Abstract

The expanse of cloud computing technologies and movement to platforms as a service bring 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. Some of the solutions are migrating from monolith to microservice architecture on already running projects 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 running on top of the Kubernetes cluster will be introduced as a solution to provide visibility, control, security and fault tolerance to application deployment [istio], [k8s]. The final goal is to demonstrate the possibilities of Istio and try out the resiliency features on the microservices application.

Keywords

Microservices, REST, Containers, Docker, Kubernetes, service mesh, Istio, resiliency, fault tolerance

Table of Contents

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 implementation 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.

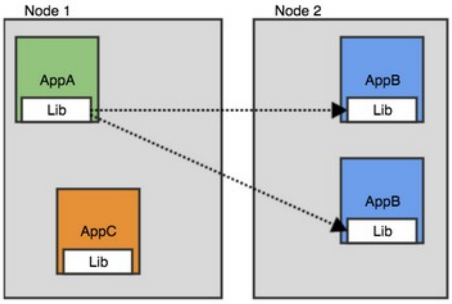


Figure 1: Service mesh based on libraries [alt]

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].

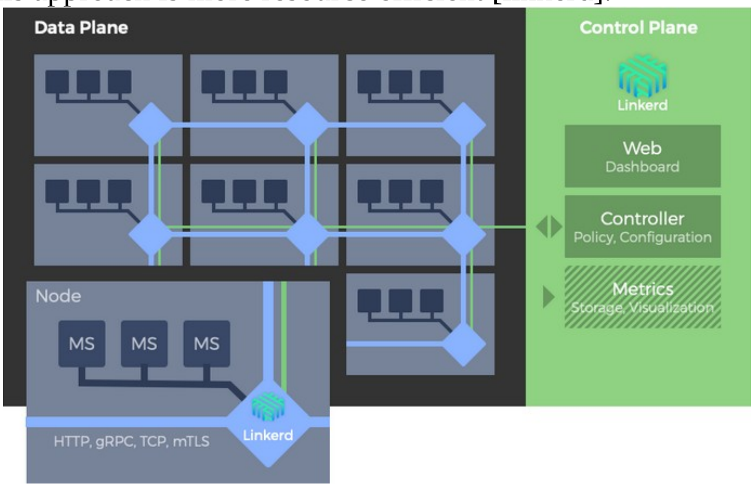


Figure 2: Linkerd architecture with node agents [linkerd]

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.

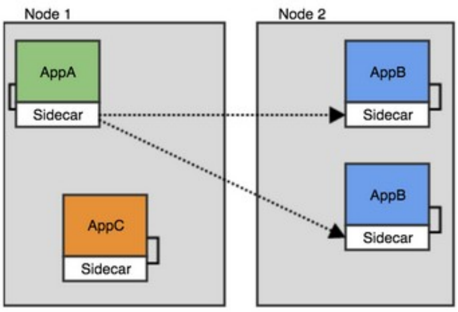


Figure 3: Service mesh based on sidecar proxies [alt]

Major idea

There are plenty of tutorials online that utilize a sample application from Istio web site ("Bookinfo" application) to show typical service mesh and specific Istio features [istio]. 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 demonstration 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].

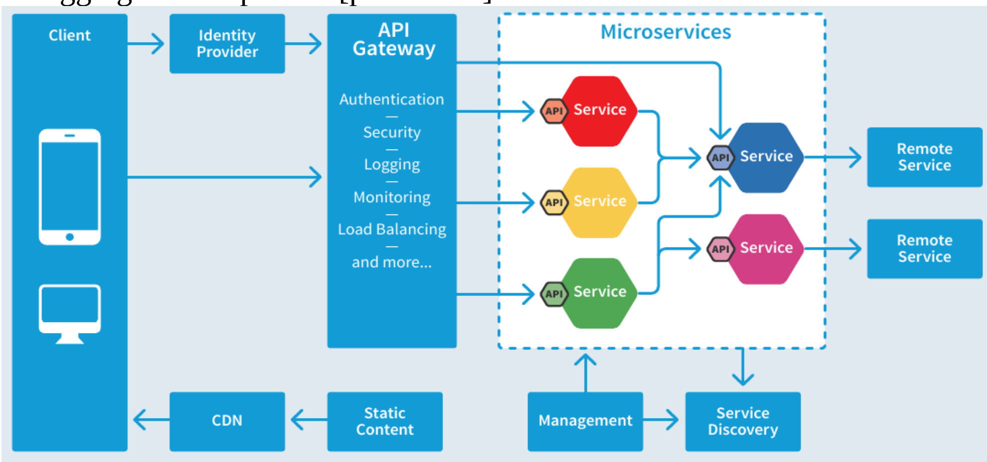


Figure 4: Web application with microservice architecture [native]

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 and deliver them from developers to testers and from testers to production environment. Immutable images and containers solved 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].

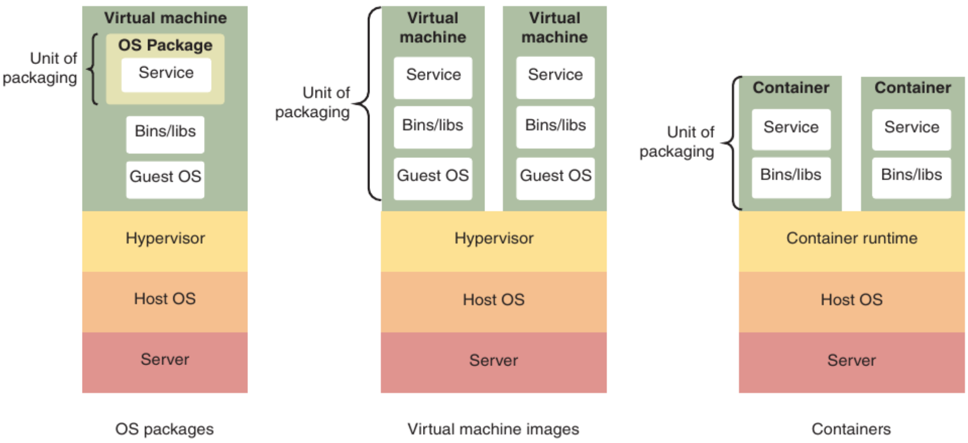


Figure 5: Comparison of packaging solutions [action]

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],[towards]:

- lack of 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. The project is open source and was started by Google, IBM and Lyft [uprun]. It is a network of services that make the application. This service mesh is designed to add application-level observability, routing and resiliency to service-to-service traffic with almost no changes to the application code. Tracing, monitoring and logging gives operators a full overview over the deployed microservice application.

Istio provides such relevant for microservice architecture attributes as: trusted communication, encryption of all internal traffic with mutual TLS enabled, tracing of requests (Jaeger), metrics (Prometheus), alerting and graphics (Grafana), visualization of the mesh topology (Kiali), advanced traffic management with routing and load balancing, communication and network resiliency, configuration API, policies to enable rate limiting, denials and white/black listing.

The only situation when some code changes will be needed is when tracing of calls between services should be enabled. This kind of feature will need to add some custom headers propagation from service to service.

The traffic inside the web application between microservices is called “east-west” traffic. The opposite kind of traffic is “north-south” and is referred to ingress and egress services.

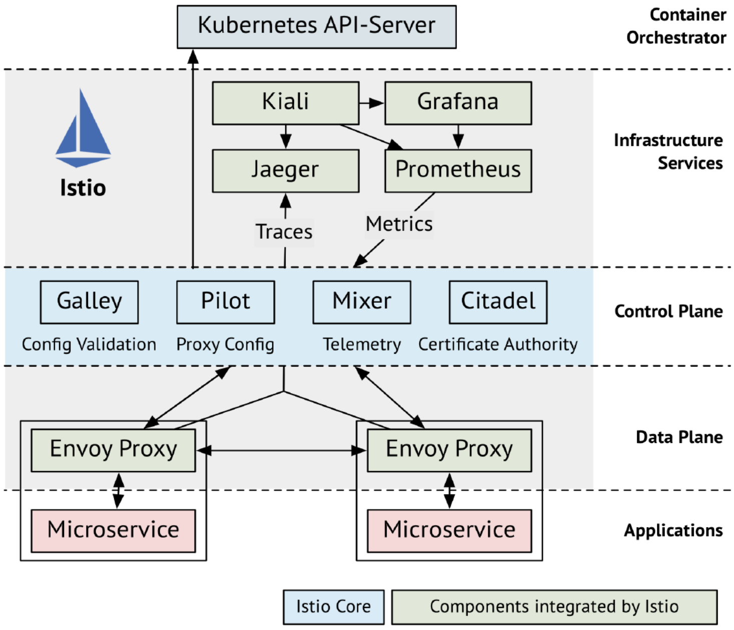


Figure 6 : Istio architecture

An Istio service mesh has two logically zones: a data plane and a control plane.

Data plane traffic is all the internal requests between the services of deployed application. Envoy proxies are injected in each pod of the microservice as sidecars. Communication with service is possible only through these proxies. Traffic routing in Istio can be done only on data plane level. Mirroring of the traffic can be enabled for microservices. It is not really efficient, can be handy for some special situations, e.g. the service is read-only so enabling mirroring will not consume much of computing resources.

Control plane traffic is used to configure and manage Istio components to reach a stable deployment of the mesh. It also manages the sidecars to allow routing and configures policies and collection of telemetry.

Control plane is also responsible for:

- automatic load balancing.
- retries, circuit breakers, fault injection.
- policies with access controls, rate limits and quotas.
- telemetry for all traffic within and incoming and outgoing from a cluster requests.
- authentication and authorization.

Pilot talks to Kubernetes via adapter to enable service discovery (can also work on top of Consul). It takes configured in Istio traffic rules and sends them to proxies in data plane. Pilot works dynamically without restart needed is also responsible for resiliency in the mesh.

Citadel provides end-to-end encryption and user authentication. Mutual TLS configurations are done with it help.

Galley holds, validates and distributes configurations.

Mixer represents a layer between the Istio components, accompanying services and the infrastructure backends used for access control with policies and telemetry. All sidecars proxies ask mixer, if request is allowed.

Sidecar proxy (based on Envoy) adds behavior to application microservice without changing its code. A combination of proxy and a microservice is seen inside Istio as one logical unit. These proxies allow such Istio features as circuit breakers, health checks, fault injections, rich metrics, load balancing and others.

Kiali helps to visualize the application to get an overview about what is deployed and who talks with whom.

The main purpose of Grafana is to give a graphical view about metric data, to create custom dashboards and trigger alerts [action]. It works with Prometheus as backend. With Grafana fault injections and overall behavior of the traffic load can be seen very good. Kiali and Grafana were widely used in the evaluation part of this thesis.

To configure traffic behavior inside service mesh Kubernetes custom resource definitions (CRD) are used. Istio provides following resources for traffic management:

- virtual services – how to route the traffic inside service mesh to a given destination. They are used to allow canary deployments. Routing is done based on different matching criteria (routing rule): weights, headers, URLs. Retries, timeouts and fault injection can be configured here.
- destination rules – applied after routing is done. They consist of named subsets, traffic policies: circuit breaker and load balancing configurations.
- gateways – used to allow incoming and outgoing traffic from the mesh. Each gateway is an Envoy proxy deployed on the edge of the mesh.
  - ingress – used to expose service outside the cluster.
  - egress – used to allow access to external services outside the cluster. By default all external traffic is blocked and need to be enabled in service entry.
- service entry – inherited automatically from Pilot, that takes service names and ports from Kubernetes service discovery. They also are used to add external services to Istio registry (for egress destinations).

Gateway and service entries control the incoming and outgoing traffic. Virtual service and destination rule control the east-west traffic (inside service mesh).

Istio provides excellent features to manage your microservice application out of the box as all-in-one solution and shows the need of service meshes in modern deployments. One of the goals of Istio service mesh is to put resiliency into the infrastructure.

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. Microservice must not block a request because then other microservices might be blocked, the error propagates and cascade error happens. Also simple network delays can lead to such problems. So it is necessary to be sure that one failed service does not bring down the entire system.

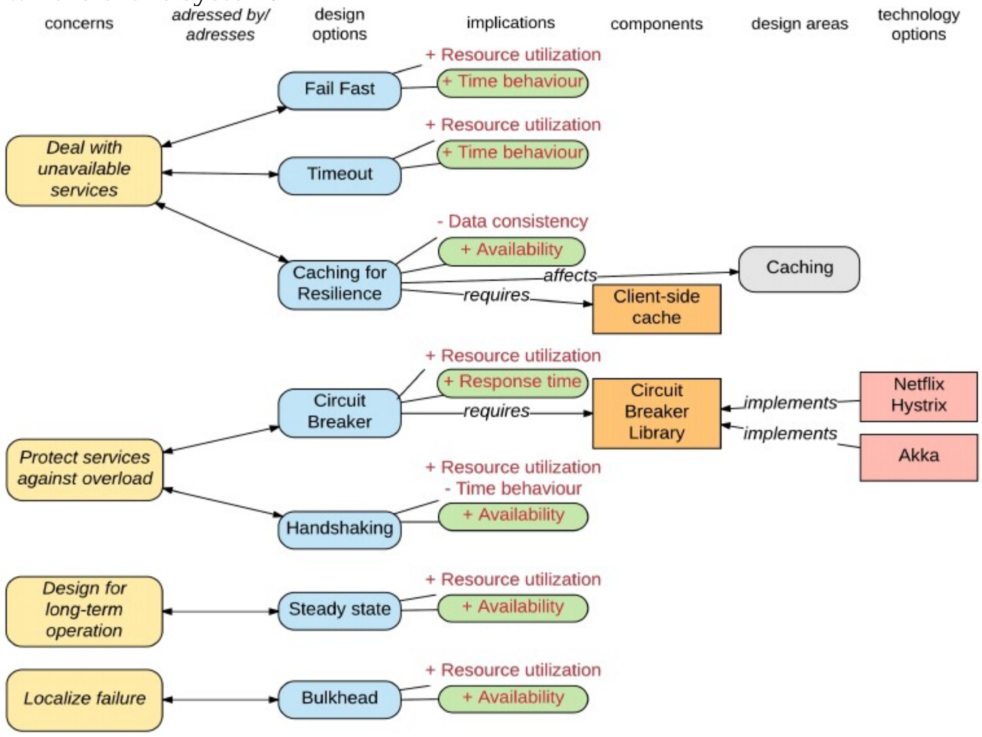


Figure 7: Possible failures and solutions for distributed applications [decision]

A resilient microservice application is one that can recover from failures at every level of the system: the hardware layer, the communication layer, the application layer and the microservice layer. There are several types of resiliency testing that can evaluate the fault tolerance of a deployment [prodmicro]. These are load and chaos testing.

Istio provides the following resiliency features: health checks, load balancing, delay injection, fault injection, timeouts, retries, rate limits, circuit breaker.

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

There are two types health checks: 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]. Probes can be HTTP GET requests, “exec” scripts executed inside a container or TCP socket checks.

Though these are the mechanisms that belong to Kubernetes they are still worth to mention because Istio proxies allow these health checks to work seamlessly. One difference is that HTTP health checks work only with enabled mutual TLS. So some configurations on the side of Istio system namespace need to be done. “Exec” and TCP health checks work straight forward without any changes in Istio.

Load balancing in Istio provides some more sophisticated algorithms then native Kubernetes solution (round robin). They can be configured in destination rules:

- round robin - used by default
- random - random pods are taken for requests from load balancing pool
- least requests - least overloaded pod get new requests

```
loadBalancer:
      simple: RANDOM
```

Timeout helps to deliver fast responses to client without waiting for response from slow service. For a better user experience it is a good practice to fail fast then to wait long for response. Default value for timeout in Istio is 15 seconds. It can be changed for each microservice in virtual service configuration file. How to choose a proper timeout for calls depends on application and microservice. A small one can not be enough to process the request. A big one can slow the overall functionality of the system. Just 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
```

Retries repeat the failed request in order to get the response faster than return an error to client and initialize a completely new request. By default no retries are configured in Istio deployment, but it can be done in virtual services for each microservice. Normally developers take care of retries 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 the service will be higher. This should be taken in consideration and could also be protected with circuit breaker.

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

Circuit breaker pattern is used to protect the application from the failing microservice. It can be configured in Istio in destination rules for each microservice. There are two types of this pattern in Istio.

The first one works at 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 (e.g. 50x). In this case Istio will eject the problem pod from the load balancing pool for some time:

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

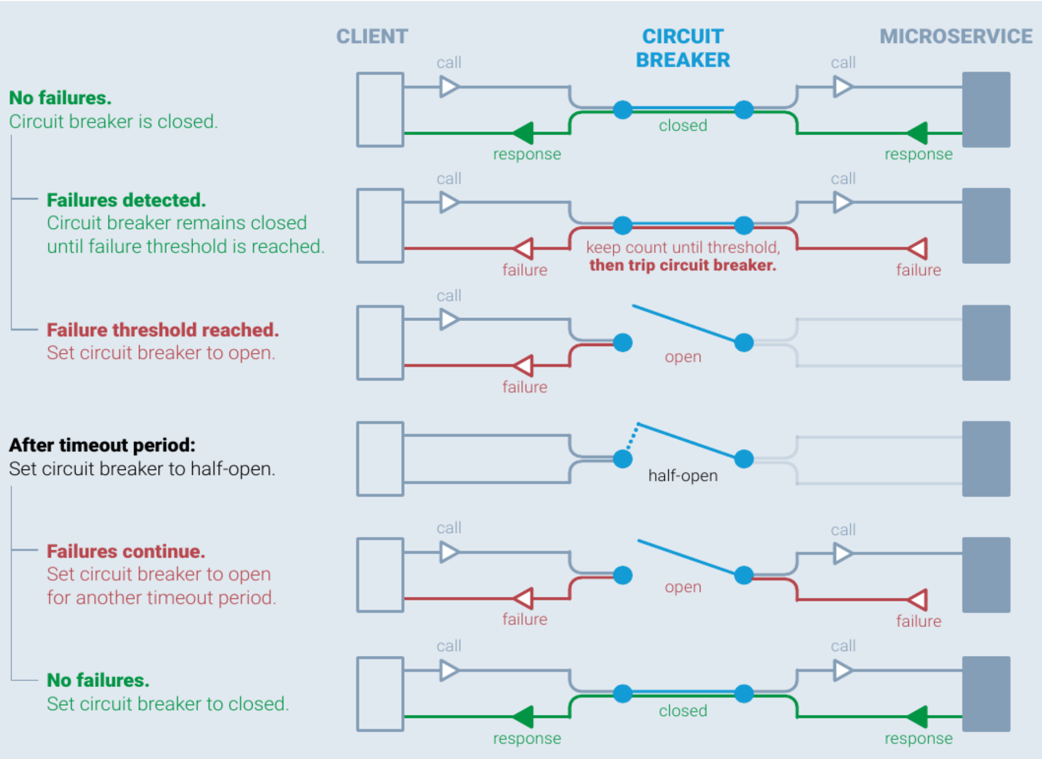


Figure 8: Circuit breaker pattern [native]

Demonstration

The main result of this thesis will be a fully working demo to show the 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, orchestration and monitoring. Docker files and Kubernetes manifests contain best practices from production ready deployments.

Implementation

The Twelve Factors Application

Application itself is a simulation of airport security system based on microservices architecture with exposed REST API [rest], [cc].

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 (PUT /config). You can list all the cameras configured in the system with “GET /cameras “.

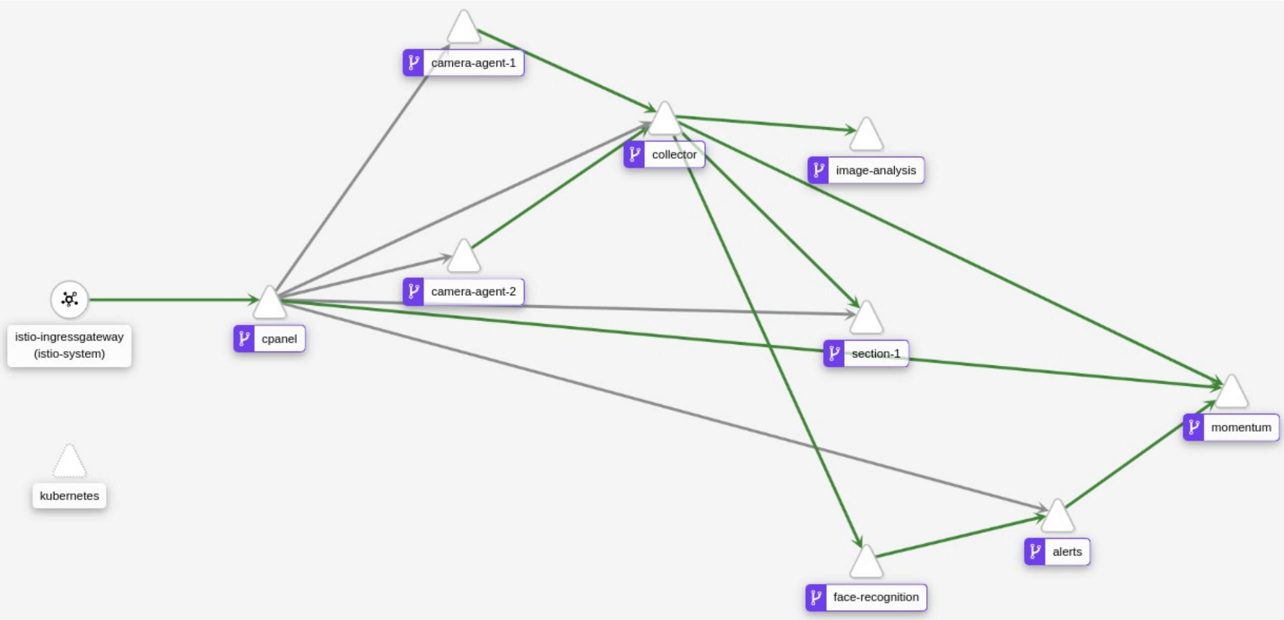


Figure 9: Visualization of deployed application from Kiali

Collector receives frames from camera agents in json format and forwards them to image analysis and face recognition microservices for analysis.

Image analysis takes frame and responses back with the statistics about how many people are there, their gender and age. After that collector forwards statistics information about current image to appropriate section for persistent storage and to 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. To keep the implementation simple no database storage was made.

Face recognition forwards response if there are any persons of interest on the image to alert microservice.

Alert microservice provides persistent storage (json file) for all the persons of interest found and also forwards the response from face recognition to momentum microservice.

There are two versions of control panel microservice that serves as a minimal dashboard for users to show currently processed images and also as an API gateway to control the state of the deployed system. It hides backend microservices from the client and exposes only necessary endpoints [microcon]. It is similar to facade pattern in object oriented design [designmicro]. The difference between version 1 (V1) and version 2 (V2) of the service is only in displayed dashboard. V1 has dashboard without any images from processed frames, but V2 is more user friendly and has images with all frames. Request routing can be configured between these two versions with the help of Istio.

...

Figure 10: Sequence diagram of the deployed application

Camera agents, image analysis (Java) and face recognition (Python) microservices were already implemented and provided as docker images. The rest of microservices (collector, section, alerts and control panel) were developed during the cloud computing course. Momentum microservice was added to separate temporally logic of saving current processing frames. Frontend dashboard was added on the server side of control panel microservice with minimal functionality – just to display currently processed frames from momentum microservice. All of newly added microservices are written in Python with Flask framework. In this way we get a polyglot stack (Python, Java) that is typical for microservices architecture and can get maximum value from Istio service mesh.

More detailed description of the initial API and the hole system itself can be found in cloud computing assignment in the supplemental materials [cc]. The following additional endpoints were implemented for each microservice:

alerts:	GET /status
collector:	GET /status
	GET /fault
section:	GET /status
cpanel:	GET /status
	GET /, /index
	GET /analysis
	GET /alert
	GET /momentum
	PUT /config
	POST /production
momentum:	GET /status
	GET/POST /analysis
	GET/POST /alert

To make the web application cloud native it should be in compliance with “The Twelve Factors Application” principles [twelve]. Most of them are realized in this application, but with some constraints not to make the simulation, deployment and debugging too complicated:

1. Codebase: all code versions of all microservices is tracked in Git and synchronized with GitHub.
2. Dependencies: all necessary dependencies for each microservice are packaged with application code inside if container image. They are added to container while building the image from the provided individual Dockerfile. These dependencies are declared in “ requirements.txt” file.
3. Configuration: all configuration of the microservices is done with environment variables in Kubernetes manifest files. The only exception is
4. Backing Services: Is not done properly as instead of databases json files are used. As workaround these files can saved in host file system with the help of Kubernetes mount volumes. A more better microservice oriented approach would be to realize persistent storage for each potential section with an independent database container.
5. Build, release, run: Docker images are used to provide portability and isolation and to package the application with all dependencies in one container.
6. Processes: each microservice is run as a separate process inside Docker containers.
7. Port binding: each microservice is exposed to internal Docker network via port binding.
8. Concurrency: adding more concurrency is done simple by scaling out the microservice container with the number of replicas and load balancing between them.

9. Disposability: Docker engine with Kubernetes orchestration is used to deal with managing the containers in fast way.
10. Dev/Prod parity: Docker containers help to keep the development and production versions of the application as similar as possible. Regular image builds and fast deployment updates make it easy. Otherwise such tools as Telepresense can be used to make fast changes in code and debugging [tele].
11. Logs: logs from each microservices are streamed to “stdout”. Then they can be aggregated and use to understand the behavior of the system for some problematic points of time.
12. Admin Processes: No administration processes should be run manually vis SSH inside of containers. These tasks should be unloaded to another process/container.

Here is a small list of changes that were done to the application after initial Git commit with the project from cloud computing assignment [git], [cc]. It is better to look into commit history to get a complete overview:

- shell scripts were added to configure and manage the development environment
- frontend dashboard was added to control panel to make it more user friendly
- control panel was divided into versions (v1, v2) to make possible the demonstration of canary and blue/green deployments [java] with routing mechanism of Istio
- Python and Docker best practices were implemented in Dockerfiles to make the containers more isolated and secure:
  - alpine image was used [sec]
  - no cache is left after installing all dependencies from “requirements.txt”
  - application is not running with root permissions
- momentum microservice was added
- health check, statistics, fault simulation endpoints were added
- Kubernetes manifest files were updated with environment variable, health checks and resource limits
- Istio configuration files were added to demonstrate each type of resiliency features
- Makefile was added to provide easy interacting while demonstrating Istio resiliency possibilities

Deploy with Kubernetes

The microservice application and Istio are deployed in Minikube Kubernetes cluster. Kubernetes of version 1.15.7 was used because it was officially tested with Istio.

Pod is the smallest unit of deployment in Kubernetes. It can consist of one or more containers. In our case each pod has two containers: sidecar proxy and microservice.

Native server side service discovery of Kubernetes was used to configure communication between microservices [microcon]. Service defines a set of pods and provides a method for reaching them, either by other services in the cluster or from the outside [action]. Services provide stable endpoints for pods and are automatically registered in service registry that is built-in in kubernetes architecture [designmicro]. Service discovery enables to use host names as destinations in calls. That is very helpful because if pod restarts it gets a new IP address and that makes the deployment inconsistent. When you create a service, it creates a corresponding DNS entry in service registry.

Deployments in Kubernetes are designed to describe the desired state of ReplicaSets [action]. By default round robin load balancing works on top of related microservices for number of replicas [microcon].

Good places to configure replication of pods in our application are collector, image analysis and face recognition. If there will be plenty of users to call the dashboard a good approach is to decouple frontend from control panel microservice and put it in a separate container to allow scalability. Momentum microservice that provides data for dashboard can also be seen as a bottleneck. In this case an Istio feature with traffic mirroring can be realized to provide more then one replica of the service with the same state. All requests to each replica of the momentum microservice will always deliver the same result.

Readiness and liveness probes were added to deployment manifests to increase resiliency. The same applies to resource limits that help to protect other pods from “starvation”. On the other hand providing a large number of resources (CPU, RAM) to a pod that doesn’t utilize it is also inefficient. Unused but reserved hardware resources may be costly [prodmicro].

Deploy with Istio

Single cluster deployment of Istio version 1.4.3 sidecar auto injection was used for the test purposes. Istio was deployed by means of shell script and enabled in demo mode with full list of additional services. Core components consist Istio pilot, ingress and egress gateways. Addons to provide the most of observability features are Grafana, Jaeger, Kiali and Prometheus. Service mesh installation verification is done in shell script. It is also possible to make a completely custom installation, but it was not the goal of this work. So full featured demo profile was taken.

According to traffic management best practices from Istio virtual services and destination rules with default subsets were configured for each microservice. Ingress gateway is added to control panel virtual service to expose it outside of Minikube cluster.

It is not recommended to use short names for as destination hosts in Istio configuration files. A problem with cross namespace communication can arise from Kubernetes side. That is why everywhere in Istio configurations the fully qualified domain names (FQDN) are used.

As workaround and to protect to the system from overloading traffic mirroring can be configured on momentum microservice with the same subset 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 Virtual Service in Istio.

All Istio configuration files for the test cases in this thesis are moved to separate yaml files to enable easy switching between them while doing the demonstration of resiliency features.

Makefile gives an opportunity to try out Istio resiliency features in more user friendly form.

How to run

Requirements: linux, virtualbox, minikube, curl. Virtual machine to run Minikube cluster need at least 4 CPU and 8GB RAM (default configuration in installation scripts is 4 CPU and 16GB RAM).

Steps to deploy the application and Istio:

- clone the project with (if SSH is configured, otherwise change to HTTPS link):
  ```
  git clone git@github.com:van15h/resilient_istio.git
  ```
- go to project folder:
  ```
  cd resilient_istio
  ```
- create Minikube virtual machine with:
  ```
  ./create_minikube_cluster.sh
  ```
- install and deploy istio to Minikube with:
  ```
  ./install_istio.sh
  ```
- to use Docker engine from Minikube locally run:
  ```
  eval $(minikube docker-env -p airport)
  ```
- export variables for local bash with:
  ```
  export INGRESS_HOST=$(minikube ip -p airport)
  export INGRESS_PORT=$(kubectl -n istio-system get service istio-
  ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
  echo "INGRESS_HOST=$INGRESS_HOST, INGRESS_PORT=$INGRESS_PORT"
  ```
- after all istio services are up and running to get Minikube IP and port as environment variable run for current shell session run:
  ```
  ./generate_minikube_url.sh
  ```
- to build Docker images locally run:
  ```
  ./build_containers.sh
  ```

- use `Makefile` to deploy the application
- use `Makefile` to play around Istio resilliency features
- to cleanup all run:
  `./cleanup.sh`

Evaluation

In this section resiliency features of Istio service mesh are introduced in practice. Kiali graphs, Grafana graphics and console outputs help to understand how fault tolerance can be configured with the help of 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.

To deploy the application:
$ make deploy-app-default
./kubectl apply -f k8s
./kubectl get pods –watch
Wait until all pods are up and running and stop monitoring them with Ctrl-c.

To deploy Istio configuration for current application deployment:
$ 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 &

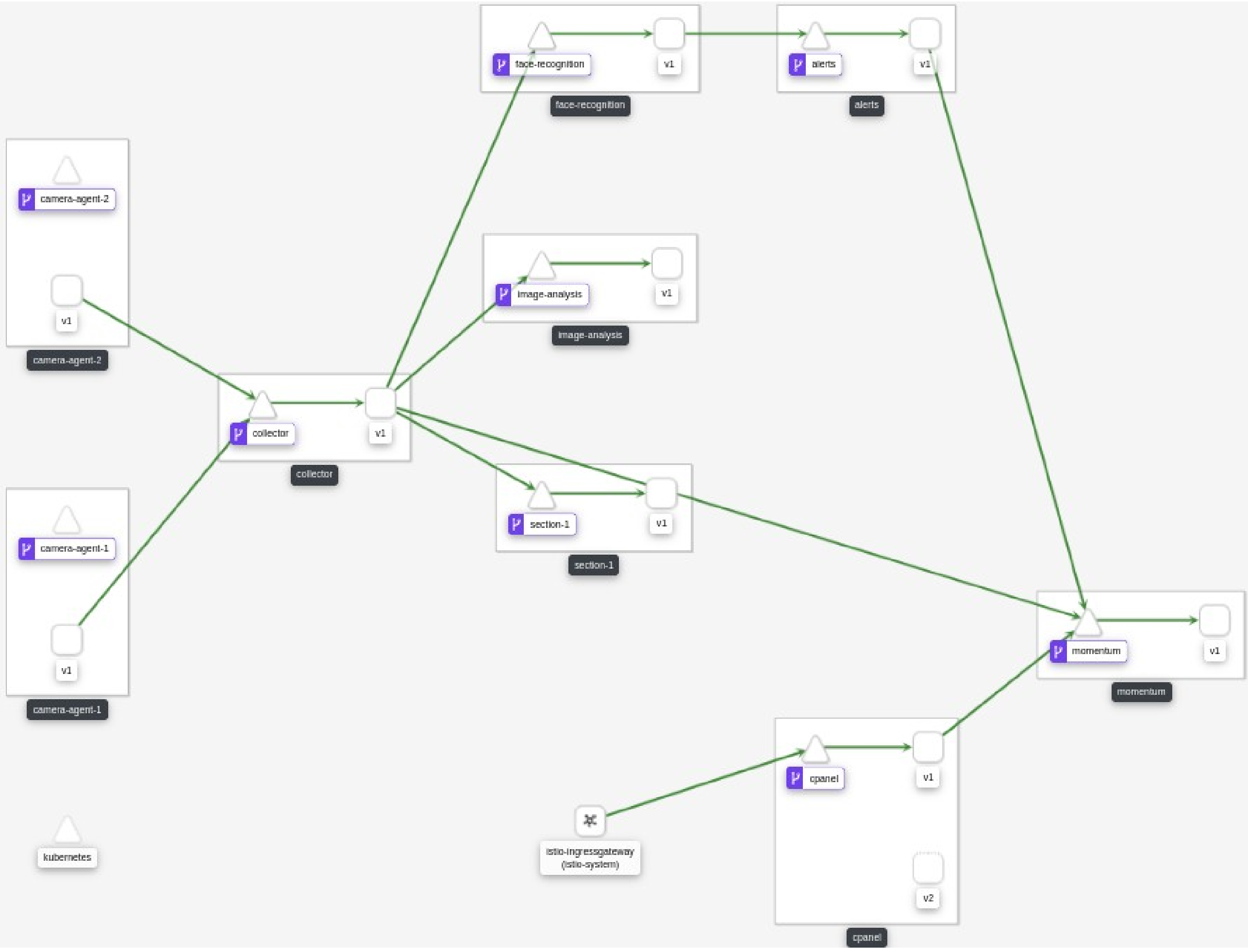


Figure :
Version 1 of Cpanel microservice displays information about latest statistic from image analysis and the most recent alert. Both are displayed without showing the photo from camera agent itself. Splitting between admin users and normal users can be done in virtual service with help of headers. Displaying the photo is made in Version 2 of Cpanel microservice.

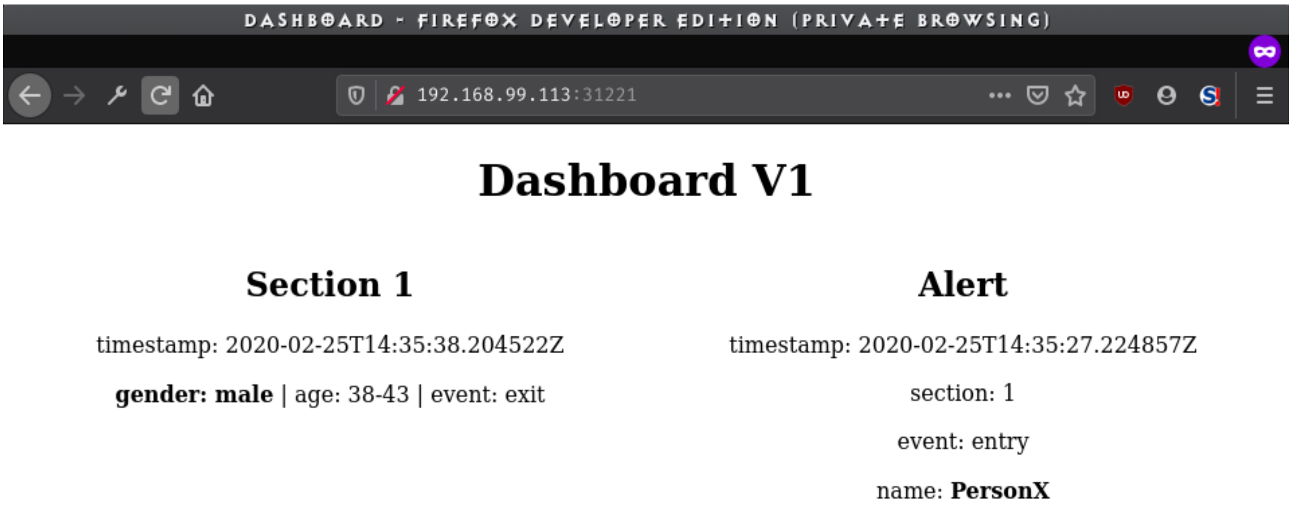
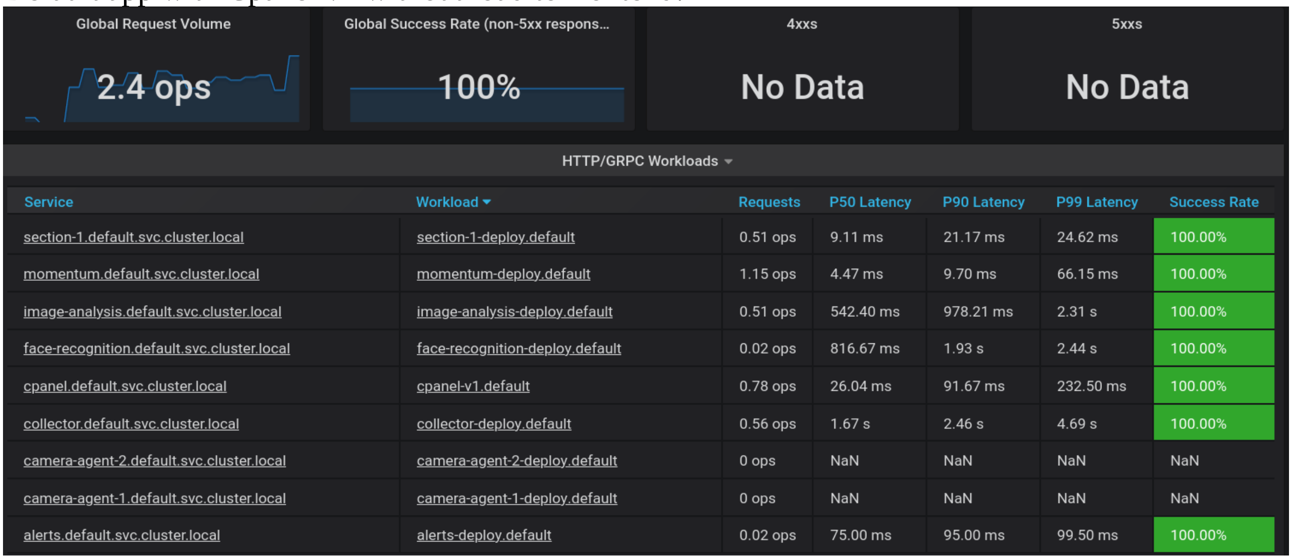


Figure :

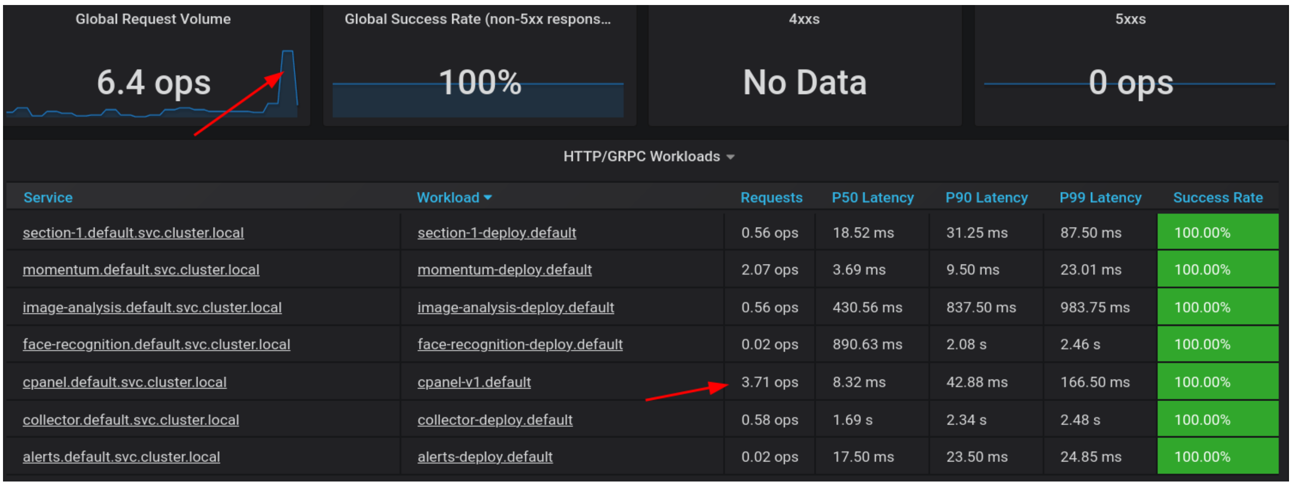
Default app with Cpanel v1 without load to frontend:



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%

Figure :

$ 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%

Figure :

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
```

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>
```

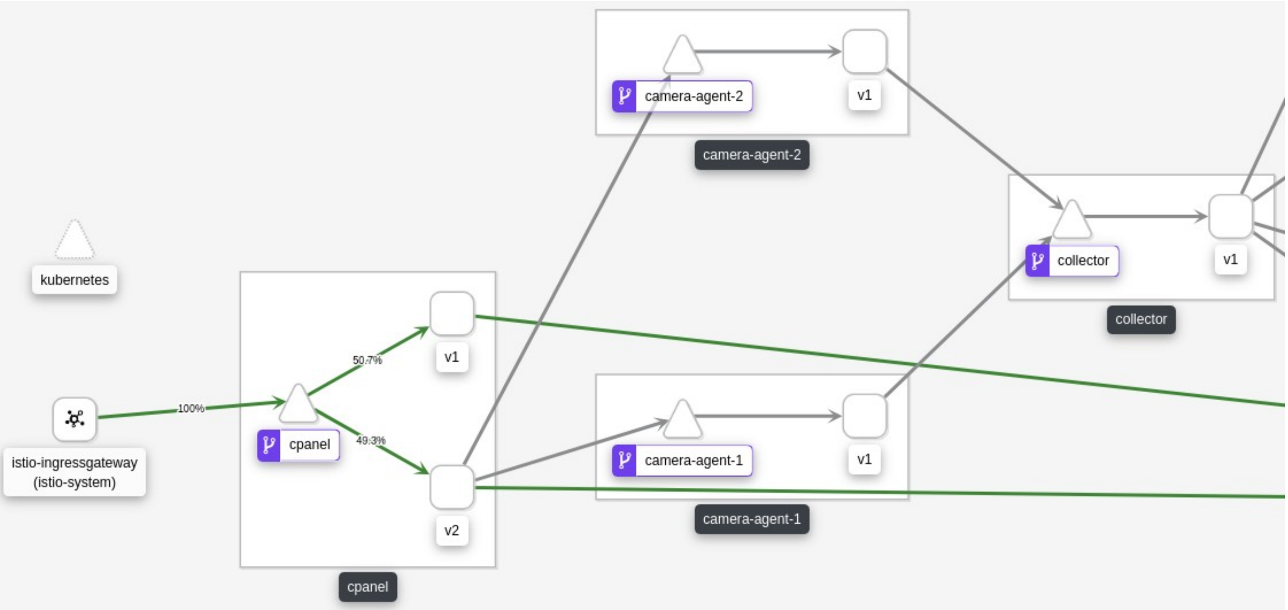


Figure :

```
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

check configuration

$ ./kubectl get virtualservices cpanel -o yaml

$ make start-cameras

curl http://192.168.99.113:31221/production?toggle=on

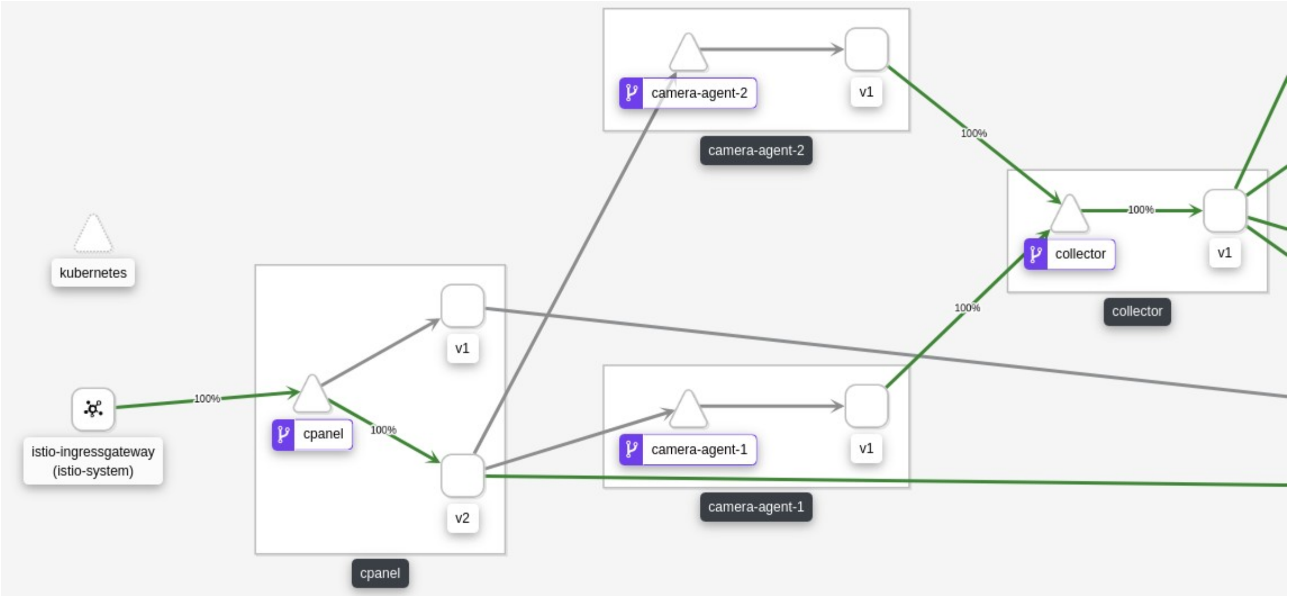


Figure :

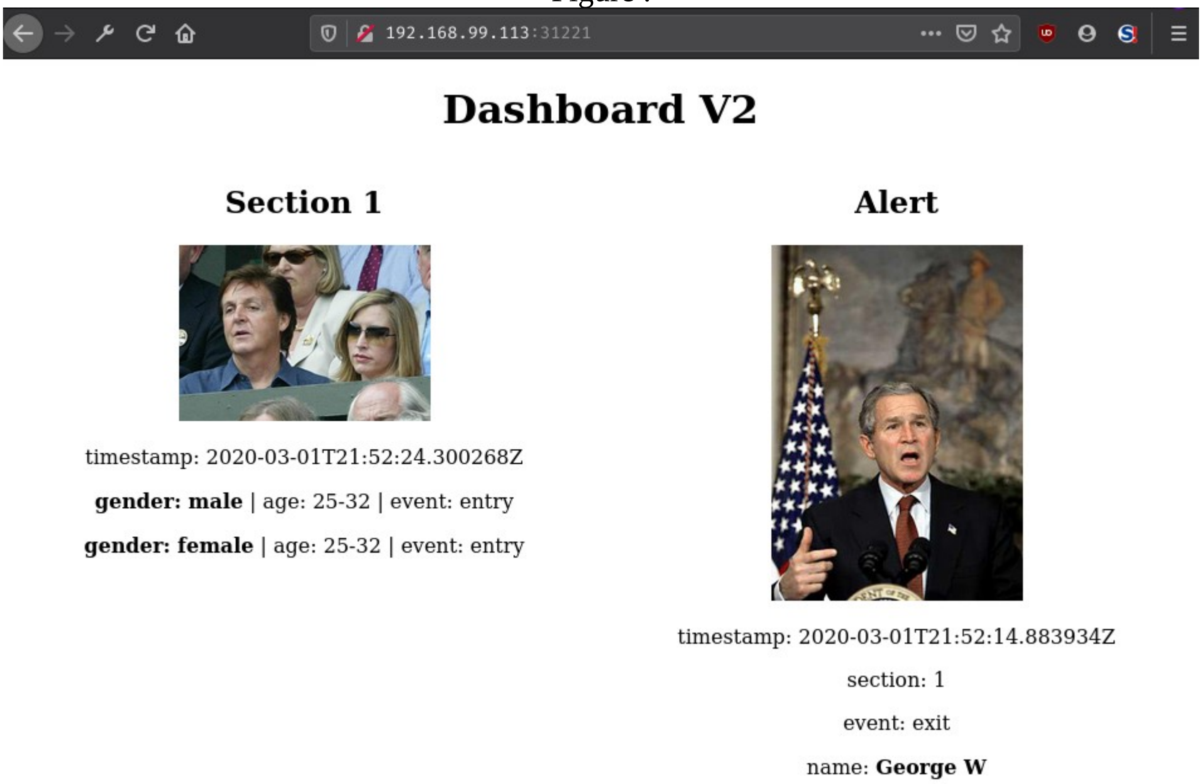


Figure :

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
```

```
$ 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
```

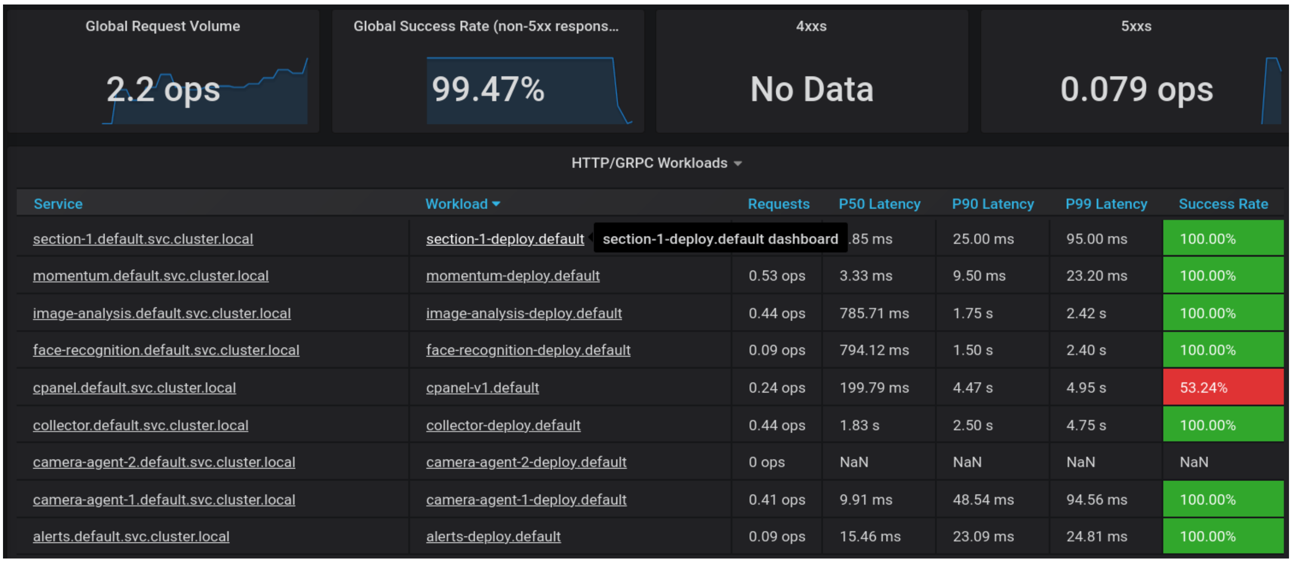


Figure :

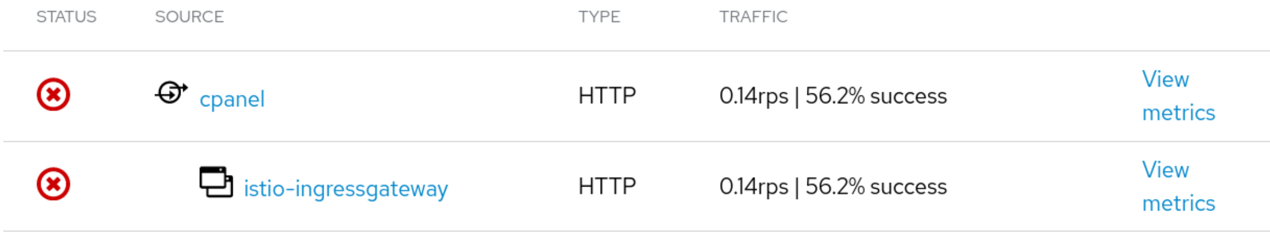


Figure :

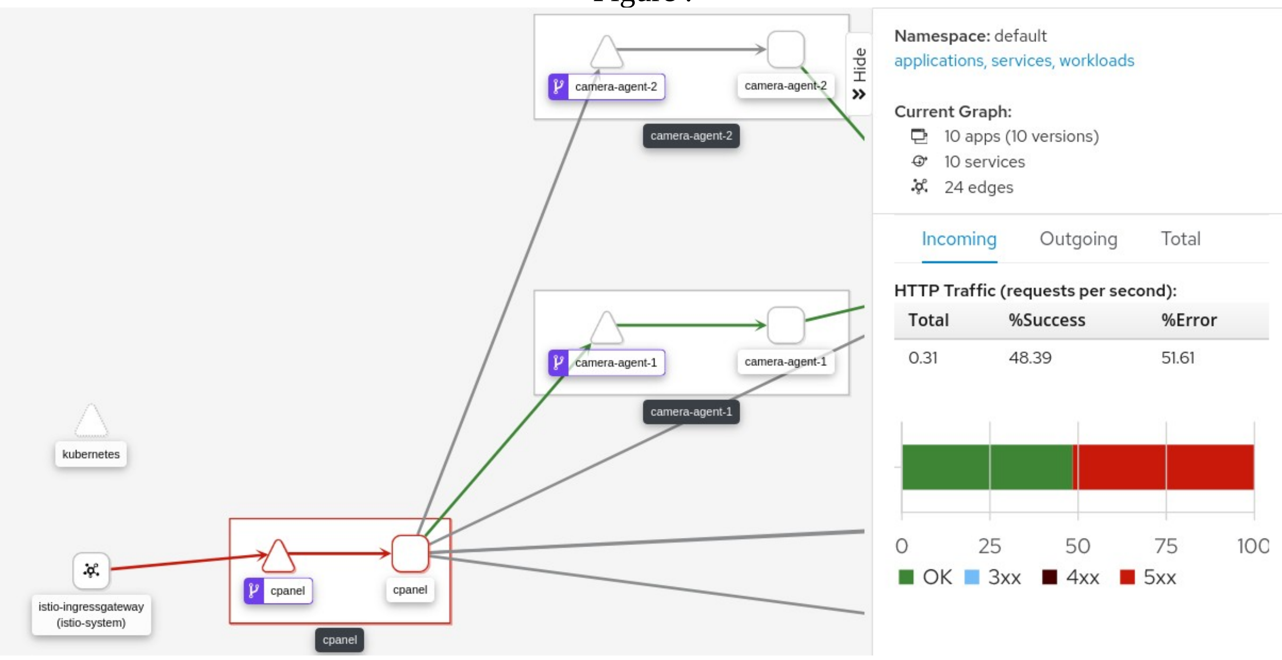


Figure :

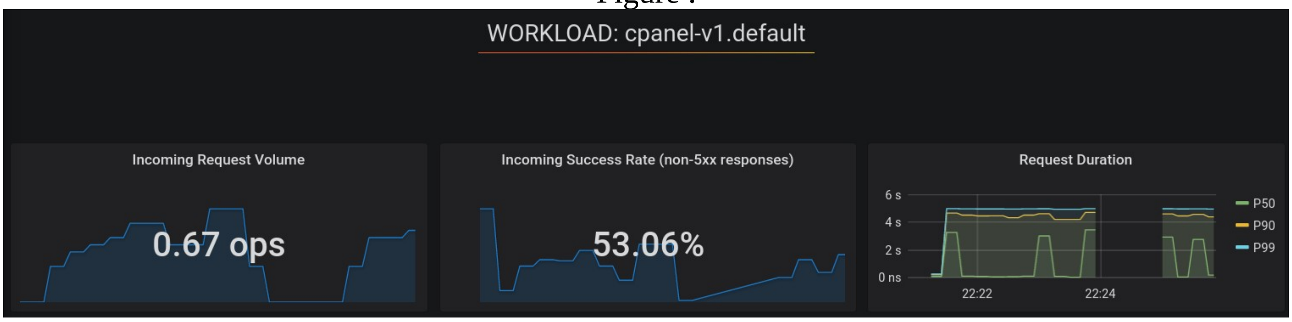


Figure :

$ 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

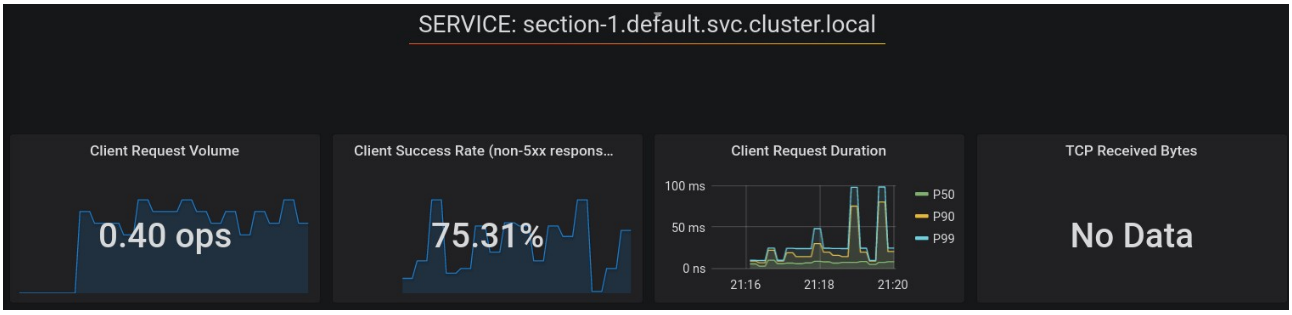


Figure :

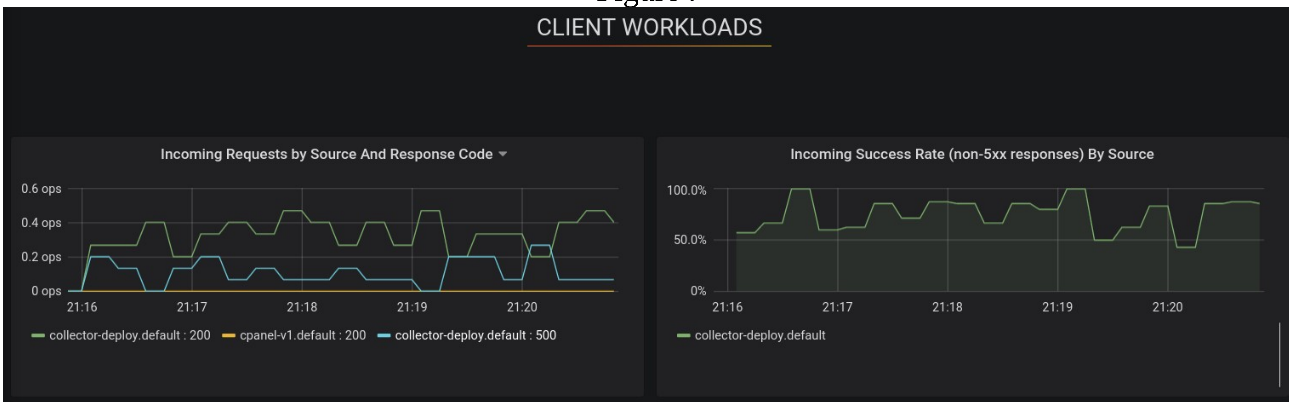


Figure :

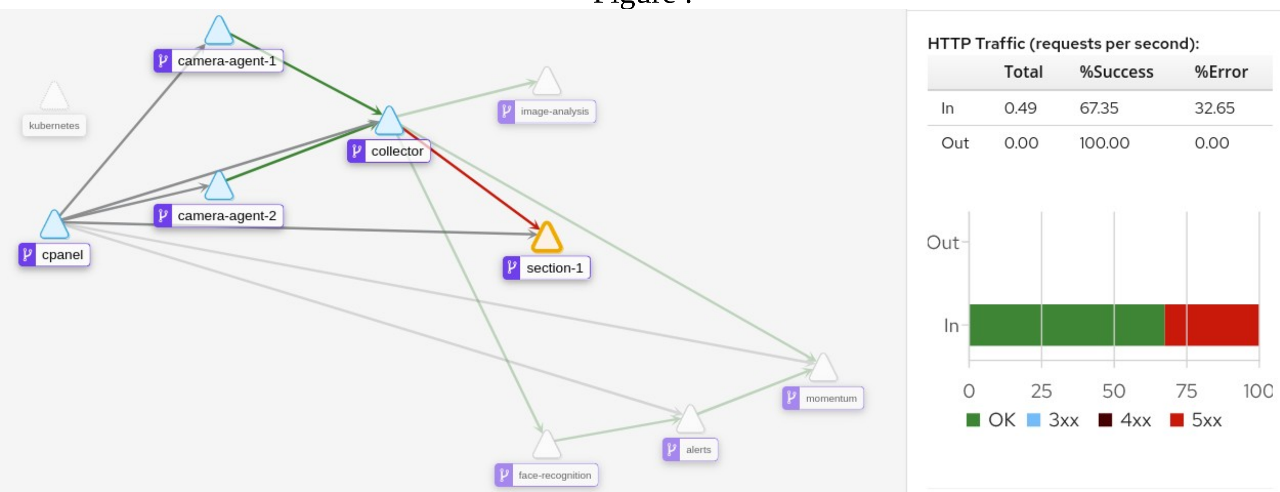


Figure :

$ 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

Circuit breaker

Outlier detection. Collector microservice was scaled to 3 replicas.
$ 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

Different mechanisms of fault tolerance were introduced above. The results of our tests show great usability of Istio features to provide necessary level of resiliency to microservice deployment almost without making any changes to a cloud native application. Some changes were still made because the initial architecture of the application was not cloud native enough. This was fixed during the implementation process.

Application was successfully deployed with Istio. Different resiliency hardening approaches were realized in demo both on side of Kubernetes and Istio configurations. It was interesting to see the next step of application deployment evolution. All the tests were successfully implemented and delivered expected results according to original Istio documentation. Graphics and console outputs show the results of test.

Small amount of microservices used in deployment should be taken in consideration, because real world application will be more complex then this one.

There was also an issue with lack of computing resources. Upgrade of notebook’s RAM was made to provide enough resources.

Need to know internal architecture of Istio.

Debugging was difficult.

Shining of the principle - separation of concerns.

The test platform with installation scripts and Makefile for demonstration purposes is very good suited to try out with other microservices applications as well. It can be proposed as a show case in cloud computing course at the university.

Conclusions and Future Work

Istio offers great features in terms of resiliency for modern microservices applications. It helps to remove operational overhead from developers to operations departments and gives developers more time to focus on business logic of the application.

Centrally managed decentralized system, extra hops - higher latency, immature – not ready for production [enterprise]. Over 1000 opened issues in Github [issues]. resource hungry – x2 containers

- 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.
High learning curve and complexity.
Many moving parts from Istio, Kubernetes, application microservices, Envoy proxies.

Discussed only a small part of Istio functions.
Try out other Istio features and also resiliency with enabled mutual TLS.
Taking a real world application from production and make it more resilient step by step.
Each new version of Istio has plenty of bugs fixed. All of them influence on the stability and resiliency of your deployment. It is up to you to test them, give the feedback and become a part of new standard for future deployments of your microservices applications.
The future is Istio.

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)