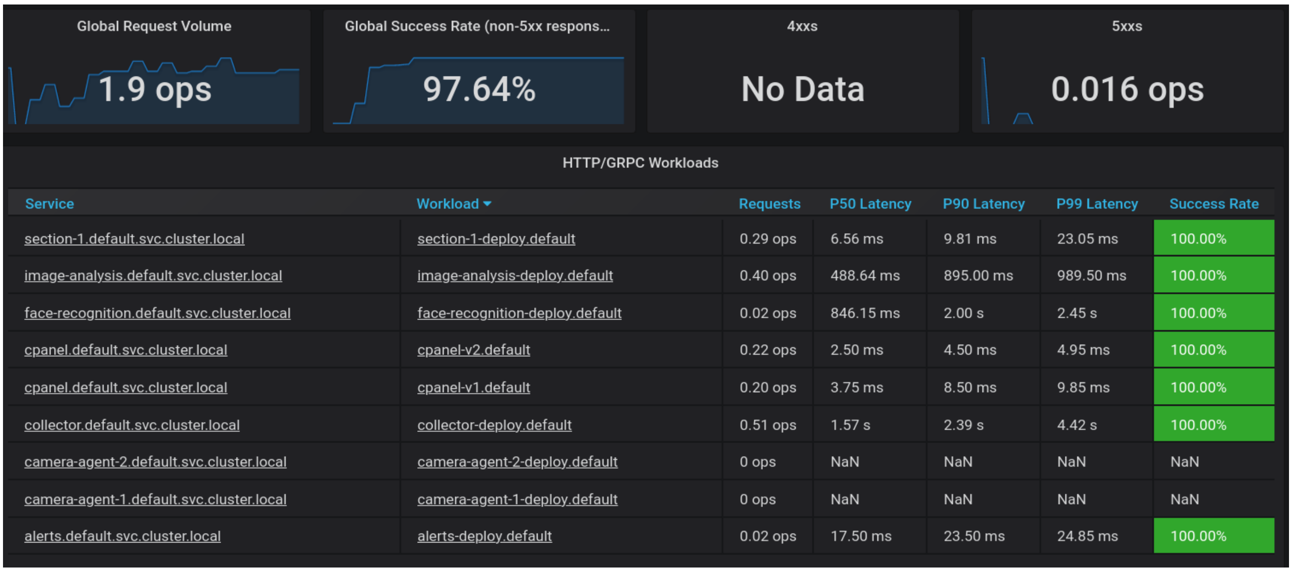
Retries

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

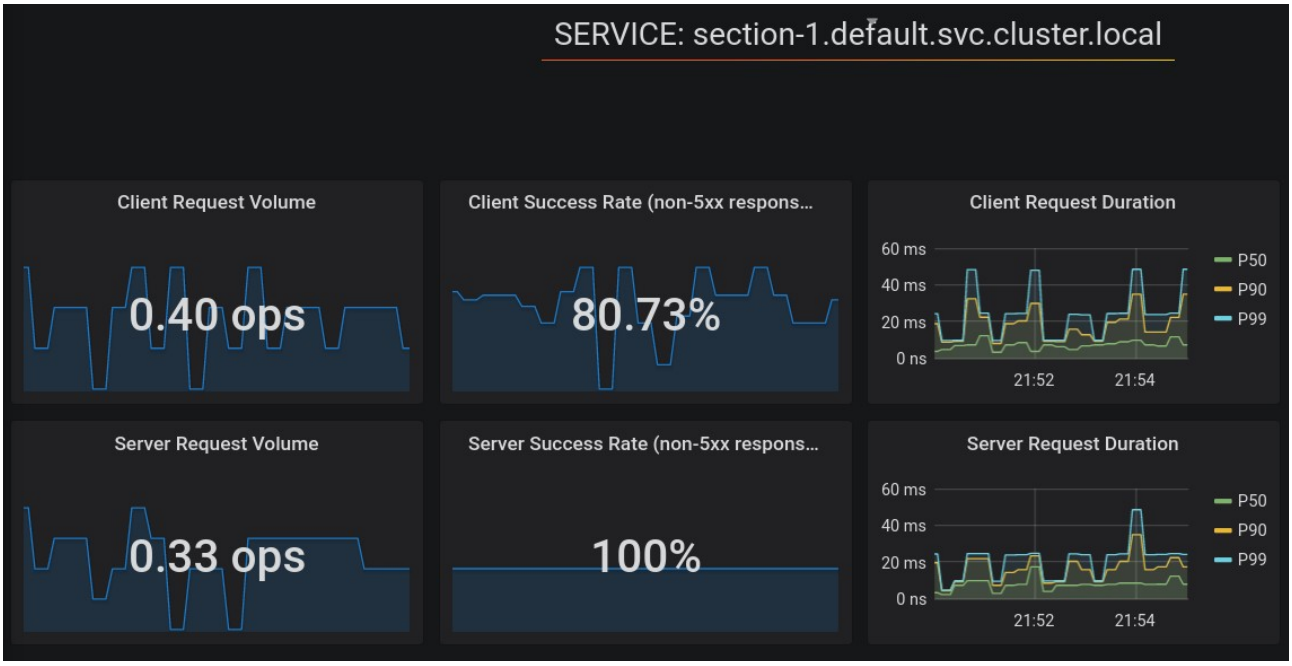
$ make health-retries
for i in {1..10}; do sleep 0.2; curl http://192.168.99.113:31221/sections/1/status; printf "\n"; done
Section 1 v1 : Online
Section 1 v1 : Online

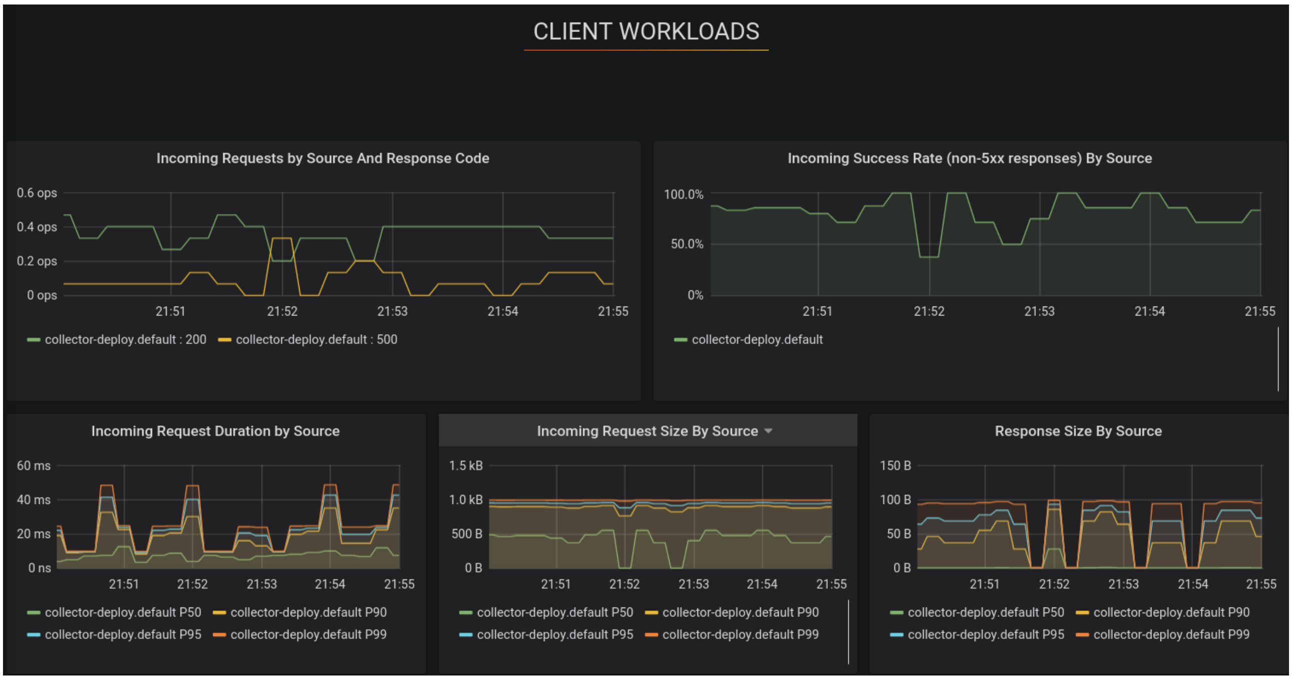
Section 1 v1 : Online
fault filter abort
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online

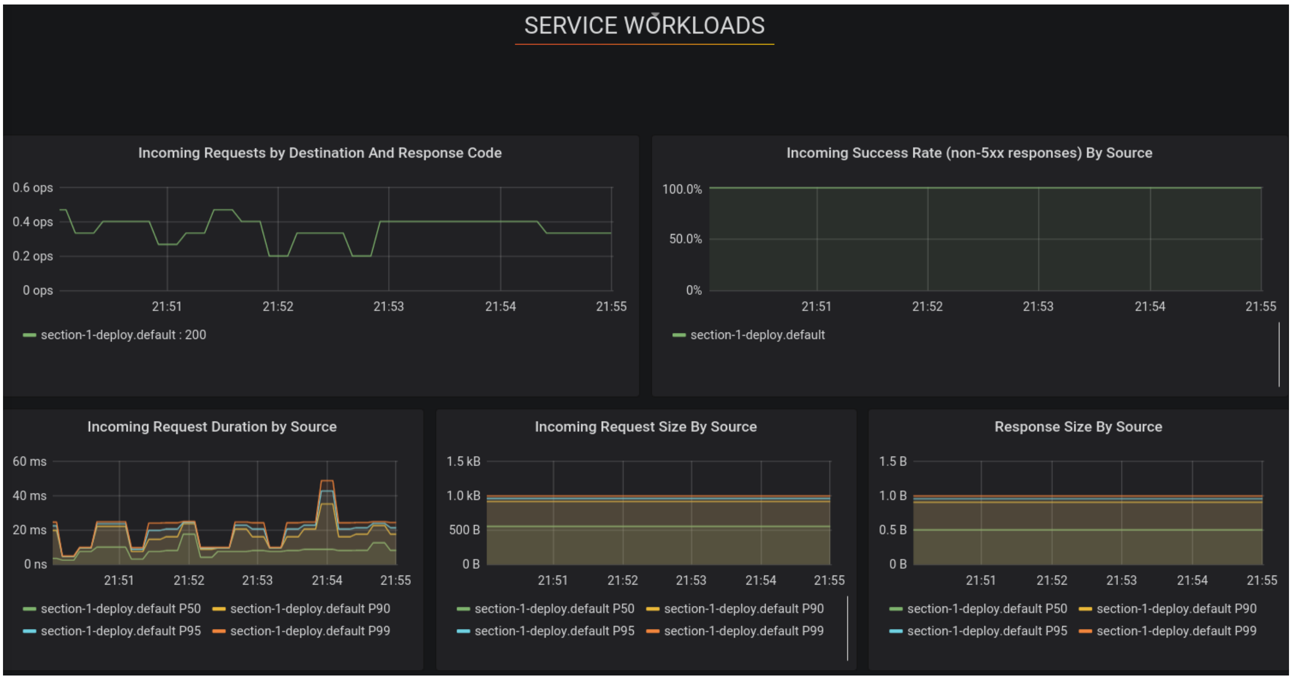

```
21:48 $ make health-retries
for i in {1..10}; do sleep 0.2; curl http://192.168.99.113:31221/sections/1/status; printf "\n"; done
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
fault filter abort
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
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

Circuit breaker pattern utilizes

Discussion

Conclusion

Istio offers great features in terms of resiliency for modern microservices applications. It helps with focus shift of operational overhead from developers to oprations departments.

- 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/
11. (action)Microservices in action. Book
12. (towards)Towards an Understanding of Microservices. Proceedings of the 23rd International Conference on Automation & Computing, University of Huddersfield, Huddersfield, UK, 7-8 September 2017
13. (native) Guide to Cloud Native Microservices. The new stack

Supplemental Material

- cc assignment
- commands