Abstract

The expanse of cloud technologies and movement to PaaS brings new challenges for developers. To stay efficient and to utilize most of the cloud features modern applications should be scalable, resilient and fast as in developing, so in testing and deploying to production. One of the solutions is migrating from monoliths to microservices architecture or start to use cloud-native development patterns for completely new projects.

The splitting of monolithic application in several microservices introduces new challenges in software engineering processes. Extremely radical changes need to be done in operations departments to monitor, scale and deliver resilient workflow in the hole software life cycle. One of the most critical things to consider when running a complex distributed application is resiliency. In this thesis service mesh Istio[istio] running on top of the Kubernetes[k8s] cluster will be introduced as a solution to provide visibility, control, security and fault tolerance to your deployments. The final goal is to demonstrate the possibilities of Istio and to try out the resiliency features on the microservices application.

Keywords

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

Table of Contents

Abstract....1
Keywords....1
Introduction....3
Related work....3
Major idea....4
- Microservices....4
- Service mesh....7
- Istio....8
- Resiliency....10
- Demo....13

Implementation....13
- The Twelve Factors Application....13
- Deploy with Kubernetes....15
- Deploy with Istio....16
- How to run....16

Evaluation....16
- Routing....19
- Load balancing....22
- Fault injection....23
- Timeout....24
- Retries....25
- Circuit breaker....27
- Discussion....29

Conclusion....29
Future Work....30
References....30
Supplemental Material....31

Introduction

The time of slow development cycles, deployments and support is gone. Users want to interact with services fast and without downtime. Cloud platforms have introduced a new advanced way to rapidly deliver results to clients. Migration to clouds also brought new challenges. Big monolithic applications were inefficient in scaling to custom loads [eval]. This lead to rethinking of the architecture of monolithic applications. Instead of packaging everything in one big project the idea with many independently developed and communicating with one another over network microservices came up.

Transition to microservices architecture helped to make application deployments more cloud friendly and made the fast code-to-market strategy possible. Automation, scalability and continuous delivery are among the most valuable attributes coming with this architectural changes in software engineering process [10years]. All these factors and independence between microservices brought application resiliency on completely new level [migrate].

Moving out from using virtual machines for deploying applications and adoption of containers and automated deployments changed the scene one more time [10years]. Containers are more lightweight and blazing fast in startup in compare to virtual machines. The problem of delivering code from developers to production environment is solved here by packaging application and dependencies in images that run everywhere the same way.

Proper and efficient deployment strategies are crucial for microservices. Kubernetes container orchestration system provides all needed functions for management of microservice applications. These includes secret management, service discovery, horizontal scalability[action]. One of the problems is that it has no way to deal with network errors.

As the number of microservices grows developers and operations engineers lost the visibility of the deployment, control of communication inside the application. In this way the overall availability of the service is falling. That is why resiliency of microservices application is very important. The failures take place on different levels: network, DNS, timeouts, internal exceptions [action]. Though it is almost impossible to eliminate all the failures, it is possible to tolerate them and to recover from them to maximize availability of the application.

There are different approaches to overcome these challenges and one of them is to use service meshes to get full control over your microservices. The most valuable feature here is that very few changes or not at all should be added to the code of microservices. This also allows developers to focus only on business logic of the application. Istio service mesh offers a complete solution to solve the complexity of distributed microservices applications [istio].

In this work microservices application will be deployed in Kubernetes cluster with already installed Istio. The application itself was developed in cloud computing course, but was refactored and adopted to make the demonstration of Istio resiliency possibilities more visible. The resiliency of the deployment will be tested with load simulating and chaos testing. As a result of experiments - installation scripts and configuration files, graphics and console outputs of application behavior with and without Istio will be introduced.

The thesis has following structure. In the first chapter different alternatives of service mesh architectures are discussed. Major idea, the theory about microservices, service meshes, Istio and resiliency are introduced in second chapter. Then the details of the implentation are described in the third chapter. Tests and the evaluation of the results is done in the last part.

Related work

There is already a strong need for service meshes for modern microservices applications. Many companies try to occupy this niche by developing their own implementations of service meshes solutions. So today with a big demand in getting observability and control over deployments there are also solutions with completely different architectures on the market. Most

relevant issues that are covered by these architectures are security, tracing, observability, fault tolerance, fault injection, advanced routing.

Libraries represent the most traditional way to add additional functionality to the application. Examples of such implementation are Hystrix and Ribbon from Netflix [alt]. These libraries are used to get rid of network faults and not to implement code for communication inside application, but these should be developed and be up to date for each programming language in software stack of the company. This approach is not really useful with microservice architecture because abuses polyglot idea of microservices. Also violates a principle of separation of business logic and communication and many changes in the code of microservices should be made.

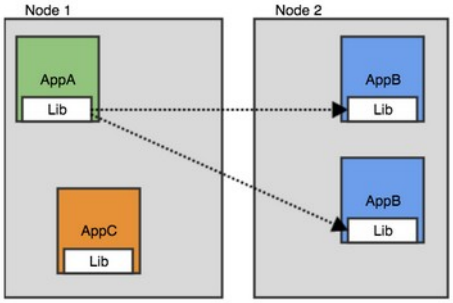


Figure 1: Service mesh based on libraries [alt]

○

Node agent is the second way to deploy service mesh. The idea here is to deploy a proxy agent on each node of cluster, the same way Kubernetes has kubelet on each node for registration purposes and to manage the pods [k8s]. An example of this type of architecture for service meshes is Linkerd [alt]. As a disadvantage of this method we can mention the existence of a one point of failure – node proxy. One failure in proxy will influence all the microservices deployed on this node. On the other hand this approach is more resource efficient [linkerd].

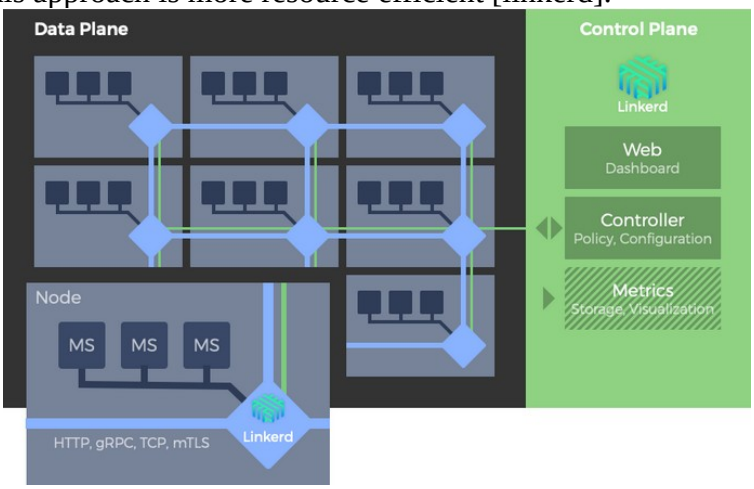


Figure 2: Linkerd architecture with node agents [linkerd]

Sidecar proxy architectural pattern introduces another approach of integrating proxies in application deployment. In this kind of service mesh sidecars are inserted along each container so that each microservice pod has two containers inside: proxy and microservice itself. Examples of such architecture are Istio, Linkerd2, Consul. More details about this approach is covered in Istio chapter.

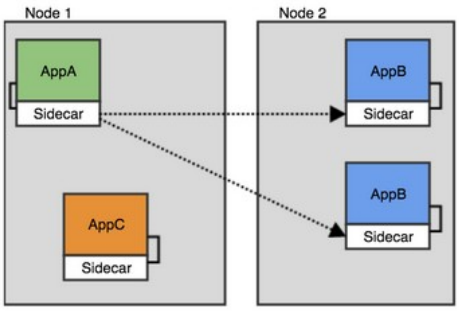


Figure 3: Service mesh based on sidecar proxies [alt]

Major idea

There are plenty of tutorials online that utilize a sample application from Istio web site ("Bookinfo" application [istio]) to show typical service mesh and specific Istio features. The idea of this thesis is to take 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 a random application along with Istio service mesh.

The web application itself must be based on microservices architecture. Trying to make focus on operational part of software engineering process and not to focus on developing the service from scratch one of the solutions was to take a ready open source project from Github and deploy it with Istio [microgit].

After researching and trying out some of these projects the decision to take the application developed by myself in cloud computing course was made [cc]. The text of the original assignment can be found in supplemental material. The application will be deployed in Kubernetes cluster with preinstalled Istio and configured sidecars auto injection for each pod. As Istio has plenty of resiliency functionalities they will be tried out one by one to provide a better overview of the results and also to minimize debugging time of possible deployment problems.

Some changes and additions were made to the original code of the web application. The initial commit in Github repository shows the start point of the project implementation. There you will find also Minikube and Istio installation scripts along with other developing environment scripts.

Microservices

Migrating from monolithic applications to microservices is a challenge on itself [today]. There some reasons why one would like to completely change the design of the production ready application. One of the reasons is that updating cycle of the monolithic application is extremely slow. The work should be synchronized between different teams, that develop different modules of the application and at the end the functionality should not be lost [eval]. The other reason is scalability. Monolithic application is just not efficient at scaling and can not provide necessary velocity on load from client. As a result we get unsatisfied users that costs the company much money.

Microservice is an architectural pattern to split big monolithic application in smaller independent services communicating with each other (often by means HTTP requests). Each microservice is built around one small business logic. This architecture takes maximum from the modern deployment automation facilities [fowler].

So by using only one service for one task without any shared libraries and dependencies a decent separation between business logic implementation can be achieved. This is opens the road to horizontal scalability on purpose (eg. high load on Christmas period).

The idea of major microservices attributes can be compared with UNIX ideas [flexible]:

- one program – one task
- universal interface for all programs – exposed APIs

- programs communicate with one another – synchronous and asynchronous

Microservices are small, independently scaled and managed applications. Each of them has its own unique and well-defined role, runs in its own process and communicates via HTTP protocol APIs or messaging [native]. Ideally one developer should be enough to understand the idea of one special microservice and maintain it [10years].

Designing of microservice system need different tools and processes: application itself, infrastructure with virtualization for hosting, monitoring and logging for all communication, organizational structures (teams), development processes (continuous integration), deployment (continuous delivery), testing [decision]. As it is very difficult to reproduce errors in big distributed applications - logging is so important [prodmicro].

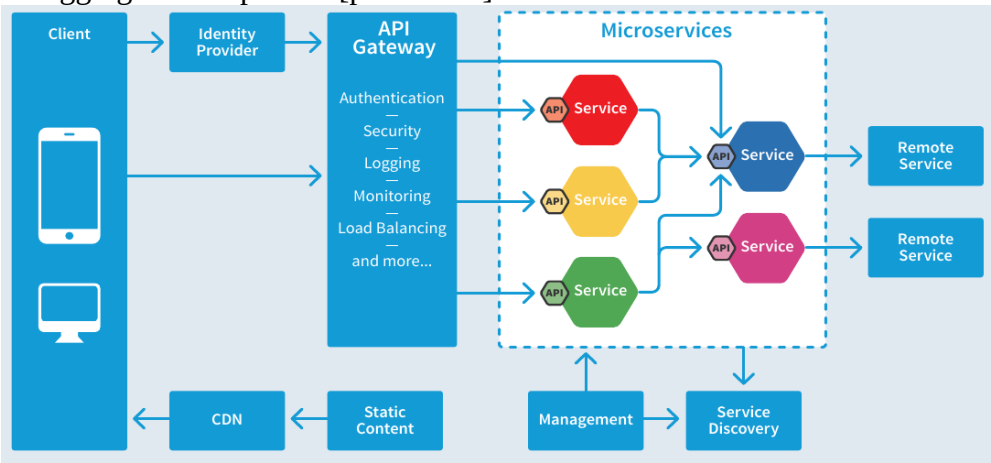


Figure 4: Web application with microservice architecture [native]

Each microservice is similar internally to monolithic application. So it has not only the code but is a full featured normal web application [action]:

- application code or runnable program
- libraries
- processes (eg. cron)
- data stores, load balancers, or other services

If we have many microservices in our fleet comes up the question what is the best way to package them and deliver from developers to testers and from testers to production. Immutable images and containers solve this problem [docker]. Containers run applications that are packaged in images, virtual machines run containers [advanced]. These containers are executable artifacts that allow to manage deployment by simple adding or removing a container from the current deployment [action]. But together with a scheduler, containers provide a particularly elegant and flexible approach that meets our two deployment goals: speed and automation.

Containers are extremely fast in start up because of shared kernel with host operating system. this can be a decent security issue. If one of the containers is compromised so are all the others. Virtual machines have much better isolation, but are resource heavy, because of independent kernel running in each machine [advanced].

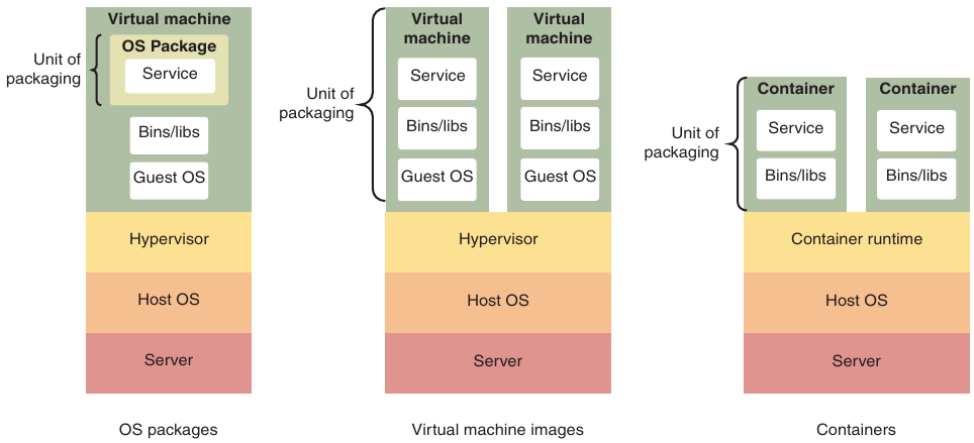


Figure 5: Comparison of packaging solutions [action]

Microservices architecture shows that containers dominate today on the market because allowing to package the application and its dependencies makes it easier to run a polyglot stack. On the other hand Kubernetes orchestration platform solves the next problem: managing, scaling and automating the deployment of the microservices across the cluster of worker nodes [mesh].

With the benefits of microservices mentioned above they bring a high complexity to deployment and maintenance of the running web application [appmicro]:

- increased operational complexity [towards]
- secure communication
- partitioned database architecture [designmicro]
- unreliable communication [flexible]
- resiliency issues – cascade failures in distributed systems
- complicated transition from monolith to microservice [towards]
- service discovery
- testing the complete system [designmicro]
- faulty on the integration level [today]

To not to loose the control and the overview of your microservice application is where service meshes come in play.

Service mesh

Modern web applications have very strict requirements such as availability (zero downtime) or fast respond to requests [java]. Having a number of microservices in your deployment makes the maintenance really challenging. Operators should manage microservices applications in large hybrid and multi-cloud deployments. With demand to get more control and observability inside the network of running microservices the concept of service meshes appeared. Some of the current solutions to this concept were already discussed in related work.

Service mesh gives opportunity to get full control over your microservices network in a uniform way and decoupled form the application code [mesh]. It focuses on networking between microservices (east-west traffic) rather than business logic of the web application. Service mesh provides out of the box plenty of features that are now implemented in different separately managed ways: libraries for logging, API gateways for routing, certificates rotation for secure communication.

Service mesh can provide following functionalities depending on the implementation: service discovery, load balancing, resiliency and failure recovery, security (end-to-end encryption, authorization), observability (layer 7 metrics, tracing, alerting, logging), routing control (A/B testing, canary deployments), API (programmable interface, Kubernetes CRD).

The most promising architecture of service meshes is based on sidecar proxy injections and works on top of Kubernetes cluster. Proxies handle all incoming and outgoing microservice traffic. Focus on the traffic between services is the difference between service mesh proxies and API

gateways or ingress proxies, which focus on requests from the outside network into the cluster [mesh].

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

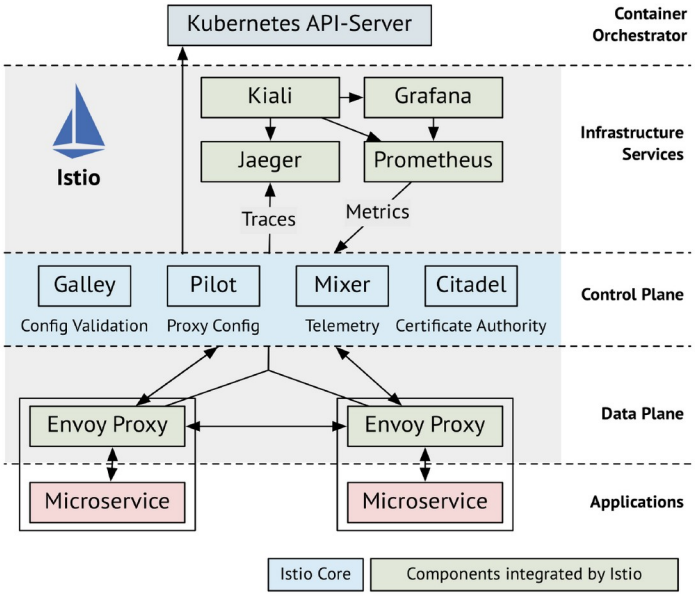
puts resiliency into the infrastructure
An Istio service mesh is logically split into a **data plane** and a **control plane**.

- 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
You can use Grafana to graph metric data, create human-readable dashboards, and trigger alerts[action].
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

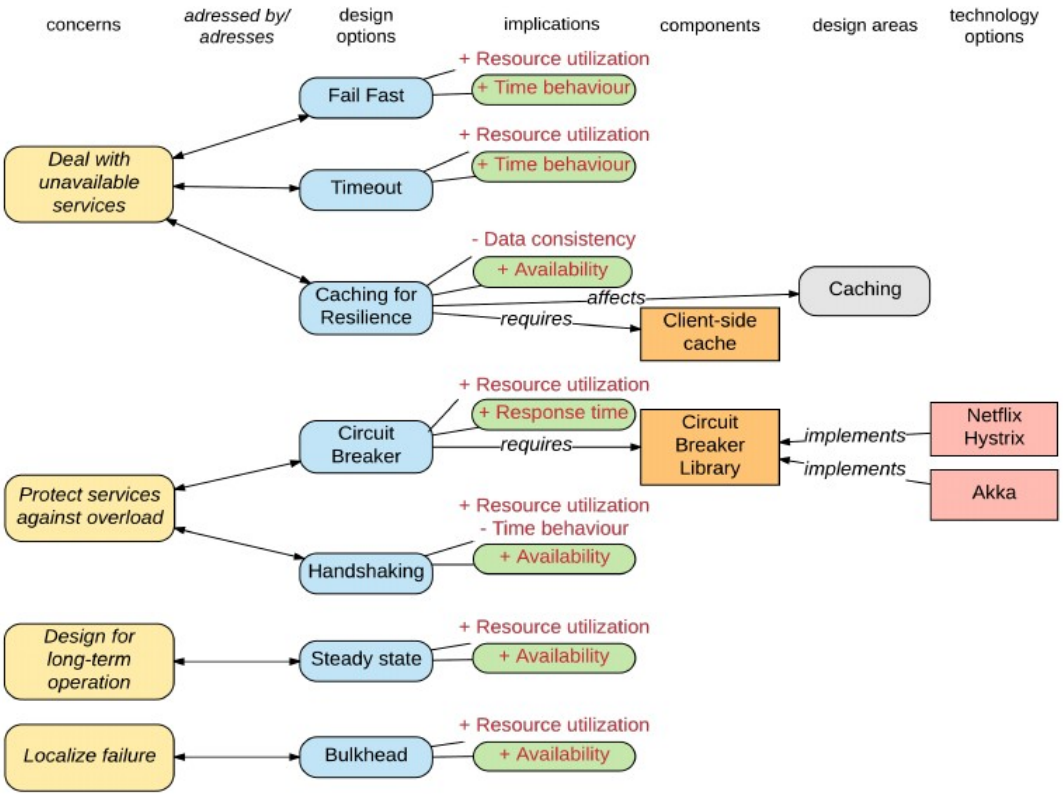
- higher latency
- resource hungry – x2 containers
- young technology

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, data center failure, poor architecture, lack of unit tests, communication over the unreliable network, dependent services etc.

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.

Services can use rate limits to protect themselves from spikes in load beyond their capacity to service [action].



[decision]

Similarly, dedicating a large number of resources (like CPU) to a microservice that doesn't utilize it is inefficient. Inefficiency reduces performance: if it's not clear at the microservice level in every case, it's painful and costly at the ecosystem level. Underutilized hardware resources affects the bottom line, and hardware is not cheap [prodmicro].

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.

A resilient microservice is one that can experience and recover from failures at every level of the microservice ecosystem: the hardware layer (e.g., a host or datacenter failure), the communication layer (e.g., RPC failures), the application layer (e.g., a failure in the deployment pipeline), and in the microservice layer (e.g., failure of a dependency, a bad deployment, or a sudden increase in traffic). There are several types of resiliency testing that, when used to evaluate the fault tolerance of a microservice, can ensure that the service is prepared for any known failures within any layer of the stack. [prodmicro]

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.
Probes can be HTTP GET requests, scripts executed inside a container, or TCP socket checks.

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

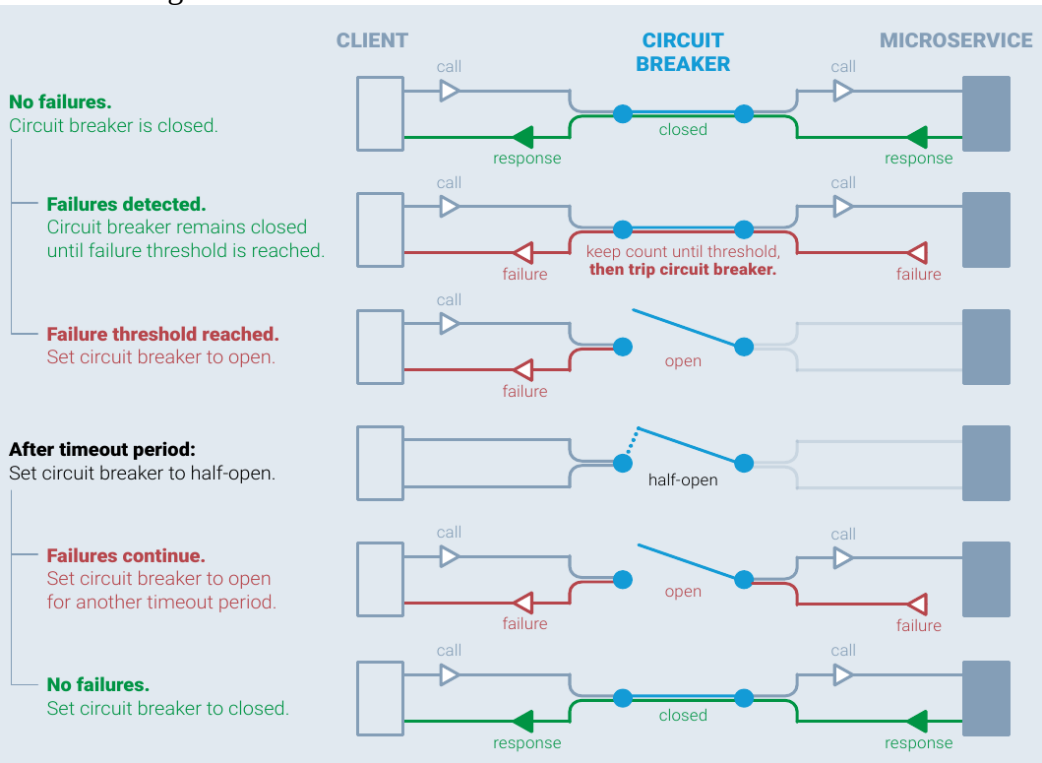
length of timeout. So infrastructure engineer need to understand how the microservices application work or need to communicate with developers direct.

```
        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 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
```

Demonstration

The main result of this thesis will be a fully working demo to show the main resiliency 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. It is deployed in single node Kubernetes cluster that runs in Minikube with Virtualbox provider. Git repository contains all necessary scripts to install and start using Kubernetes with Istio in development environment [git].

With the help of this demo you can 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.

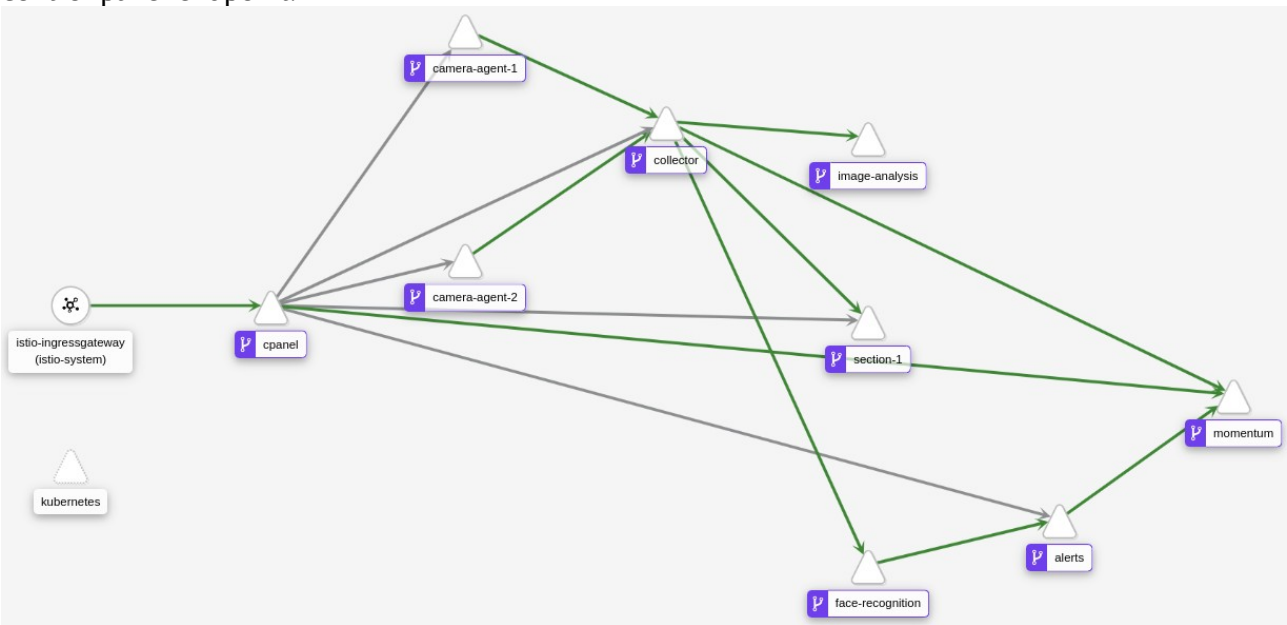
Implementation

The Twelve Factors Application

Application itself is a simulation of airport security system. REST api [rest].
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.

Figure : Sequence diagram

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.

Cpanel takes a role of API gateway. It hides backend microservices from the client and exposes only necessary endpoints [microcon]. Request routing. Is similar to facade pattern in object oriented design [designmicro].

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[java], user resiliency
- python + docker best practices:
  - alpine, root, no cache [sec]
- 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

Load balancing is done on top of related microservices for number of replicas [microcon].
Replication of pods is configured for collector, image analysis and face recognition.
Native server side service discovery of kubernetes is used to configure communication between microservices [microcon].
Service discovery is abstrated from client. They need to make the request just to load balancer.
Service regustry is built-in in kubernetes architecture. [designmicro].
Services provide stable endpoints for Pods and are automatically registered in service registry. if pod restart - new IP.
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).
In Kubernetes, a service defines a set of pods and provides a method for reaching them, either by other applications in the cluster or from outside the cluster [action].

Deployments on Kubernetes are designed to maintain service availability by executing rolling updates of pods across replica sets[action].

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.

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.

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.

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

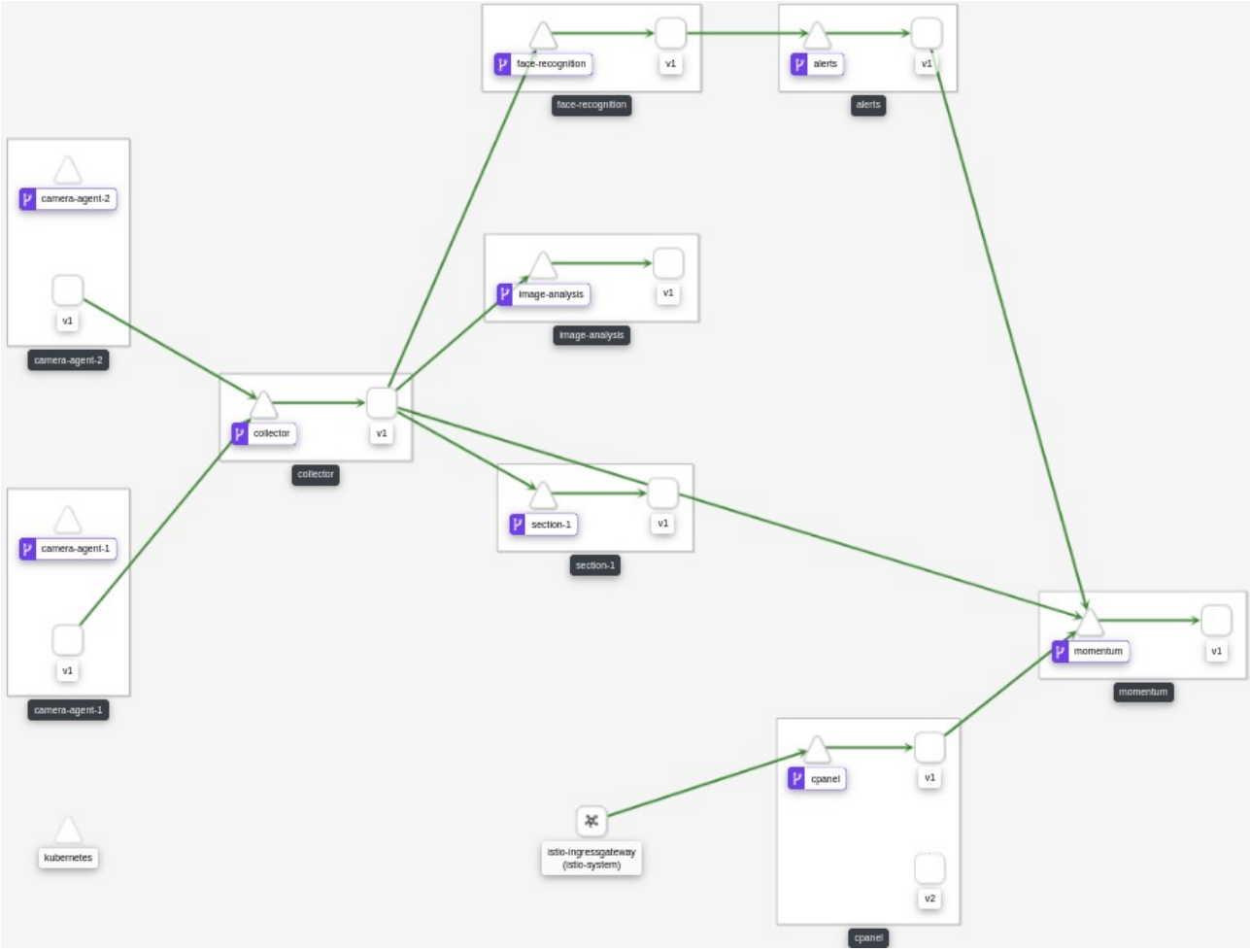
Wait till all pods are up and running and stop monitoring them with Ctrl-c.

$ 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

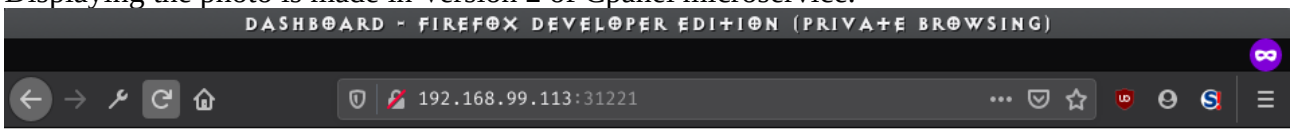
Check that application is deployed properly with istio configuration files.

$ 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

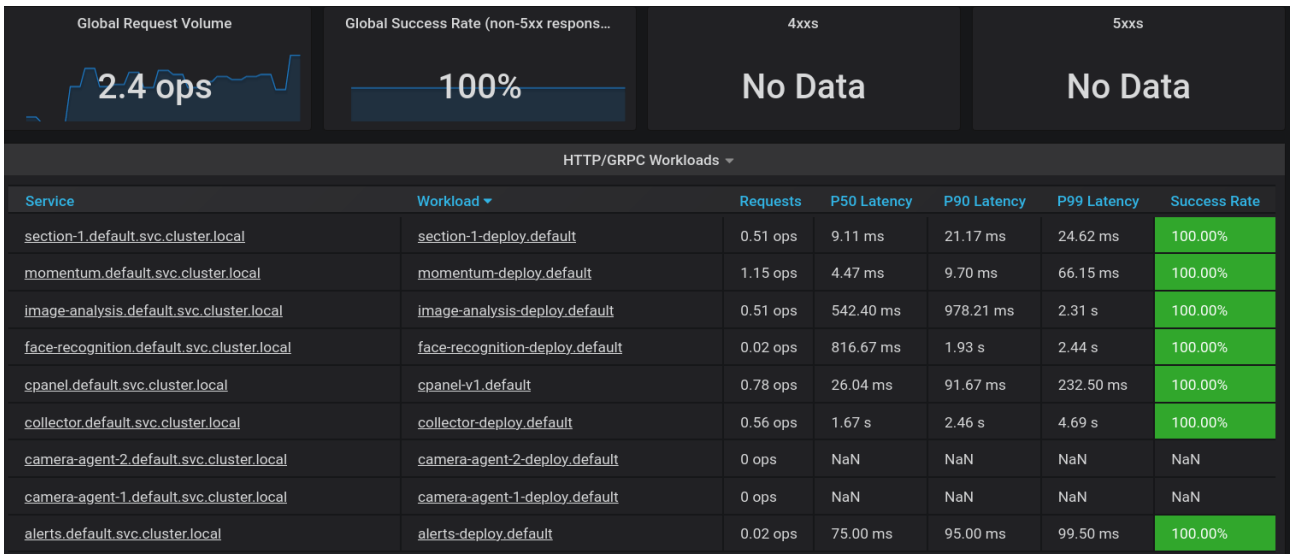
timestamp: 2020-02-25T14:35:27.224857Z

section: 1

event: entry

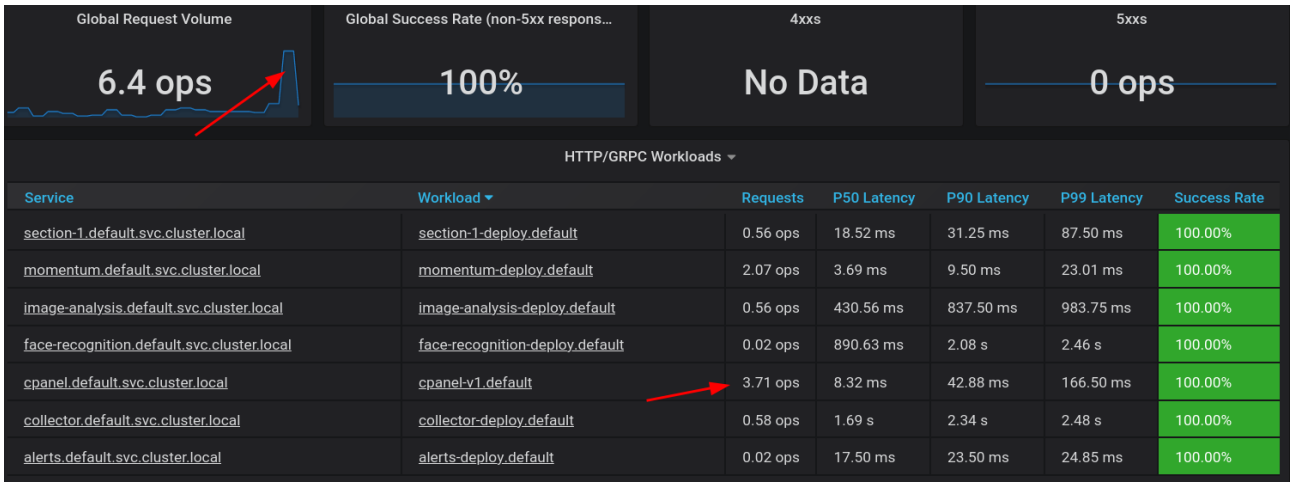
name: **PersonX**

Default app with Cpanel v1 without load to frontend:



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
…



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[java].

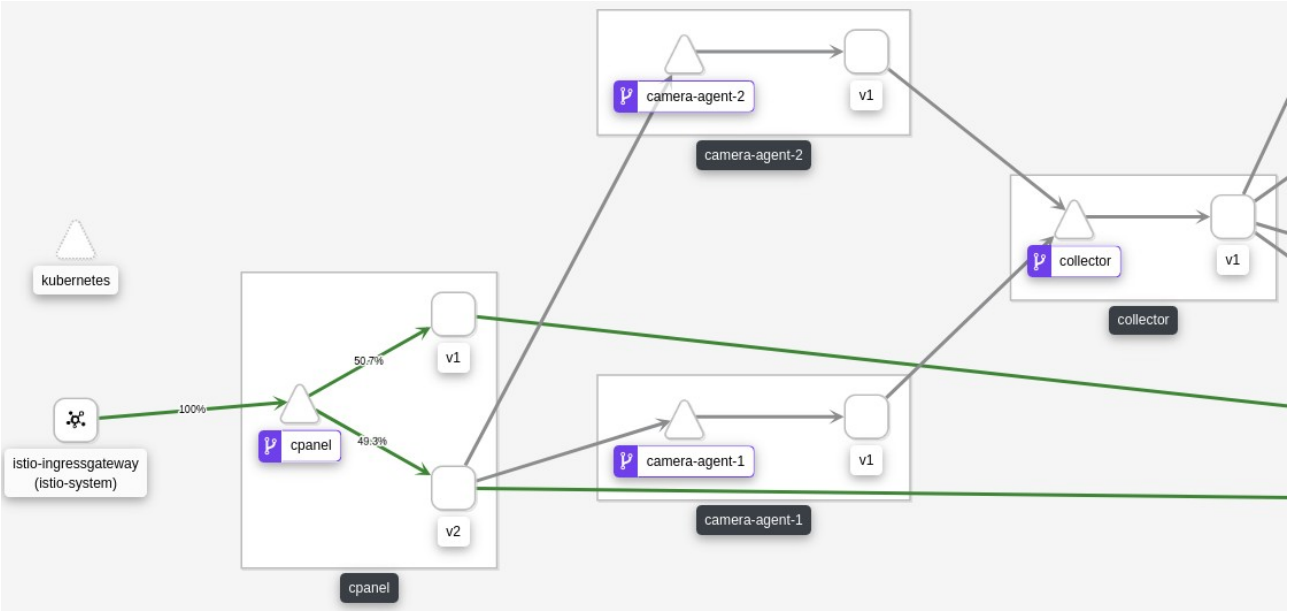
route:

```
- 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>



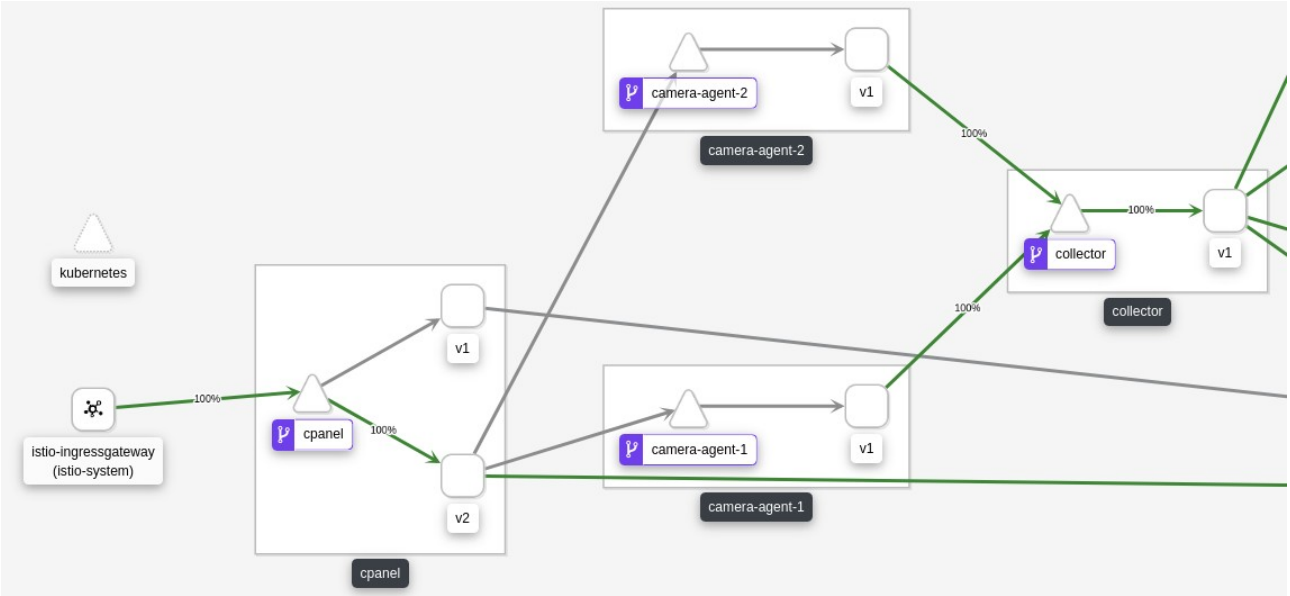

```
route:
- destination:
    host: cpanel.default.svc.cluster.local
    port:
```

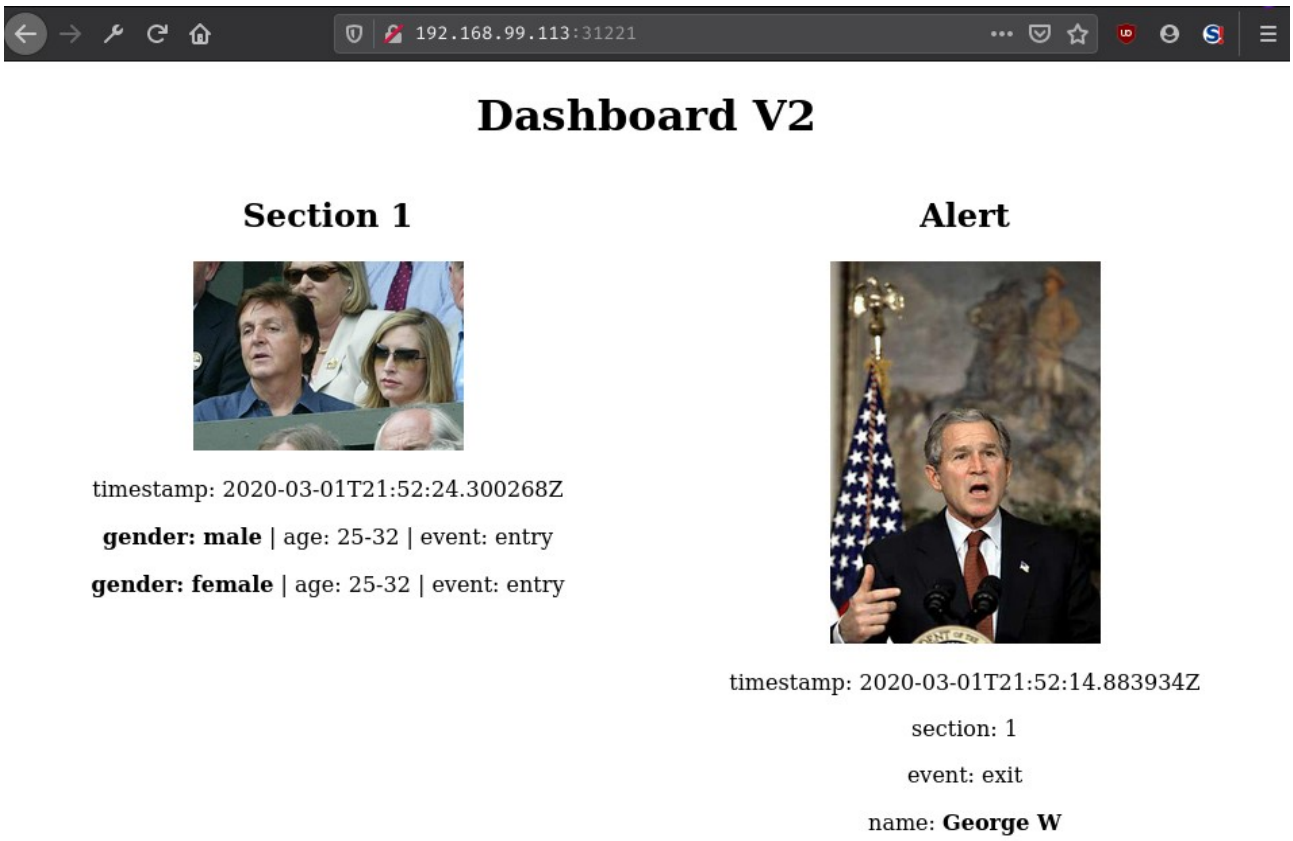
```
      number: 8080
    subset: v1
  weight: 0
- destination:
    host: cpanel.default.svc.cluster.local
    port:
      number: 8080
    subset: v2
  weight: 100
```

$ 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





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
```

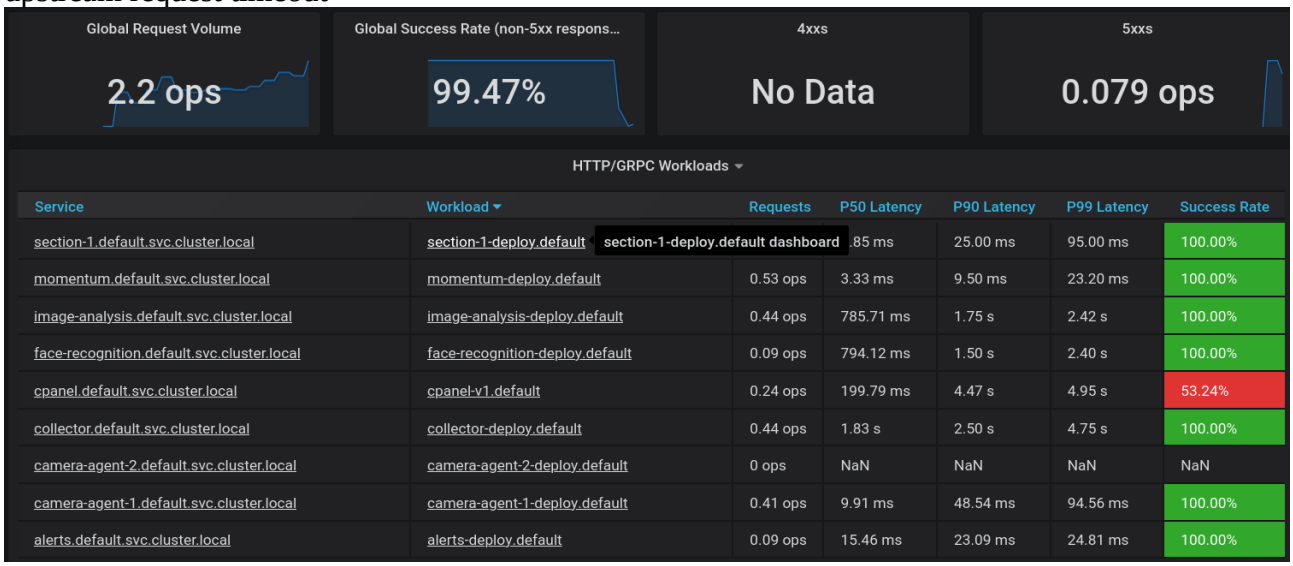
$ make fault-injection-delay10

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

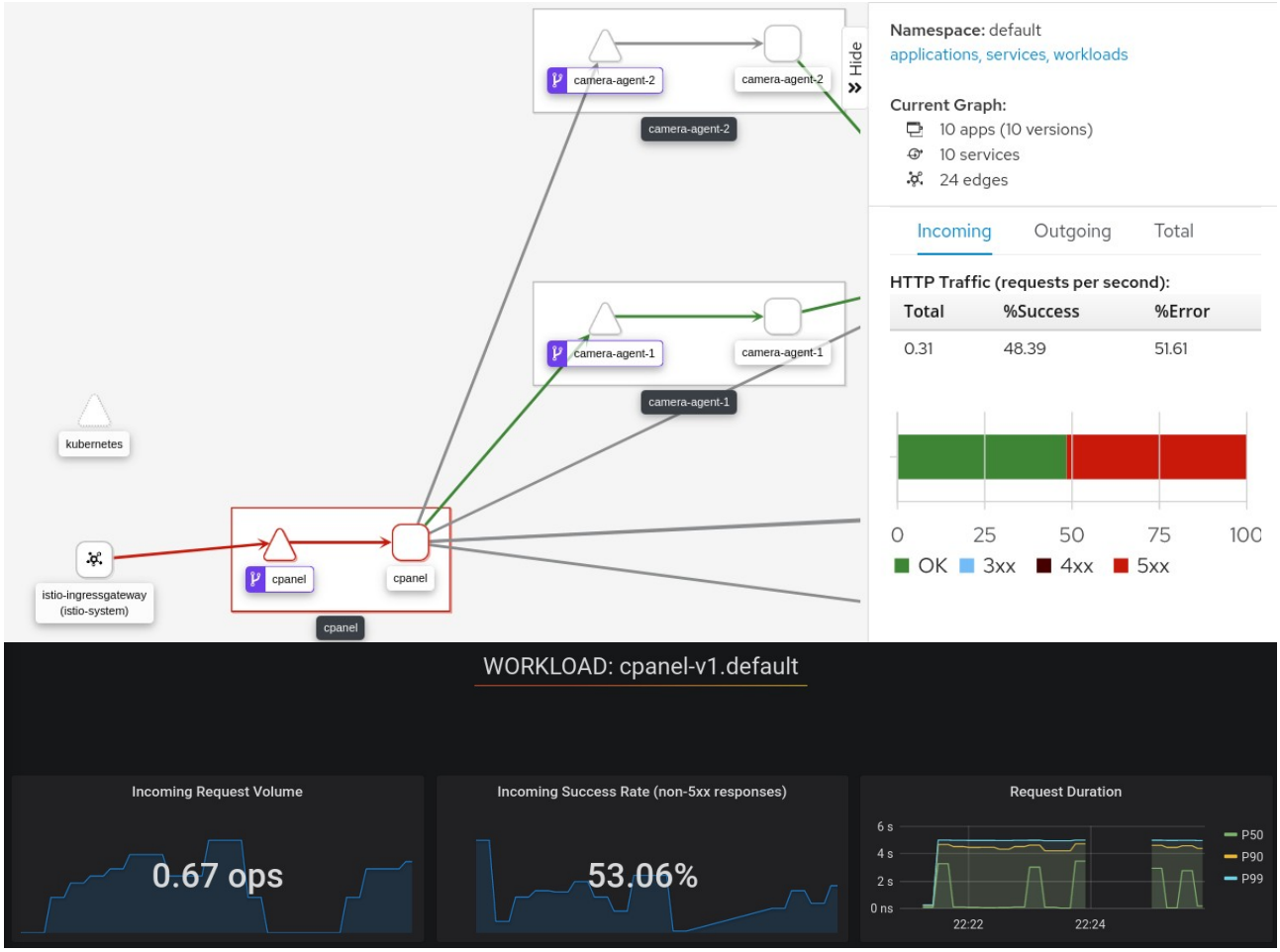
In order to check how timeout mechanism works fixed delay 10 seconds for camera-agent-1 with success rate 50% was configured in virtual service. So after applying this configuration every second request to camera agent will be delayed. To protect client from waiting to long (full 10 seconds) timeout 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
```



Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
section-1.default.svc.cluster.local	section-1-deploy.default		.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



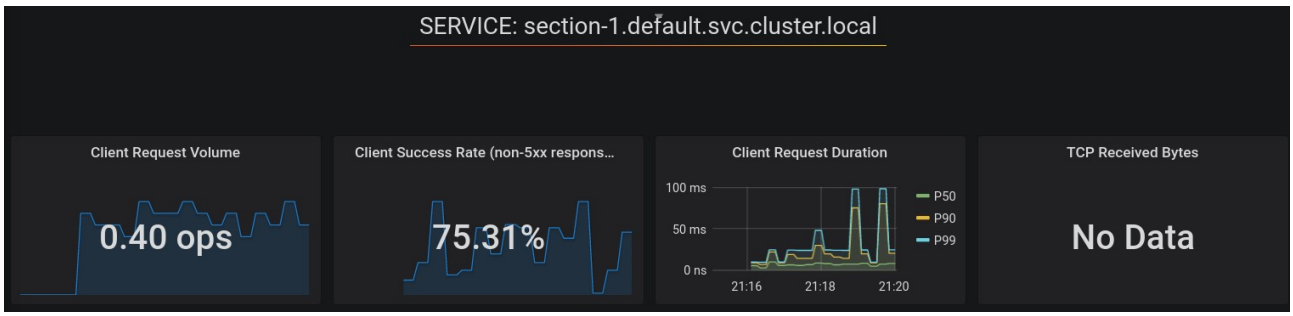
$ make deploy-istio-default

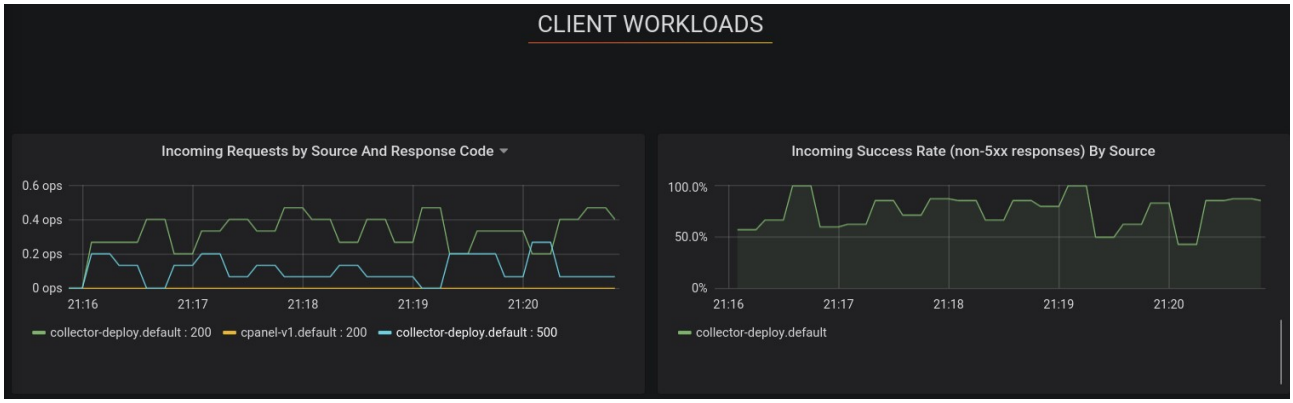
Retries

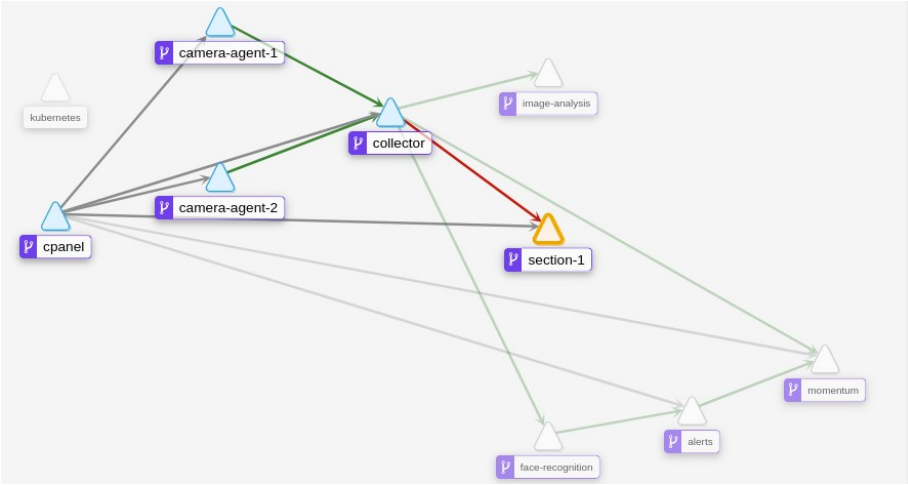
To demostrate retry pattern in istio fault-injection was made in virtual service section-1 so that every fourth request will response with 500 HttpError. As a result collector will have failed requests and the data can be lost. To tolerate this artificial error rate retries were configured in collector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

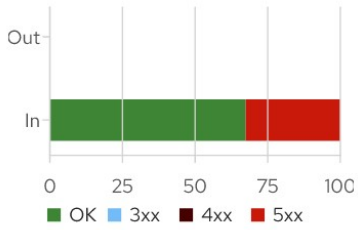






HTTP Traffic (requests per second):

	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
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

Application was successfully deployed with istio. Different resiliency hardening approaches were realized in demo both on side of kubernetes and istio. Graphics and console output shows the results.

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.

Centrally managed decentralized system, extra hops, immature [enterprise].

- 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

over 1062 opened issues in github [issues].

not really production ready

Future Work

Discussed only a small aspect of istio functionallity. Try out other istio features and also resiliency with enabled mTLS.

Moving to a real world application and applying resiliency step by step in it.

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. Report bugs on github to paricapate in develeopment process.

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)https://www.martinfowler.com/articles/microservices.html (accessed 07.03.2020)
3. (sec)https://snyk.io/blog/top-ten-most-popular-docker-images-each-contain-at-least-30-vulnerabilities/ (accessed 07.03.2020)
4. (cc)Cloud Computing Assignment
5. (twelve)https://12factor.net/ (accessed 07.03.2020)
6. (k8s)https://kubernetes.io/ (accessed 07.03.2020)
7. (istio)https://istio.io/ (accessed 07.03.2020)
8. (docker)https://www.docker.com/ (accessed 07.03.2020)
9. (alt)https://aspenmesh.io/service-mesh-architectures/ (accessed 07.03.2020)
10. (tele)https://www.telepresence.io/ (accessed 07.03.2020)
11. (action)Bruce, Morgan, and Paulo A. Pereira. *Microservices in Action*. Manning, 2019.
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) Alex Williams. Guide to Cloud Native Microservices. The New Stack, 2018.
14. (mesh)https://servicemesh.io/ (accessed 07.03.2020)
15. (microgit)https://github.com/davidetaibi/Microservices_Project_List (accessed 07.03.2020)
16. (enterprise)Indrasiri, Kasun, and Prabath Siriwardena. *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress, 2018.
17. (advanced)Hunter, Thomas. *Advanced Microservices: a Hand-on Approach to Microservices Infrastructure and Tooling*. Apress, 2017.
18. (microcon)Kocher, Parminder Singh. *Microservices and Containers*. Addison Wesley, 2018.
19. (meshpath)Lee Calcote. The Enterprise Path to Service Mesh Architectures. O'Reilly Media, 2018

20. (appmicro)Alex Williams, Benjamin Ball. Applications and microservices with docker and containers. The New Stack, 2016
21. (prodmicro)Fowler, Susan J. *Production-Ready Microservices: Building Standardized Systems across an Engineering Organization*. O'Reilly Media, 2017.
22. (designmicro)Chris Richardson, Floyd Smith. Microservices From Design to Deployment. NGINX Inc., 2016
23. (buildmicro)Newman, Sam. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2018.
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)Calcote, Lee and Butcher Zack. *Istio: up and Running*. O'Reilly Media, 2019.
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. (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.
31. (java) Michael Hofmann, Erin Schnabel, Katherine Stanley. Microservices Best Practices for Java. IBM Corp., 2016
32. (issues)https://github.com/istio/istio/issues (accessed 08.03.2020)
33. (linkerd)https://glasnostic.com/blog/comparing-service-meshes-linkerd-vs-istio (accessed 07.03.2020)
34. (git)https://github.com/van15h/resilient_istio (accessed 07.03.2020)

Supplemental Material

- Cloud computing assignment (git)
- Useful commands (git)