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, REST, Containers, Docker, Kubernetes, Service Mesh, Istio, resiliency, fault tolerance

Table of Contents

Introduction

migration to **clouds** → microservices, devops, fast code-to-market, leave only business logic for developers
Transition to microservices architecture helped to make application deployments more cloud friendly. Automation, scalability and continuous deployment are among the most valuable attributes coming with this architectural changes in software engineering process [10years]. All this factors and independence between microservices bring application resiliency on completely new level [migrate].
Containers **vs** vm
adoption of containers and automated deployments[10years], 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. The problem of delivering code from developers to productions is solved with packaging application and dependencies in images.

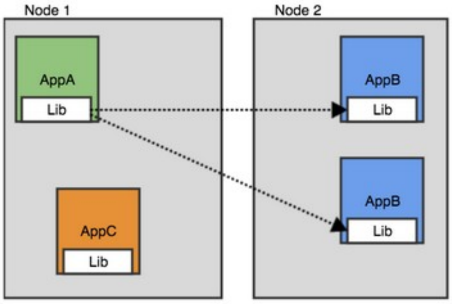
kubernetes just orchestration, no possibility to deal with network errors – focus on pods
number of **microservices** grows, lack of visibility and control,

goals, metrics: deploy microservices app, compare resiliency with and without istio
cc project as template (refactored, adopted), deploy istio, demo in minikube, test resiliency

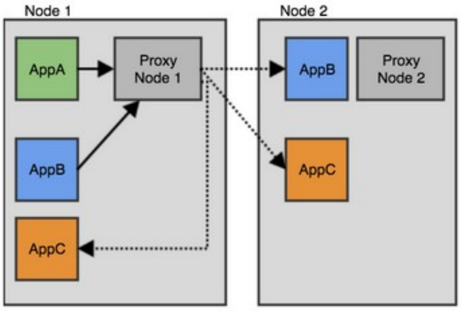
Related work

Need for service meshes. Many digital firms try to achieve results in developing or adoption of new service meshes solutions. As there is a big demand in service meshes there are also solutions with completely different architectures on the market.
Pictures from [alt]

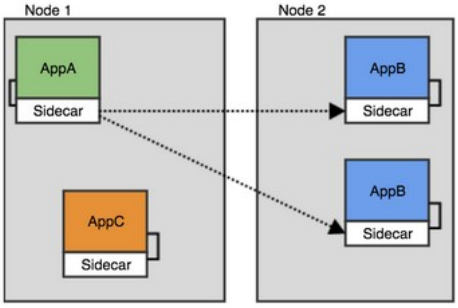
- libs: cons - code change (hystrix, ribbon)



- node agent (linkerd)



- sidecar (istio, linkerd2, consul)



Major idea

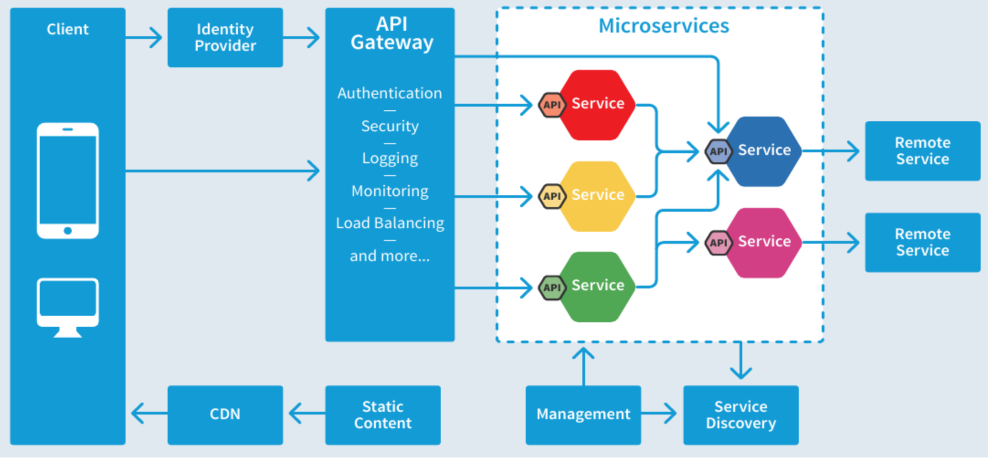
There are plenty of tutorials online that utilize a sample application from istio web site (“Bookinfo” application) to show typical service mesh and specific istio features. The idea of this thesis is to take the already implemented project, but not the one from istio, adopt it a little bit and provide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

Microservices

Motivation
Virtualization was not meant to speed application development.
The idea of microservices attributes can be compared with UNIX ideas [flexible]:

- one program – one task
- universal interface for all programs
- programs communicate with one another

Ideally one developer should be enough to understand the idea of one special microservice and develop it further on [10years].
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 storage
- 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

Why? Well, **Docker** solves one big thing: the **packaging** problem. By allowing you to package your app and its (non-network) runtime dependencies into a container, your app is now a fungible unit that can be thrown around and run anywhere. By the same token, Docker makes it exponentially easier to run a *polyglot* stack: because the container is an atomic unit of execution, for deploy and operational purposes it doesn't really matter what's inside the container, and whether it's a JVM app or a Node app or Go or Python or Ruby. You just run it.
Kubernetes solves the next step: now that I have a bunch of "executable things", and I also have a bunch of "things that can execute these executable things" (aka machines), I need a mapping between them. In a broad sense, you give **Kubernetes** a bunch of containers and a bunch of machines, and it figures out this **mapping**. (Which of course is a dynamic and ever-shifting thing, as new containers roll through the system, machines come in and out of operation, and so on. But Kubernetes figures it out.) [mesh]

Service mesh

Motivation

If I had to put it into a single sentence, the value of the service mesh comes down to this: **The service mesh gives you features that are critical for running modern server-side software in a way that's uniform across your stack and decoupled from application code.[mesh]**

operators should manage microservices apps in **large** hybrid and multi-cloud deployments.
Tracing solves a common problem in microservices systems. A request to a microservice might result in other requests. Tracing helps to understand these dependencies, thus facilitating root cause analysis.
Logging is another important technology to gain more insight into a system. A service

mesh collects information about the network communication. The logging support of a service mesh has the advantage that developers do not have to care about these logs at all. Besides, the logs are **uniform** no matter what kind of technology is used in the microservices and how they log. Enforcing a common logging approach and logging format takes some effort.

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.

Service mesh provides:

service discovery, LB, resiliency, security (end-to-end encryption, authorization), observability (layer 7 metrics, tracing, alerting), routing control, API (programmable interface, kubernetes CRD)

focus on east-west traffic, manage and control services inside network, brokerin internal resources, is between network and application – no business logic.

service mesh - network of microservices that make up distributed microservice applications and the interactions between them.

- requirements - discovery, load balancing, failure recovery, metrics, and monitoring.
- also operational- A/B testing, canary rollouts, rate limiting, access control and end-to-end authentication.

service mesh focuses on networking between microservices rather than business logic

[mesh]
What are these proxies? They’re **Layer 7**-aware TCP proxies, just like haproxy and NGINX.
What do these proxies do? They proxy calls to and from the services. (they act as both “proxies” and “reverse proxies”, handling both **incoming** and **outgoing** calls.) And they implement a featureset that focuses on the calls *between* services. This focus on **traffic between services** is what differentiates service mesh proxies from, say, API gateways or ingress proxies, which focus on calls from the outside world into the cluster as a whole.

pros/cons

Istio

Istio is described as a tool to connect, secure, control, and observe services - a service mesh for microservices application.
It’s designed to add application-level Layer (L7) observability, routing, and resilience to service-to-service traffic and this is what we call “east-west” traffic.
tracing, monitoring, and logging – deep view of microservice deployment

sidecars – to get plenty of signals about traffic that are used in mixer for policies and also for monitoring.

  - service mesh
    - **security** – who talks whom, trusted communication, encryption
    - **observability** – tracing of requests, metrics, alerting, topology
    - **traffic management** - routing control
      - load balancing
      - communication resiliency
      - mirroring – OK if service is read-only to avoid computing overload
    - **API** (kubernetes CRD)
    - policies – rate limits, denials and white/black listing

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

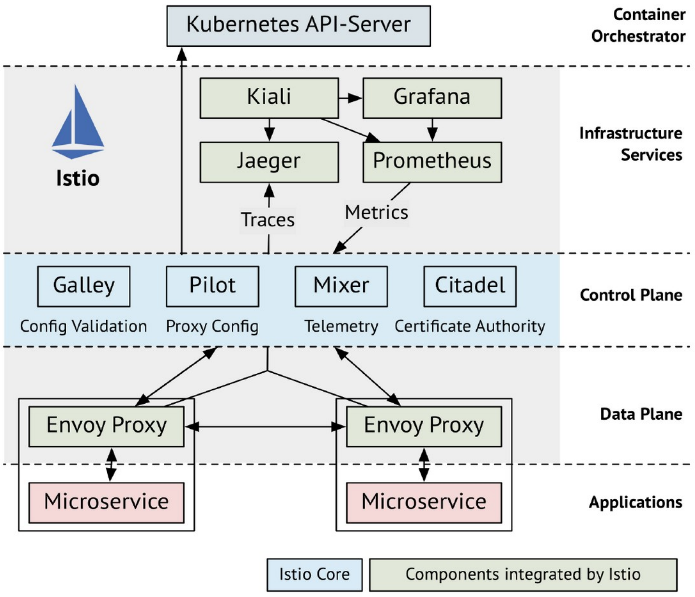
puts resiliency into the infrastructure
An Istio service mesh is logically split into a **data plane** and a **control plane**.

- The **data plane** is composed of a set of intelligent proxies (Envoy) deployed as sidecars. These proxies mediate (**intercept**) and **control** all network communication between microservices along with **Mixer**, a general-purpose policy and telemetry hub.
- The **control plane** manages and configures the proxies to **route** traffic. Additionally, the control plane configures Mixers to enforce policies and collect telemetry.

Traffic in Istio is categorized as **data plane traffic** and **control plane traffic**. Data plane traffic refers to the messages that the business logic of the workloads send and receive. Control plane traffic refers to configuration and control messages sent between Istio components to program the behavior of the mesh. **Traffic management** in Istio refers exclusively to data plane traffic.

control plane functionality:

- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas.
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress.
- Secure service-to-service communication in a cluster with strong identity-based authentication and authorization.



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, add external services to istio registry, enables retry, timeout, fault-injection
Gateway and ServiceEntriy control the north-south traffic (incoming and outgoing)
VirtualService and DestinationRule control the east-west traffic (inside service mesh)
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.

Resiliency in istio: health checks, load balancing, delay injection, fault injection, timeouts, retries, rate limits, circuit breaker.

Resilience means that individual microservices still **work** even if other microservices **fail**. If a microservice calls another microservice and the called microservice fails, this will have an impact. Otherwise, the microservice would not need to be called at all. So the calling microservice will behave differently and might not be able to respond successfully to each request. However, the microservice **must** still **respond**. It must not block a request because then other microservices might be blocked and an error **cascade** might occur. Also **delays** in the network communication might lead to such problems.

What can go wrong in a Microservice architecture?

There are a number of moving components in a Microservice architecture, hence it has more points of failures. Failures can be caused by a variety of reasons – errors and exceptions in code, release of new code, bad deployments, hardware failures, datacenter failure, poor architecture, lack of unit tests, communication over the unreliable network, dependent services etc.

Why do you need to make service resilient?

A problem with Distributed applications is that they communicate over network – which is unreliable. Hence you need to design your microservices in such a way that they are fault tolerant and handle failures gracefully. In your microservice architecture, there might be a dozen of services talking with each other. You need to ensure that one failed service does not bring down the entire architecture.

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.

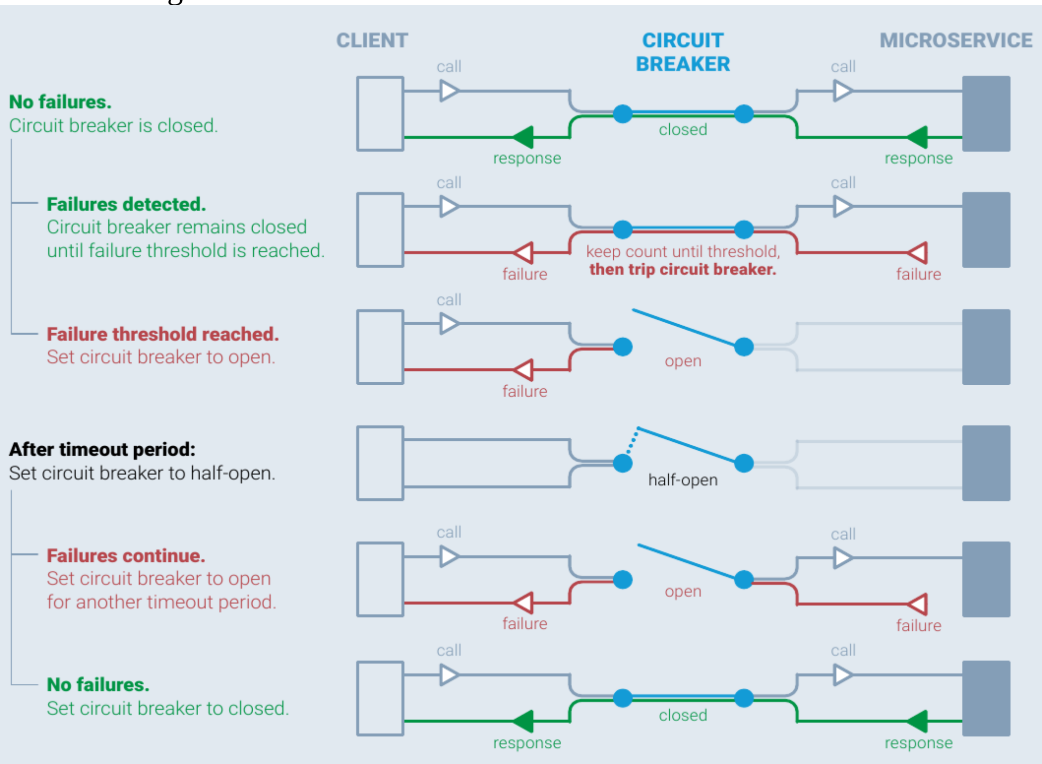
```
        percent: 100
        fixedDelay: 2s
```

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.



CLIENT
CIRCUIT BREAKER
MICROSERVICE
No failures.
Circuit breaker is closed.
call
closed
response
Failures detected.
Circuit breaker remains closed until failure threshold is reached.
failure
keep count until threshold, then trip circuit breaker.
Failure threshold reached.
Set circuit breaker to open.
open
After timeout period:
Set circuit breaker to half-open.
half-open
Failures continue.
Set circuit breaker to open for another timeout period.
No failures.
Set circuit breaker to closed.

[native]

General explanation - ...

We can see two 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 development 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.

The IP address of istio ingress can be different from test to test, because new cluster was installed multiple times while working on the implementation part.

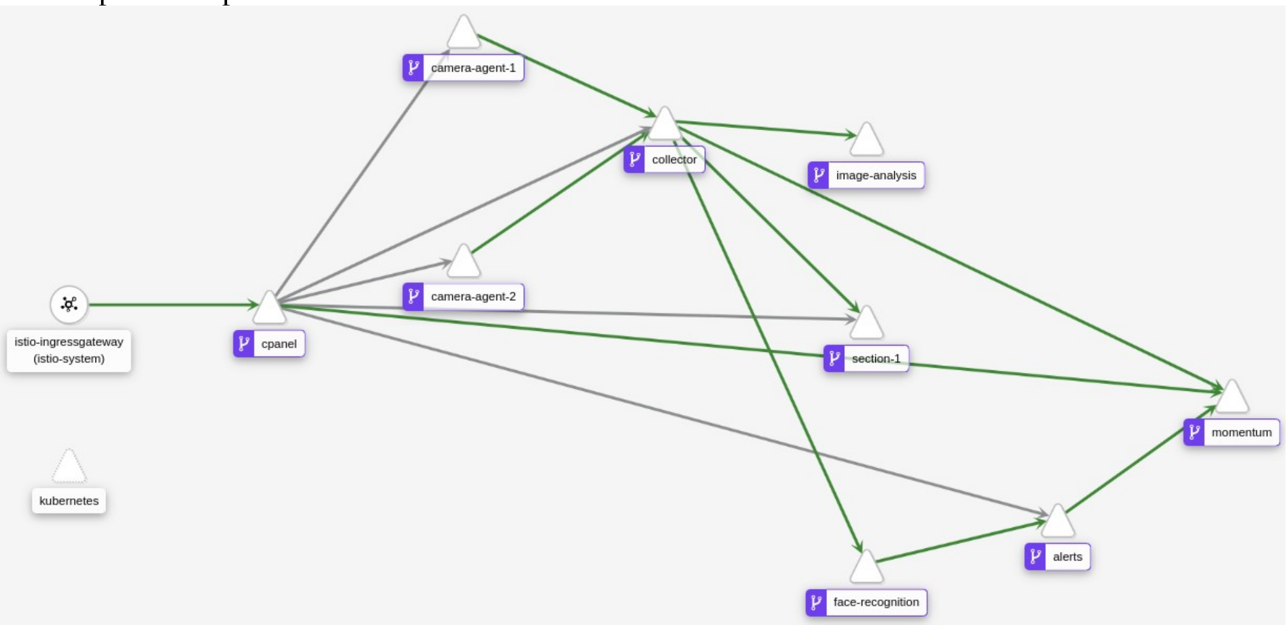
Implementation

The Twelve Factors Application

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 appropriate section and momentum microservice.
Momentum microservice serves to store current processing frames with information about them from image analysis and face recognition.
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.
Alert microservice provides API to create, read and delete alerts from database (json file) and also forwards the response from face recognition to momentum microservice.

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. Momentum microservice was added to separate temporally logic of saving current processing frames. Frontend was added to cpanel microservice was minimal functionality – just to display currently processed frames from momentum microservice.

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:	GET /status
collector:	GET /status
section:	GET /status
cpanel:	GET /status GET /, /index GET /analysis GET /alert
momentum:	GET /status GET/POST /analysis GET/POST /alert

The Twelve Factors App [twelve]

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

- frontend v1/v2
  - canary, blue/green deployment, user resiliency
- python + docker best practices:
  - alpine, root, no cache
- scaling deployment:
  - collector, image-analysis, face-recognition
- momentum microservice
- 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

Replication of pods is configured for collector, image analysis and face recognition.

Understanding namespaces and DNS
When you create a Service, it creates a corresponding DNS entry. This entry is of the form <service-name>.<namespace-name>.svc.cluster.local, which means that if a container just uses <service-name> it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).

Labels are k8s and its end users way to filter similar resources in the system.
Annotations are very similar to labels, but are usually used to keep metadata for different objects in the form of freestyle strings.
Services provide stable endpoints for Pods. if pod restart - new IP.
There is a Label selector which determines the pods which services target. Service without selector is not possible.

Deploy with Istio

single cluster deployment
istio verify install done in script
virtual services , destination rules for subsets, ingress gateway
gateway is added to cpanel virtual service to expose it outside of minikube cluster
Mirroring can be enabled on momentum microservice with the same version. In such a way we achieve additional resiliency for this read-only service. As there is no business logic and so no computing overload it is quite acceptable. Mirroring can be done in VirtualService in istio.
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

Here resiliency features of istio service mesh will be introduced in practice. Kiali graphs, grafana graphics and console outputs will help to understand how fault tolerance can be configured with istio.

Running application

```
$ make deploy-app-default
./kubectl apply -f k8s
./kubectl get pods –watch
```

Wait till all pods are up and running and stop monitoring them with Ctrl-c.

```
$ make deploy-istio-default
./kubectl apply -f istio/dest_rule_all.yaml
./kubectl apply -f istio/virt_svc_all.yaml
./kubectl apply -f istio/ingress_gateway.yaml
```

Check that application is deployed properly with istio configuration files.

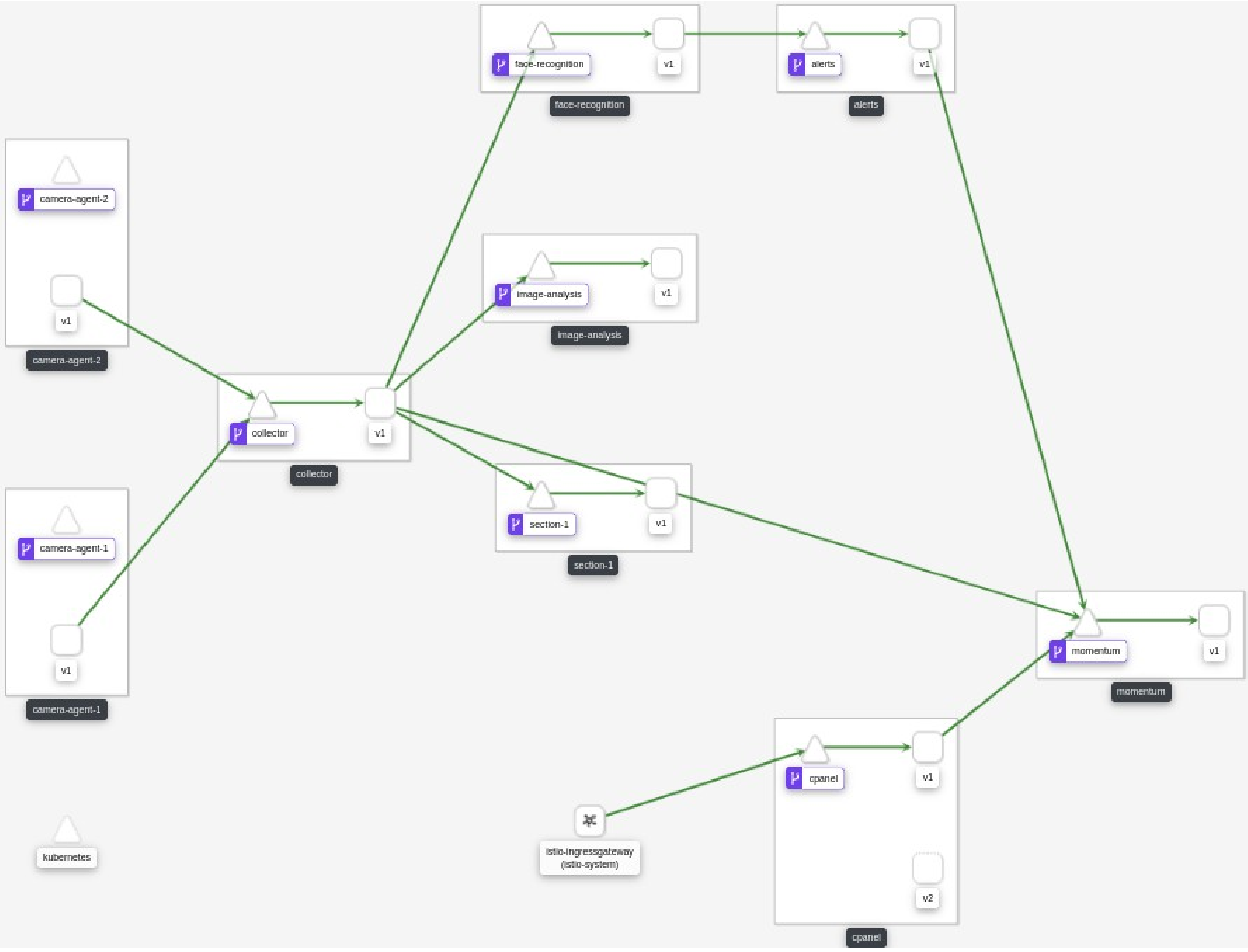
```
$ make health
curl http://192.168.99.113:31221/status
CPanel v1 : Online
curl http://192.168.99.113:31221/cameras/1/state
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
curl http://192.168.99.113:31221/cameras/2/state
{"streaming":false,"cycle":0,"fps":0,"section":null,"destination":null,"event":null}
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/momentum/status
Momentum v1 : Online
```

```
$ make start-cameras
curl http://192.168.99.113:31221/production?toggle=on
```

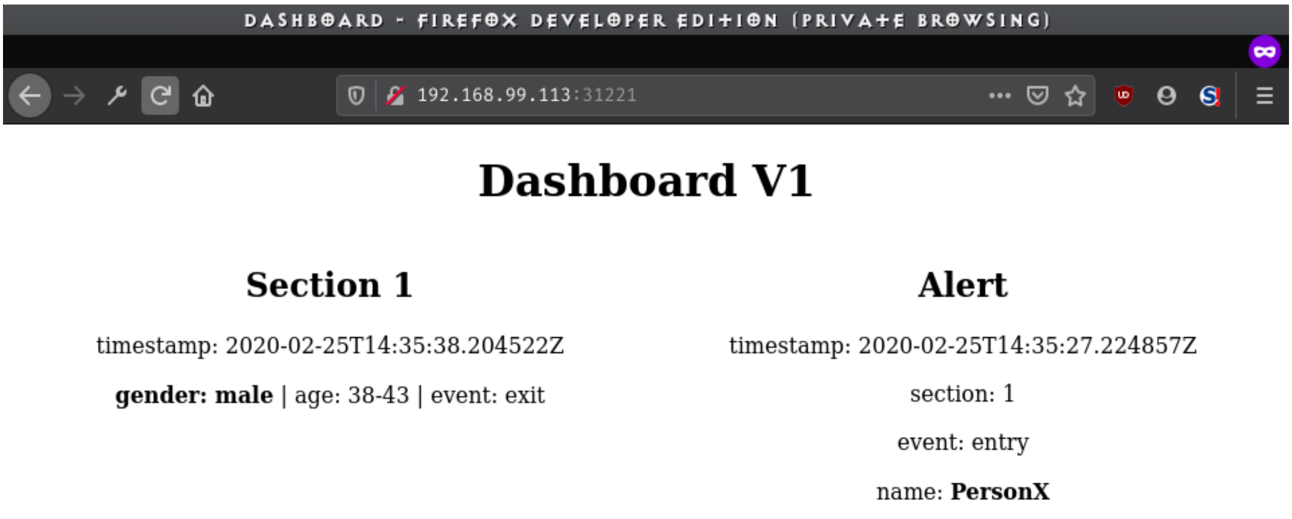
```
$ make health
curl http://192.168.99.113:31221/status
CPanel v1 : Online
curl http://192.168.99.113:31221/cameras/1/state
{"streaming":true,"cycle":8,"fps":0,"section":"1","destination":"http://
collector.default.svc.cluster.local:8080","event":"exit"}
curl http://192.168.99.113:31221/cameras/2/state
{"streaming":true,"cycle":6,"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
curl http://192.168.99.113:31221/momentum/status
Momentum v1 : Online
```

```
$ make kiali
istio-1.4.3/bin/istioctl dashboard kiali
http://localhost:44517/kiali
$ ./kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l app=grafana -o jsonpath='{.items[0].metadata.name}') 3000:3000 &
```



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. Splitting between admin users and normal users can be done in virtual service with help of headers. 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

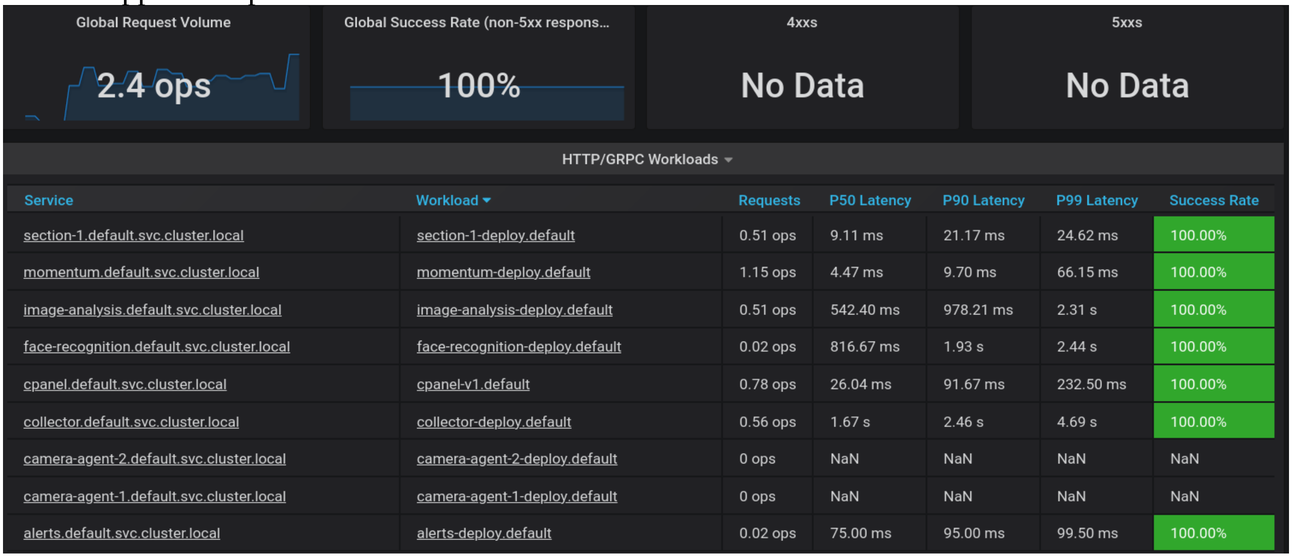
timestamp: 2020-02-25T14:35:27.224857Z

section: 1

event: entry

name: **PersonX**

Default app with Cpanel v1 without load to frontend:



Global Request Volume: 2.4 ops

Global Success Rate (non-5xx respons...: 100%

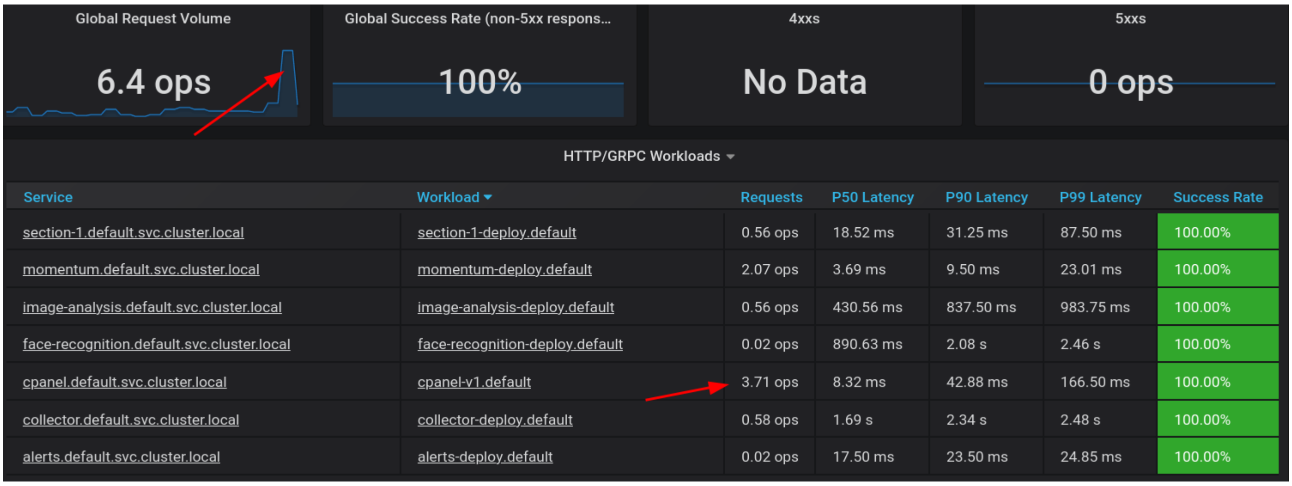
4xxs: No Data

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	0.51 ops	9.11 ms	21.17 ms	24.62 ms	100.00%
momentum.default.svc.cluster.local	momentum-deploy.default	1.15 ops	4.47 ms	9.70 ms	66.15 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.51 ops	542.40 ms	978.21 ms	2.31 s	100.00%
face-recognition.default.svc.cluster.local	face-recognition-deploy.default	0.02 ops	816.67 ms	1.93 s	2.44 s	100.00%
cpanel.default.svc.cluster.local	cpanel-v1.default	0.78 ops	26.04 ms	91.67 ms	232.50 ms	100.00%
collector.default.svc.cluster.local	collector-deploy.default	0.56 ops	1.67 s	2.46 s	4.69 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	75.00 ms	95.00 ms	99.50 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
…
```



Global Request Volume: 6.4 ops

Global Success Rate (non-5xx respons...: 100%

4xxs: No Data

5xxs: 0 ops

HTTP/GRPC Workloads

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default	0.56 ops	18.52 ms	31.25 ms	87.50 ms	100.00%
momentum.default.svc.cluster.local	momentum-deploy.default	2.07 ops	3.69 ms	9.50 ms	23.01 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.56 ops	430.56 ms	837.50 ms	983.75 ms	100.00%
face-recognition.default.svc.cluster.local	face-recognition-deploy.default	0.02 ops	890.63 ms	2.08 s	2.46 s	100.00%
cpanel.default.svc.cluster.local	cpanel-v1.default	3.71 ops	8.32 ms	42.88 ms	166.50 ms	100.00%
collector.default.svc.cluster.local	collector-deploy.default	0.58 ops	1.69 s	2.34 s	2.48 s	100.00%
alerts.default.svc.cluster.local	alerts-deploy.default	0.02 ops	17.50 ms	23.50 ms	24.85 ms	100.00%

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. 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.

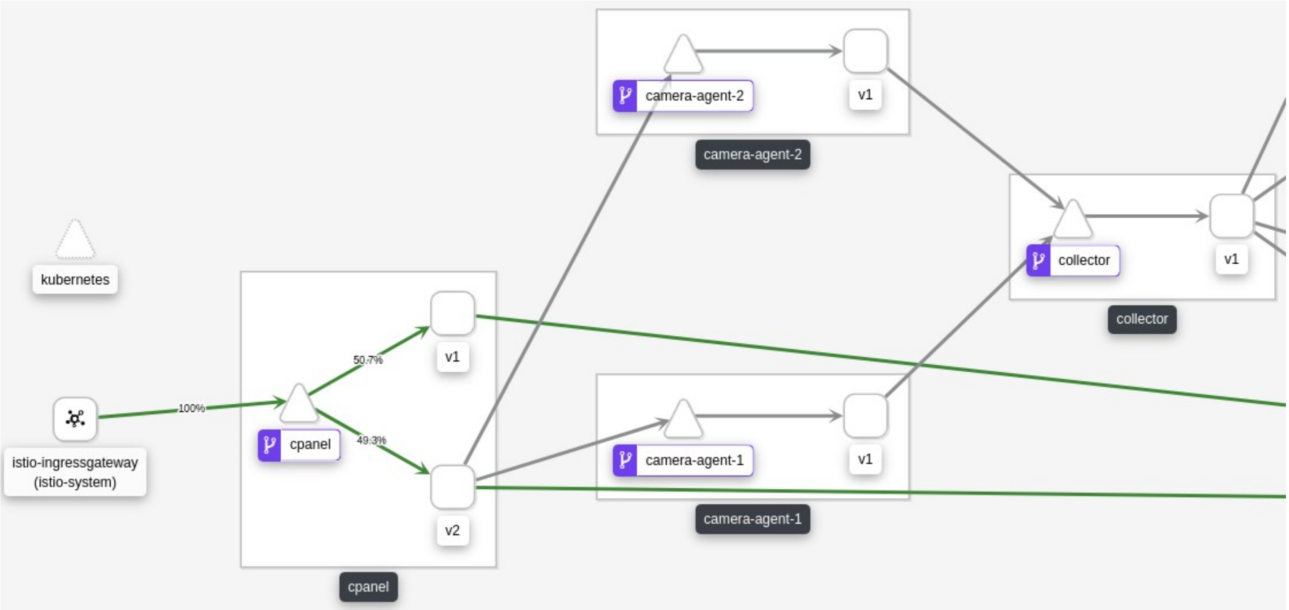
```
route:
- destination:
    host: cpanel.default.svc.cluster.local
    port:
      number: 8080
    subset: v1
  weight: 50
- destination:
    host: cpanel.default.svc.cluster.local
    port:
      number: 8080
    subset: v2
  weight: 50
```

```
$ make cpanel-50-50
./kubectl apply -f istio/virt_svc_50-50.yaml
virtualservice.networking.istio.io/cpanel configured
```

check configuration

```
$ ./kubectl get virtualservices cpanel -o yaml
```

```
$ make load-front
for i in {1..100}; do sleep 0.2; curl --silent http://192.168.99.113:31221/ | grep -o "<h1>.*</h1>";
done
<h1>Dashboard V2</h1>
<h1>Dashboard V2</h1>
<h1>Dashboard V1</h1>
<h1>Dashboard V2</h1>
<h1>Dashboard V1</h1>
<h1>Dashboard V1</h1>
<h1>Dashboard V2</h1>
```



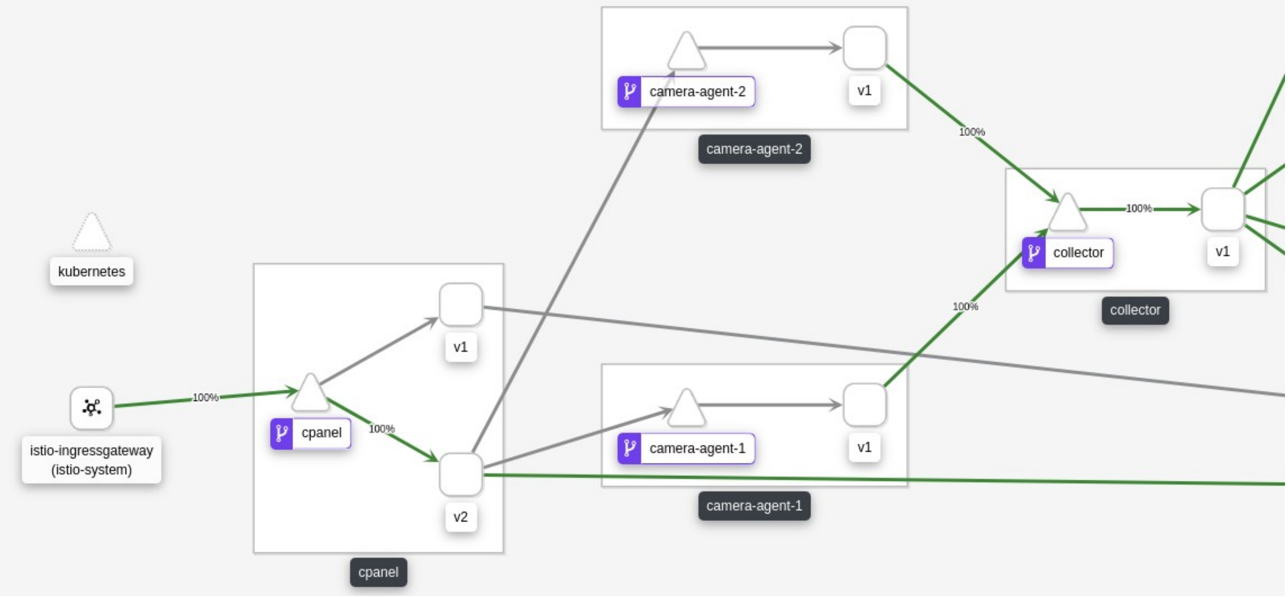
```
route:
- destination:
    host: cpanel.default.svc.cluster.local
    port:
      number: 8080
    subset: v1
  weight: 0
- destination:
    host: cpanel.default.svc.cluster.local
    port:
      number: 8080
    subset: v2
  weight: 100
```

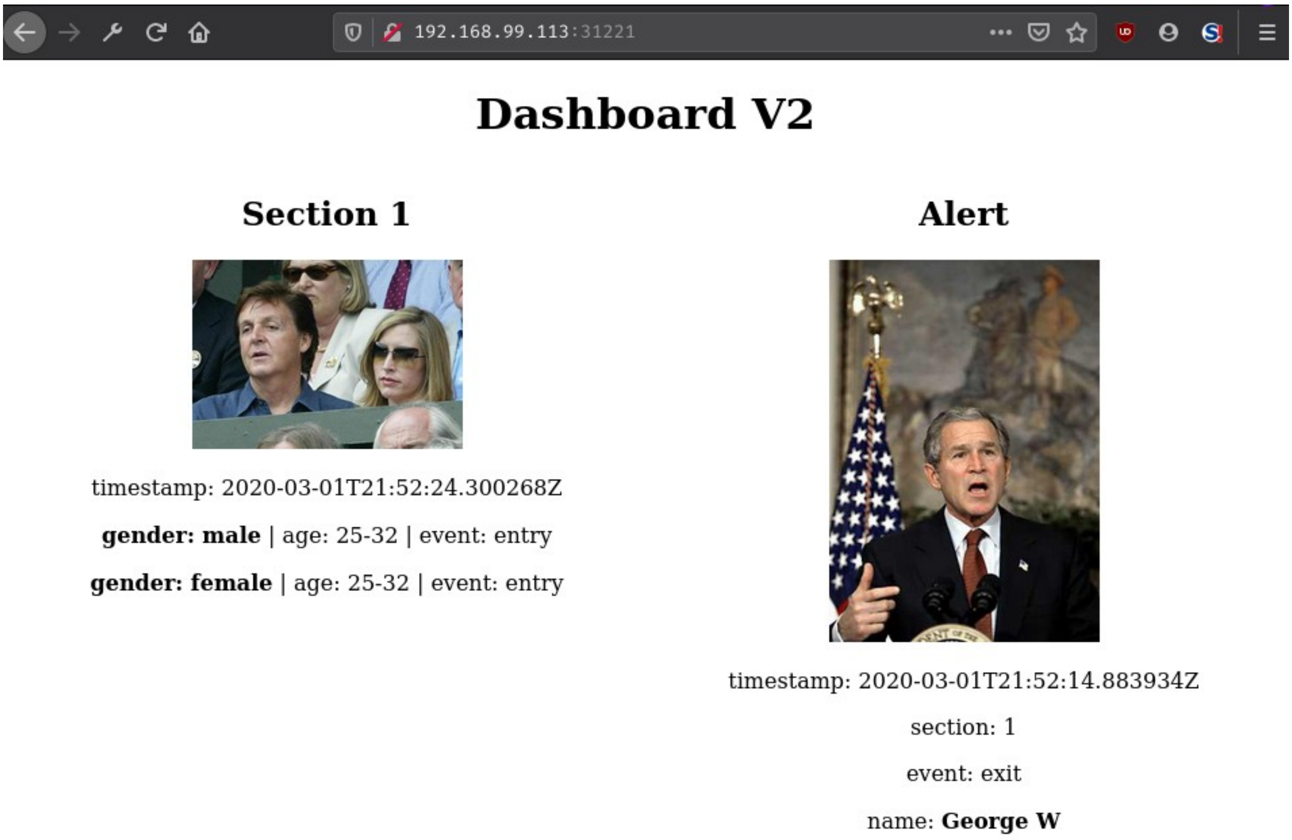
```
$ make cpanel-v2
./kubectl apply -f istio/virt_svc_v2.yaml
virtualservice.networking.istio.io/cpanel configured
```

check configuration

```
$ k get virtualservices cpanel -o yaml
```

```
$ make start-cameras
curl http://192.168.99.113:31221/production?toggle=on
```





Dashboard V2

Section 1

timestamp: 2020-03-01T21:52:24.300268Z

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

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

Alert

timestamp: 2020-03-01T21:52:14.883934Z

section: 1

event: exit

name: **George W**

Load balancing

To show advanced load balancing in istio we can scale our cpanel-v2 deployment to 3 replicas. Then default round robin load balancing, without any configurations between v1 and v2 cpanel virtual services, can be recognized (should be 1:3).

```
$ make scale_v2_x3
./kubectl scale deployment cpanel-v2 --replicas=3
deployment.extensions/cpanel-v2 scaled
```

here we can see how kubernetes scales our service

```
$ ./kubectl get deployments
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
cpanel-v1   1/1     1            1           3m52s
cpanel-v2   3/3     3            3           3m52s
```

There is no more subset version from destination rule in ingress virtual services. So istio will split all incoming traffic between running pods of cpanel service based on default round robin load balancing strategy.

```
route:
- destination:
    host: cpanel.default.svc.cluster.local
    port:
      number: 8080
```

```
$ make round_robin
./kubectl apply -f istio/round_robin_lb.yaml
virtualservice.networking.istio.io/cpanel configured
```

```
$ make load
for i in {1..100}; do sleep 0.2; curl http://192.168.99.114:32460/status; printf "\n"; done
CPanel v2 : Online - cpanel-v2-86f86bc679-z5s2q
CPanel v2 : Online - cpanel-v2-86f86bc679-5l2p5
CPanel v2 : Online - cpanel-v2-86f86bc679-srgws
CPanel v1 : Online - cpanel-v1-76864df47-hndph
CPanel v2 : Online - cpanel-v2-86f86bc679-z5s2q
CPanel v2 : Online - cpanel-v2-86f86bc679-5l2p5
CPanel v1 : Online - cpanel-v1-76864df47-hndph
CPanel v2 : Online - cpanel-v2-86f86bc679-z5s2q
CPanel v2 : Online - cpanel-v2-86f86bc679-srgws
```

Changing load balancing strategies in destination rules for cpanels we want to show that random load balancing will be applied to subsets v1 and v2. So the distribution of responses from services should be 50/50 in average. For load balancing between replicas of cpanel v2 round robin is used.

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

```
$ make load
for i in {1..100}; do sleep 0.2; curl http://192.168.99.114:32460/status; printf "\n"; done
CPanel v2 : Online - cpanel-v2-86f86bc679-z5s2q
CPanel v2 : Online - cpanel-v2-86f86bc679-5l2p5
CPanel v1 : Online - cpanel-v1-76864df47-hndph
CPanel v1 : Online - cpanel-v1-76864df47-hndph
CPanel v1 : Online - cpanel-v1-76864df47-hndph
CPanel v2 : Online - cpanel-v2-86f86bc679-srgws
CPanel v2 : Online - cpanel-v2-86f86bc679-5l2p5
```

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

```
./kubectl apply -f istio/dest_rule_all.yaml
./kubectl apply -f istio/virt_svc_all.yaml
./kubectl apply -f istio/ingress_gateway.yaml
$ ./kubectl scale deployment cpanel-v2 --replicas=1
```

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. This istio ability is extremely helpful while testing application deployments on resiliency. Operations department can test the configuration files of istio deployments with making any code changes for simulation of unhealthy behavior of microservices.

To try both of this features separately the following predefined configuration for our application can be used. Both of them should be configured in virtual services.

```
$ make fault-injection-500
$ make fault-injection-delay10
```

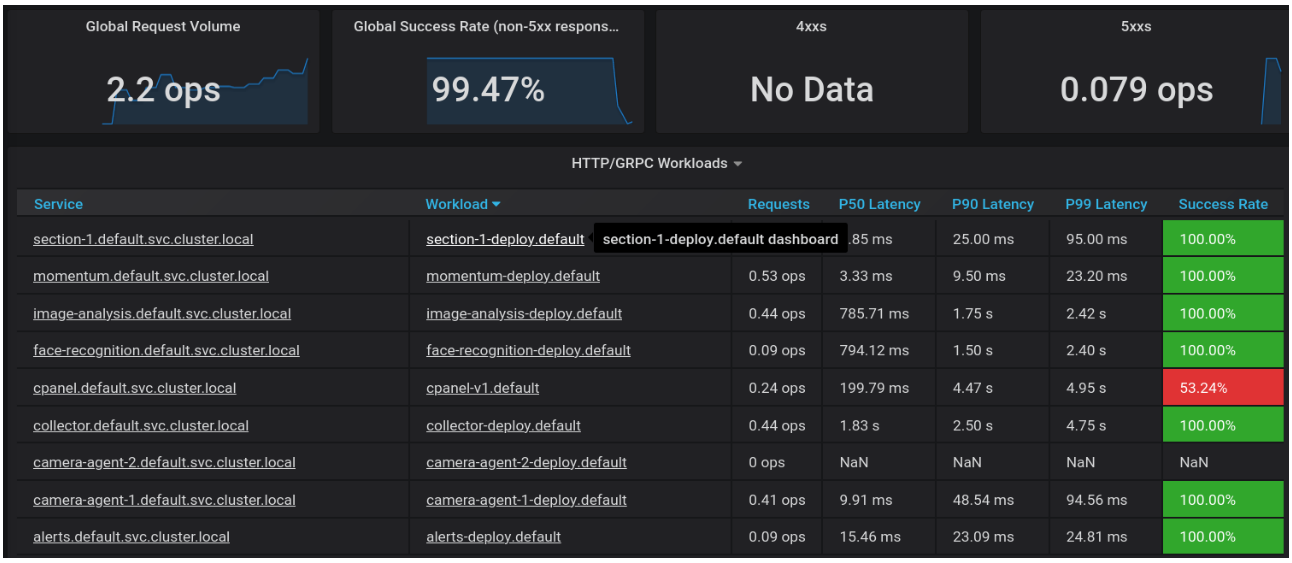
But more interesting and sophisticated real world scenario is introduced when using fault injection mechanisms of istio together with such resiliency features for network communication as timeouts and retries of failed requests.

Timeout

In order to check how timeout mechanism works fixed delay 10 seconds for camera-agent-1 with success rate 50% was configured in virtual service. So after applying this configuration every second request to camera agent will be delayed. To protect client from waiting to long (full 10 seconds) timeout of 3 seconds is configured in virtual service. In such a way user will get response on request 3 times faster though it will be not positive.

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

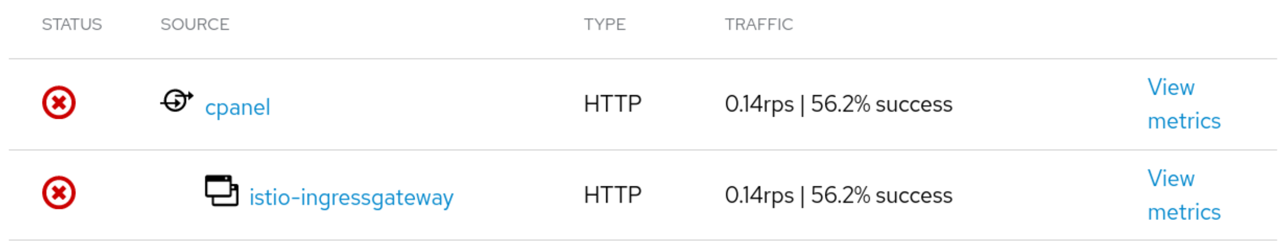
```
$ make health-timeout
for i in {1..100}; do sleep 0.2; curl http://192.168.99.114:32460/cameras/1/state; printf "\n"; done
{"streaming":true,"cycle":42,"fps":0,"section":"1","destination":"exit","event":"exit"}
{"streaming":true,"cycle":42,"fps":0,"section":"1","destination":"exit","event":"exit"}
{"streaming":true,"cycle":43,"fps":0,"section":"1","destination":"exit","event":"exit"}
{"streaming":true,"cycle":43,"fps":0,"section":"1","destination":"exit","event":"exit"}
upstream request timeout
upstream request timeout
{"streaming":true,"cycle":44,"fps":0,"section":"1","destination":"exit","event":"exit"}
upstream request timeout
{"streaming":true,"cycle":45,"fps":0,"section":"1","destination":"exit","event":"exit"}
upstream request timeout
upstream request timeout
upstream request timeout
```



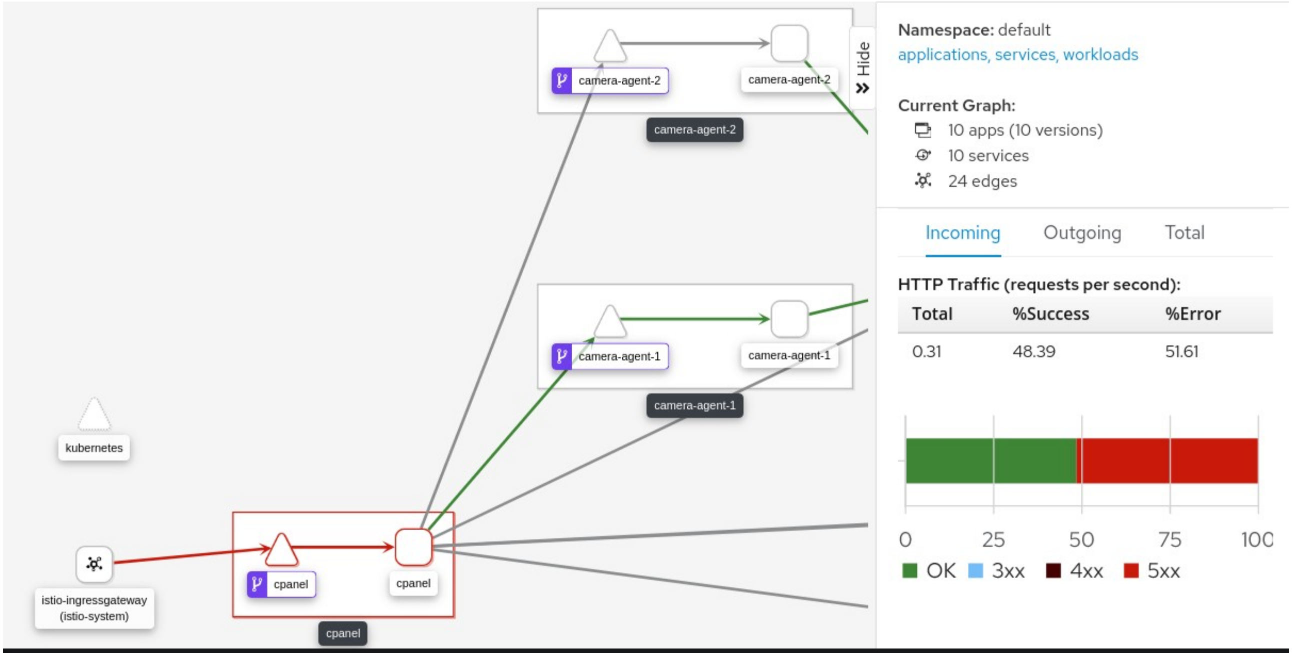
Global Request Volume	Global Success Rate (non-5xx respons...	4xxs	5xxs
2.2 ops	99.47%	No Data	0.079 ops

HTTP/GRPC Workloads

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default section-1-deploy.default dashboard		.85 ms	25.00 ms	95.00 ms	100.00%
momentum.default.svc.cluster.local	momentum-deploy.default	0.53 ops	3.33 ms	9.50 ms	23.20 ms	100.00%
image-analysis.default.svc.cluster.local	image-analysis-deploy.default	0.44 ops	785.71 ms	1.75 s	2.42 s	100.00%
face-recognition.default.svc.cluster.local	face-recognition-deploy.default	0.09 ops	794.12 ms	1.50 s	2.40 s	100.00%
cpanel.default.svc.cluster.local	cpanel-v1.default	0.24 ops	199.79 ms	4.47 s	4.95 s	53.24%
collector.default.svc.cluster.local	collector-deploy.default	0.44 ops	1.83 s	2.50 s	4.75 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	0.41 ops	9.91 ms	48.54 ms	94.56 ms	100.00%
alerts.default.svc.cluster.local	alerts-deploy.default	0.09 ops	15.46 ms	23.09 ms	24.81 ms	100.00%



STATUS	SOURCE	TYPE	TRAFFIC	
☒	cpanel	HTTP	0.14rps | 56.2% success	View metrics
☒	istio-ingressgateway	HTTP	0.14rps | 56.2% success	View metrics



Namespace: default
applications, services, workloads

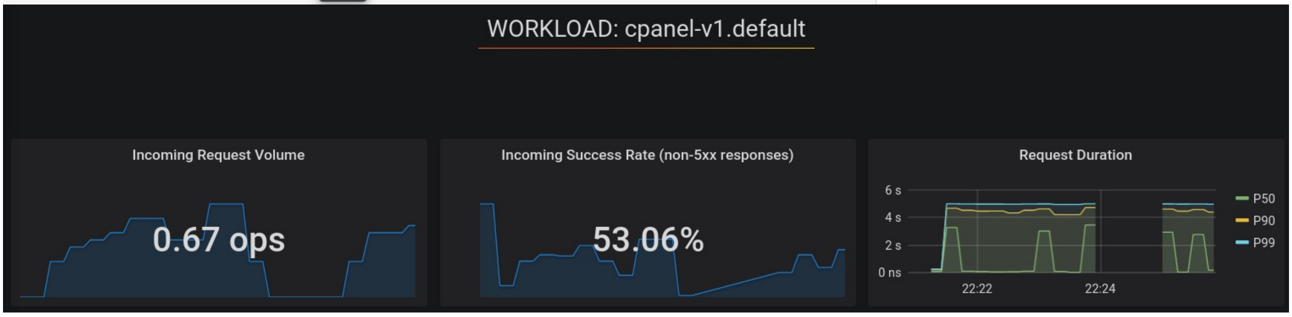
Current Graph:

10 apps (10 versions)
10 services
24 edges

Incoming Outgoing Total

HTTP Traffic (requests per second):

Total	%Success	%Error
0.31	48.39	51.61



WORKLOAD: cpanel-v1.default

Incoming Request Volume	Incoming Success Rate (non-5xx responses)	Request Duration
0.67 ops	53.06%	

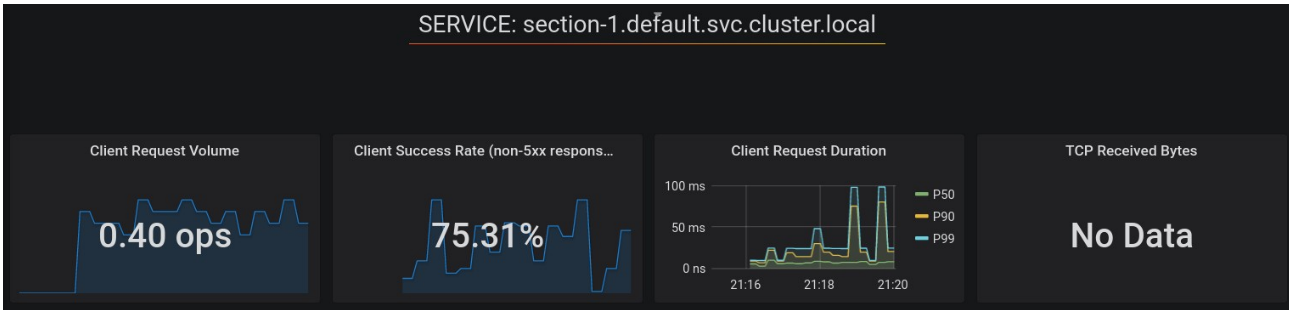
$ make deploy-istio-default

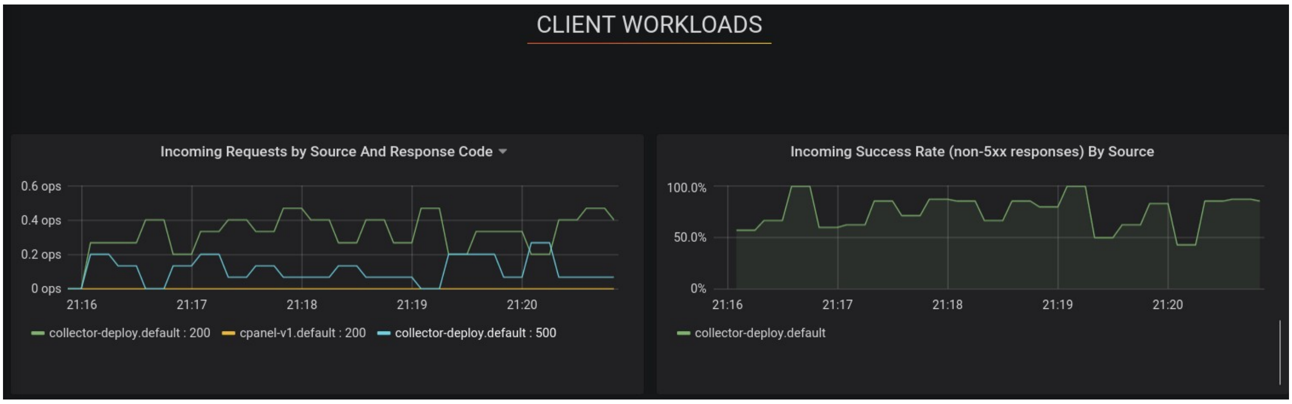
Retries

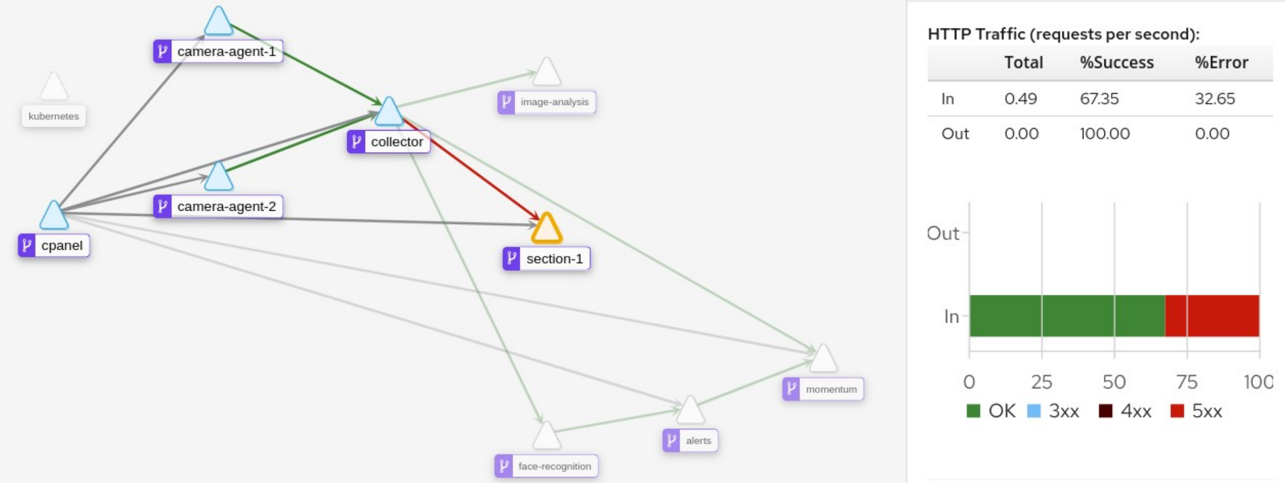
To demostrate retry pattern in istio fault-injection was made in virtual service section-1 so that every fourth request will response with 500 HttpError. As a result collector will have failed requests and the data can be lost. To tolerate this artificial error rate retries were configured in collector virtual service. With 3 retries versus 25% error rate the system should behave much more stable.

```
$ make retries-fault
./kubectl apply -f istio/retry_fault.yaml
```

```
$ make start-cameras
curl http://192.168.99.114:32460/production?toggle=on
```




CLIENT WORKLOADS
Incoming Requests by Source And Response Code
0.6 ops
0.4 ops
0.2 ops
0 ops
21:16
21:17
21:18
21:19
21:20
collector-deploy.default : 200
cpanel-v1.default : 200
collector-deploy.default : 500
Incoming Success Rate (non-5xx responses) By Source
100.0%
50.0%
0%
21:16
21:17
21:18
21:19
21:20
collector-deploy.default



HTTP Traffic (requests per second):

	Total	%Success	%Error
In	0.49	67.35	32.65
Out	0.00	100.00	0.00

```
$ make health-retries
for i in {1..100}; do sleep 0.2; curl http://192.168.99.114:32460/sections/1/status; printf "\n"; done
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
fault filter abort
Section 1 v1 : Online
fault filter abort
```

```
make retries
./kubectl apply -f istio/retry.yaml
```

```
$ make health-retries
for i in {1..100}; do sleep 0.2; curl http://192.168.99.114:32460/sections/1/status; printf "\n"; done
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
Section 1 v1 : Online
```

Circuit breaker

Outlier detection

```
$ make outlier
./kubectl apply -f istio/outlier_detection_collector.yaml
```

```
$ make outlier-scale
./kubectl scale deployment collector-deploy --replicas=3
```

```
$ make health-outlier
for i in {1..100}; do sleep 0.2; curl http://192.168.99.114:32460/collector/status; printf "\n"; done
Collector v1 : Online - collector-deploy-69c878f4b7-mdqcx
Collector v1 : Online - collector-deploy-69c878f4b7-jg5l4
Collector v1 : Online - collector-deploy-69c878f4b7-jg5l4
Collector v1 : Online - collector-deploy-69c878f4b7-9h8b9
Collector v1 : Online - collector-deploy-69c878f4b7-mdqcx
Collector v1 : Online - collector-deploy-69c878f4b7-jg5l4
Collector v1 : Online - collector-deploy-69c878f4b7-9h8b9
```

```
$ make outlier-fault
./kubectl exec -it collector-deploy-69c878f4b7-9h8b9 -c collector http localhost:8080/fault
HTTP/1.0 200 OK
Content-Length: 10
Content-Type: text/html; charset=utf-8
Date: Wed, 04 Mar 2020 13:52:24 GMT
Server: Werkzeug/1.0.0 Python/3.7.6
Now faulty
```

```
$ make health-outlier
for i in {1..100}; do sleep 0.2; curl http://192.168.99.114:32460/collector/status; printf "\n"; done
Collector v1 : Online - collector-deploy-69c878f4b7-mdqcx
Collector v1 : Online - collector-deploy-69c878f4b7-jg5l4
Collector v1 : Online - collector-deploy-69c878f4b7-jg5l4
Collector v1 : Online - collector-deploy-69c878f4b7-mdqcx
Collector v1 : Online - collector-deploy-69c878f4b7-jg5l4
Collector v1 : Online - collector-deploy-69c878f4b7-mdqcx
Collector v1 : Online – collector-deploy-69c878f4b7-jg5l4
```

So we can see that faulty pod is extracted from load balancing pool and no traffic is forwarded to it.

Connection pool

```
$ make deploy-fortio
./deploy_fortio.sh
service/fortio created
deployment.apps/fortio-deploy created
fortio pod:  fortio-deploy-68c7549cc6-qc2lj
get response from collector

HTTP/1.1 200 OK
content-type: text/plain; charset=utf-8
content-length: 57
server: envoy
date: Wed, 04 Mar 2020 14:46:40 GMT
x-envoy-upstream-service-time: 5
```

```
$ make circuit-breaker
./kubectl apply -f istio/circuit_breaker.yaml
```

```
$ make load-fortio
./load_fortio.sh
fortio pod:  fortio-deploy-68c7549cc6-qc2lj
generating load to cpanel
15:20:33 I logger.go:97> Log level is now 3 Warning (was 2 Info)
Fortio 1.3.1 running at 0 queries per second, 4->4 procs, for 20 calls: http://collector:8080/status
Starting at max qps with 3 thread(s) [gomax 4] for exactly 20 calls (6 per thread + 2)
15:20:33 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)
15:20:33 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)
15:20:33 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)
15:20:33 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)
```

```
15:20:33 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)
15:20:33 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)
15:20:33 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)
Ended after 249.209801ms : 20 calls. qps=80.254
Aggregated Function Time : count 20 avg 0.02127085 +/- 0.02412 min 0.000701932 max
0.079165391 sum 0.425416991
# range, mid point, percentile, count
>= 0.000701932 <= 0.001 , 0.000850966 , 5.00, 1
> 0.001 <= 0.002 , 0.0015 , 25.00, 4
> 0.002 <= 0.003 , 0.0025 , 30.00, 1
> 0.007 <= 0.008 , 0.0075 , 40.00, 2
> 0.008 <= 0.009 , 0.0085 , 45.00, 1
> 0.009 <= 0.01 , 0.0095 , 50.00, 1
> 0.012 <= 0.014 , 0.013 , 55.00, 1
> 0.018 <= 0.02 , 0.019 , 65.00, 2
> 0.02 <= 0.025 , 0.0225 , 70.00, 1
> 0.025 <= 0.03 , 0.0275 , 80.00, 2
> 0.03 <= 0.035 , 0.0325 , 85.00, 1
> 0.06 <= 0.07 , 0.065 , 90.00, 1
> 0.07 <= 0.0791654 , 0.0745827 , 100.00, 2
# target 50% 0.01
# target 75% 0.0275
# target 90% 0.07
# target 99% 0.0782489
# target 99.9% 0.0790737
Sockets used: 9 (for perfect keepalive, would be 3)
Code 200 : 13 (65.0 %)
Code 503 : 7 (35.0 %)
Response Header Sizes : count 20 avg 108.3 +/- 79.47 min 0 max 167 sum 2166
Response Body/Total Sizes : count 20 avg 229.7 +/- 8.301 min 223 max 241 sum 4594
All done 20 calls (plus 0 warmup) 21.271 ms avg, 80.3 qps

$ make get-fortio
./kubectl exec fortio-deploy-68c7549cc6-qc2lj -c istio-proxy -- pilot-agent request GET stats | grep
collector | grep pending
cluster.outbound|8080|v1|
collector.default.svc.cluster.local.circuit_breakers.default.rq_pending_open: 0
cluster.outbound|8080|v1|collector.default.svc.cluster.local.circuit_breakers.high.rq_pending_open:
0
cluster.outbound|8080|v1|collector.default.svc.cluster.local.upstream_rq_pending_active: 0
cluster.outbound|8080|v1|collector.default.svc.cluster.local.upstream_rq_pending_failure_eject: 0
cluster.outbound|8080|v1|collector.default.svc.cluster.local.upstream_rq_pending_overflow: 7
cluster.outbound|8080|v1|collector.default.svc.cluster.local.upstream_rq_pending_total: 22
```

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 $$$

Not everyone will need istio right now. Complexity

debugging too complex. Many moving parts of istio, kubernetes, application, envoy deployments

1200 open issues in github

not really production ready

Future Work

Each new version of istio has plenty of bugs fixed. All of them influence on your the stability and resiliency of your deployment. It is up to you to test them, give feedback and become a part of new standard for future deployments of your microservices applications.

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
14. (mesh)https://servicemesh.io/
15. (micro_git)https://github.com/davidetaibi/Microservices_Project_List
16. (enterprise)Microservices for the Enterprise
17. (advanced)Advanced Microservices. A Hands-on Approach to Microservice Infrastructure and Tooling
18. (microcon)Microservices and Containers
19. (meshpath)The Enterprise Path to Service Mesh Architectures
20. (appmicro)Applications and microservices with docker and containers. The new stack
21. (prodmicro)Production-Ready Microservices Building Standardized Systems Across an Engineering Organization Susan J. Fowler

22. (designmicro)Microservices From Design to Deployment by Chris Richardson with Floyd Smith
23. (buildmicro)Building Microservices by Sam Newman
24. (eval)Villamizar, Mario & Garcés, Oscar & Castro, Harold & Verano Merino, Mauricio & Salamanca, Lorena & Casallas, Rubby & Gil, Santiago. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. 10.1109/ColumbianCC.2015.7333476.
25. (uprun)Istio: up and running
26. (10years) N. Kratzke and P.-C. Quint. Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. Journal of Systems and Software, 126:1–16, 2017.
27. (flexible)E. Wolff. Microservices: Flexible Software Architectures. CreateSpace Independent Publishing Platform, 2016.
28. (migrate)A. Balalaie, A. Heydarnoori, and P. Jamshidi. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report, pages 201–215. Springer International Publishing, Cham, 2016.
29. (today)N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, Today, and Tomorrow, pages 195–216. Springer International Publishing, Cham, 2017.
30. (recovery)G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pages 298–302, April 2017.
31. (decision)S. Haselbock and R. Weinreich. Decision guidance models for microservice monitoring. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pages 54–61, April 2017.

Supplemental Material

- cc assignment
- commands