



universität
wien

BACHELORARBEIT / BACHELOR'S THESIS

Titel der Bachelorarbeit / Title of the Bachelor's Thesis

„Service meshes as a tool for resilient microservice
deployment on example of Istio“

verfasst von / submitted by

Elcin Bunyatov 01449747

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Bachelor of Science

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

A 033 521

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Bachelor Computer Science UG2002

Betreut von / Supervisor:

Amine El Malki BSc. MSc.
Research Group Software Architecture

Contents

1	Introduction	3
2	Core concepts	5
2.1	Containers	5
2.2	Container Orchestration	6
2.3	Service mesh	7
2.4	Istio	8
2.5	Resiliency	10
3	Project overview and setup	11
3.1	Microservices	11
3.2	Docker & Kubernetes	12
3.3	Setting up Kubernetes & Istio	12
3.4	Deploying microservices	14
3.5	Deploying custom ingress gateway	14
4	Case study	16
4.1	Load balancing	16
4.1.1	Round Robin	17
4.1.2	Random	18
4.2	Rate limiting	18
4.2.1	Global	19
4.2.2	Local	20
4.3	Canary Deployment	22
4.4	Circuit Breaking	24
5	Summary	27
6	Conclusion	28
7	Future work	29

Abstract

The purpose of this paper is to explain the purpose of service meshes on the example of open source Istio project. It will be discussed, when it came to the point that service meshes became relevant and why; briefly review the history of microservice architecture up to the modern days. Then, we will analyze problems that service meshes can solve with focus on resiliency, discuss how they were handled before istio.

Then, on the example of simple distributed microservice app we will configure and use istio features such as load balancing and rate limiting and discuss how they help to solve some common issues. We will compare native kubernetes load balancing with istio and see how different policies work. We will also see how to use different rate limiting strategies to prevent some common security and resiliency issues. Lastly, we will set up istio to run canary deployments and also compare it with native kubernetes setup.

To sum it all up we will review what was learned and briefly discuss the future of service mesh technology.

1 Introduction

Distributed microservice applications has become part of software development for quite a long time. With the lapse of time it slowly replaced 3-tier or architecture for large scale systems like social networks, banking, music streaming and etc. And it makes sense. Mentioned systems tend to scale not only in user base, but also in the amount and quality of data these users consume. Have Facebook or Spotify been monolithic, it would be a monstrous abomination of a software and would require way larger number of people to maintain it. Huge amount of facebook's or spotify's services are developed by third parties and are used via API's that interact and share's data with core data centers. Nonetheless, monolithic approach is not completely gone. Some applications are still monolithic (e.g. some isolated corporate backends).

However, in general, monolithic approach became outdated with development of cloud technologies [1]

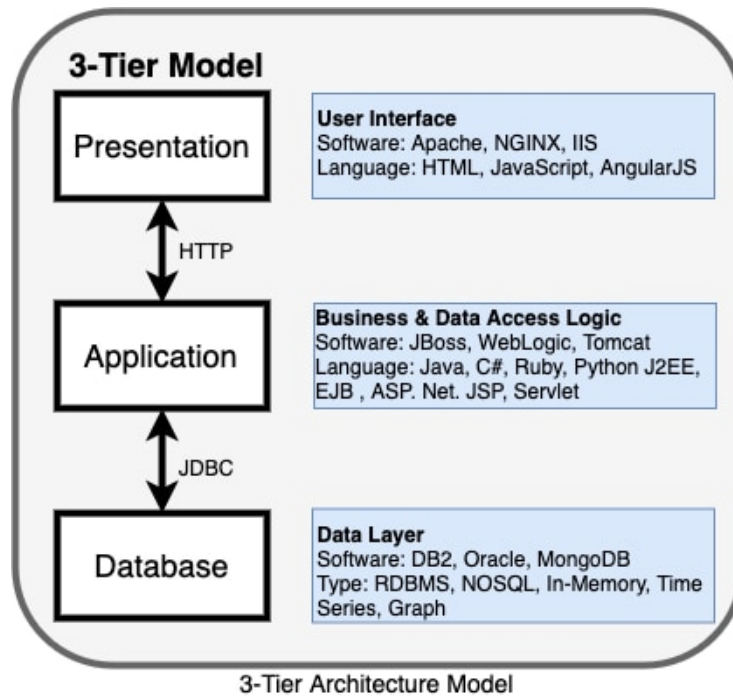


Figure 1: [1]

As seen in the figure above 3-tier model heavily relies on its tiers. It can only scale vertically until certain point. When scaling reaches its critical mass, the system quickly becomes overwhelmed and hence, harder to maintain. Each tier has to be considered.

On the other hand, microservice based system relies less on vertical scaling. It's main strength is and that services are decoupled from each other and can communicate via API, event streams or brokers in cloud . This independence makes it possible to develop each service on the tech stack that is best suitable for the business logic that service is responsible for vastly increasing freedom in choosing technologies, and as consequence making debugging and deployment way easier and faster [23].

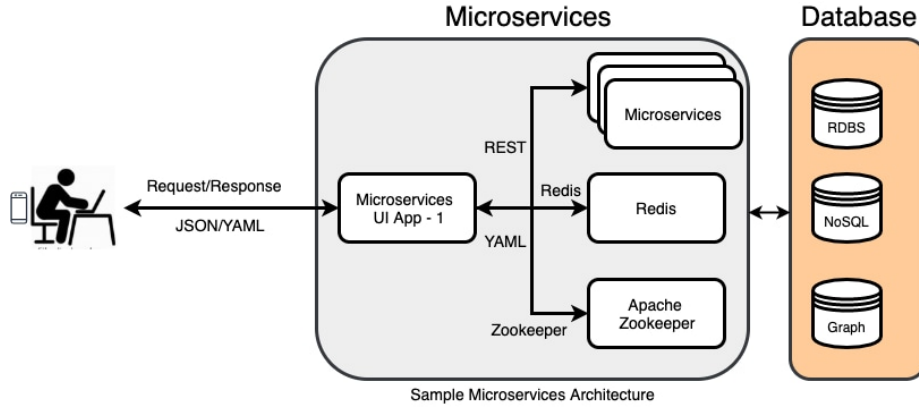


Figure 2: [1]

One of the most significant sacrifices is performance though. Each tier of the monolithic app is located on the same piece of hardware which makes data exchange extremely fast. In case of microservices though, communication over API or brokers means extra level of complexity for data exchange. Additionally, it becomes more and more difficult to manage microservice system as it grows. This growing complexity requires new management approach. This approach is known as DevOps. According to Amazon: "*DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes.* [5]".

However, the scope of our interest lies not in the DevOps itself, but in the tools that DevOps teams use to make this philosophy work. Namely, containerization and container orchestration. And more precisely docker and kubernetes. These two technologies are the solid foundation of this paper and in order to understand the core idea of the papers topic we first need to briefly discuss them in the section below.

2 Core concepts

2.1 Containers

Let's use another direct quote, this time from Citrix: "*Containerization is a form of virtualization where applications run in isolated user spaces, called containers, while using the same shared operating system (OS). A container is essentially a fully packaged and portable computing environment.* [3]"

The backbone of container is a virtual machine (VM). In other words we run a software that starts the VM with the app of our need anywhere we need, without caring about version discrepancy. It really doesn't matter. Containerized app can be run literally anywhere on a cloud or on another computer.

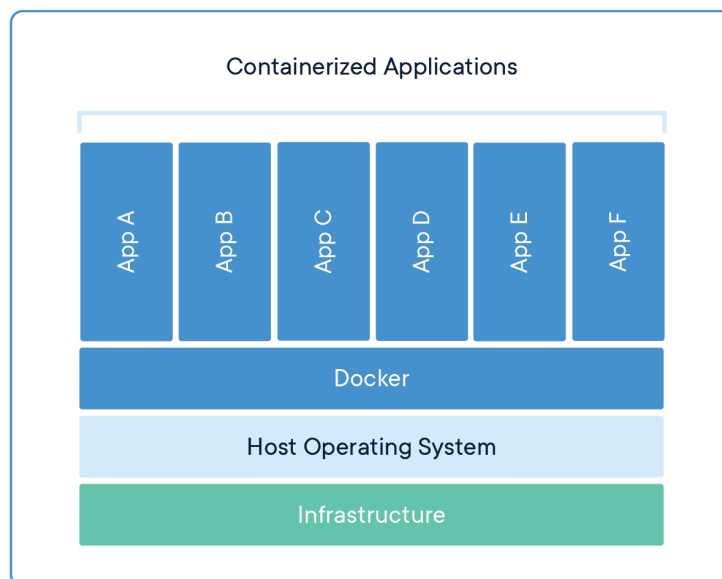


Figure 3: Structure of docker based containerized apps [6]

These containers are lightweight and have everything that our app needs to run: dependencies, runtime libraries, programming language versions and etc. This is great for abstracting our app for further easy deployment in the cloud environment of our choice. This makes containers best friends of microservices [10]. Most of the modern large scale applications use exactly this approach. For the below case study industry standard docker will be used.

However, as time showed, docker alone was not enough. Although it was a huge breakthrough back in time, containers very quickly reached the limit where managing them became a tedious task. Another concept was required to efficiently manage containers in the cloud.

2.2 Container Orchestration

One of the biggest cons of cloud based systems is their complexity. The bigger the system growth, the harder it is to observe it's individual elements and manage their lifecycles. The following aspects should be considered: resource allocation, container availability, container health, load balancing, traffic routing and many other things. [18]. Doing it manually can be very demanding and time consuming. Hence, the concept of container orchestration was introduced.

Main purpose of container orchestration is automation of the above mentioned tasks and providing tools for logging and debugging. In order to achieve, special automation tools are required. One of such tools is an open source container orchestration tool Kubernetes [13]. Basically, it is a framework that gives another higher level of abstraction and control over the system. It allows to dynamically scale apps by adding, removing containers upon need to load balance them and to health check them in order to timely prevent faults.

Kubernetes cluster is where the things happen. It is a set of nodes where our containers are running. The backbone of the entire system. From clusters control pane we have access and can manage system state, schedule container deployments across groups of machines, manage traffic and etc. [2]

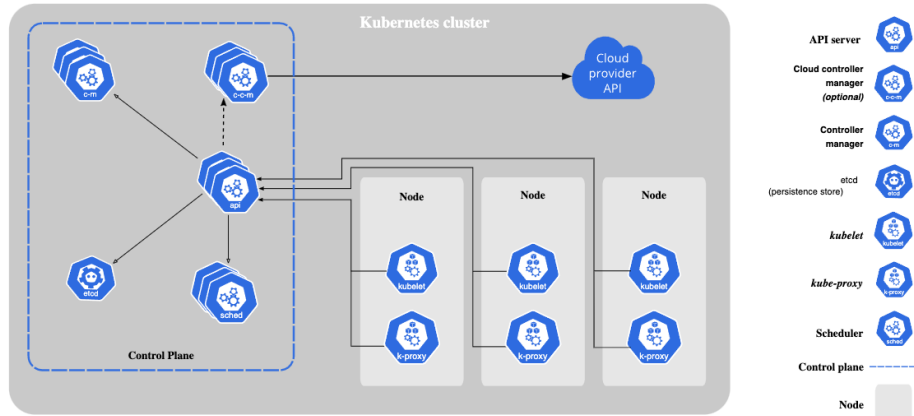


Figure 4: Kubernetes components [15]

However, things can get big even with orchestration. Although, Kubernetes gives us lots of tools for deployment automation, traffic management and security it is still necessary to have some means for observing individual services, their versions or even further fine tune traffic distribution and security. These features are usually implemented on individual service basis [17]. But what if our system has a decent amount of microservices that use different technologies? We will be having hard time adapting each microservice to observe and control the mentioned data. Not only it is necessary to adapt each MS individually, some way to collect and process this data from each of them is also required. Lots of room for errors. But luckily, there is a solution to this situation.

2.3 Service mesh

What is a service mesh? To answer this question let's first try to assume a typical situation, that leads to migration of system to a service mesh.

Imagine some distributed microservice app (for example social network) that is orchestrated by kubernetes. With lapse of time its user base will grow and it'll need to start being scaled. At this stage just kubernetes should be fine to solve scaling issues. But now assume that new features are constantly being added and they require new microservices and even more scaling as user base grows even bigger. It is still possible to manage situation by just using kubernetes features. However, at some point, the app becomes extremely popular and we found ourselves in a situation where hundreds of thousands of replicas of different microservices that are scattered across numerous clusters that interact and share data with each other must be observed, load balanced, further, recovered from failures and etc. A small misconfiguration can possibly lead to many wasted hours of debugging and even worse, could lead to services' downtime that can last longer than acceptable, which, considering the size huge of our application, can cost us some millions if not tenths of millions of outgoings. It could still be acceptable if our profit margin is high, but let's just make a rough simple and approximate calculation.

Let's say that the app brings us 100 millions per year and the maintenance budget is 10 millions (10%). For now, pin this information in our heads and return to scaling our app, namely to the point where we mentioned about wasted hours for debugging and downtimes. As the system gets more complex, the more(complex) kubernetes configs will be required to handle possible failures and faults. Also gathering metrics for each individual microservice will become a mess. It will be required to modify each microservice separately with additional code to retrieve necessary data. And with 100% probability these microservices are not written on just one programming language, since each MS is designed for its specific purpose on a tech stack that is suitable for it the most. Already feels daunting isn't it? And now imagine debugging all of this clutter. It's possible with just kubernetes, but really hard. And complexity will grow exponentially as the app scales. Now, with all we have from above let's unpin the the information about maintenance cost. All that was described above costs us 10 millions per year. Can the costs be reduced? Yes. How? Service mesh. So let's answer the above question now.

As already mentioned, our system is a collection of distributed microservice apps, where each MS is responsible for some business rule that is relevant for the entire system. A service mesh is an additional infrastructure layer that can be added to our services. This layer allows us to transparently add capabilities like observability, traffic management, and security, without adding them to our own code. The term "service mesh" describes both the type of software that is used to implement this pattern, and the security or network domain that is created when that software is used. [11]. Think of it in that way - if a person is a microservice (since we all have some purpose, interact and share data with each other), then our gadgets are service meshes in some way:

1. **Observability:** track our vital signs (e.g. fitness trackers & smart watches)
2. **Traffic management and load balancing:** based on our cars' location traffic lights can be adjusted accordingly to reduce car load on the streets.
3. **Security:** smoke detectors in our houses can call fire brigades in case of fire.

Same with the our microservices and service meshes. Each MS will get a "sidecar" or a proxy that transparently adds above mentioned capabilities. This gives us level of abstraction that is required to reduce the complexity of maintenance routines and drastically save us extra expenses. Now, just few config files are required to affect more than one service with additional capabilities without modifying each of them separately. Huge time and money saver. Since we have basic understanding of what a service mesh is let's see what's let's have a more closer look at Istio.

2.4 Istio

Istio is an open source service mesh project based on EnvoyProxy. It creates additional layer with proxies (sidecars) that are attached to any desired service and allows it utilize features mentioned in the section above. To understand how istio works let us first briefly see what role does envoy play in it.

According to Envoy's own documentation: "*Envoy is a high performance C++ distributed proxy designed for single services and applications, as well as a communication bus and "universal data plane" designed for large microservice "service mesh" architectures.* [7]". So envoy gives us functionality to attach individual proxies to our microservices and provides means to organize these proxies in a mesh. In other words envoy is the skeleton of Istio.

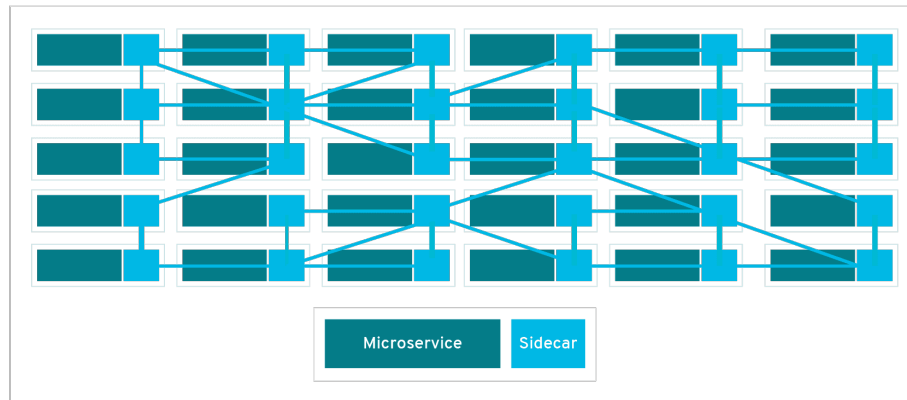


Figure 5: This is how proxies could look like when deployed [16]

Istio has two main components [12]:

1. **Data plane:** is responsible for communication between services. This is basically our mesh deployed via envoy proxies. It makes our network understand what traffic is flowing through it.
2. **Control plane:** is responsible for reading configs and "explaining" them to our network.

This is how a microservice looks before using istio

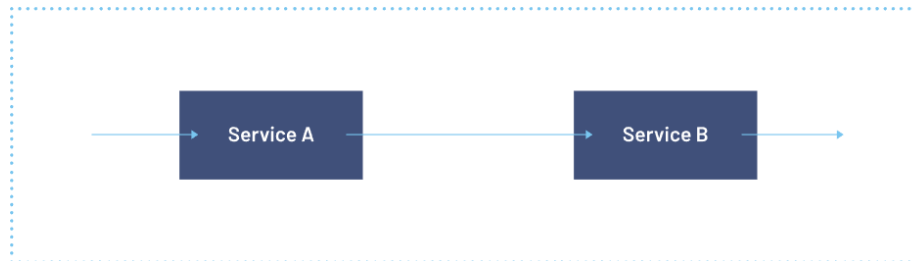


Figure 6: before istio [11]

and after:

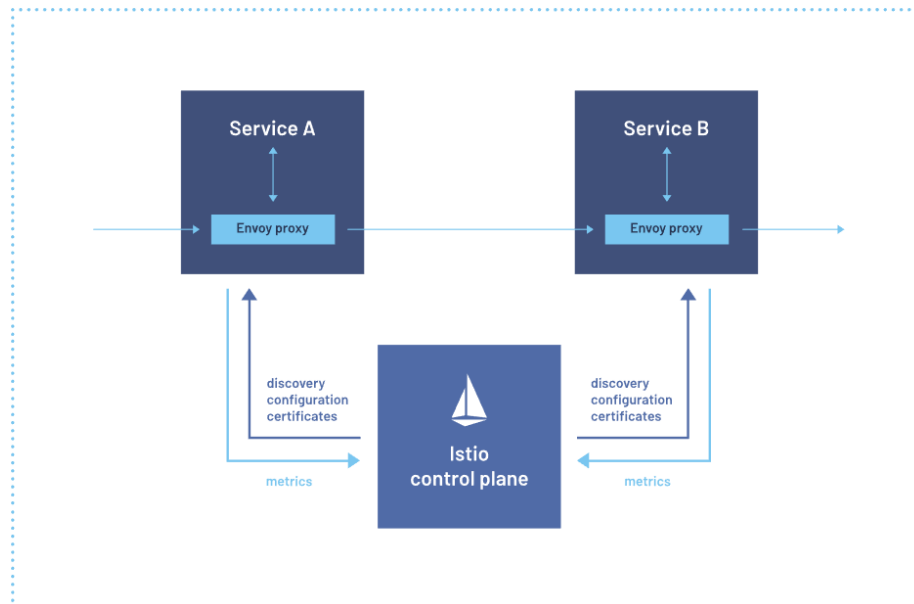


Figure 7: after istio [11]

Now that we have understanding of what service meshes are let's see how resiliency by utilizing Istios' features can be achieved.

2.5 Resiliency

Every system is prone to errors and faults. There is no guarantee that our app will be 100% up. As was mentioned in previous sections microservice approach means that there are more things to pay attention to and these things have to work in unison with one another. The more areas that needs to be controlled, the more places appear where something can go wrong. And there is no way to consider absolutely everything. With so many dependencies and intersections it is only the matter of time until something will go wrong. But it doesn't mean that nothing can be done about it. Yes, it's not possible to think through every single problem, but we can utilize mechanisms that let us observe services and see when and where something went out of hands in order to fix it timely. The goal is to maintain system stability at the highest possible level, as every second of downtime costs money. As was already noted every single failure can't be prevented. But it can be assured that after every failure the system is up and running as fast as possible. This is called resiliency. In order to call the system resilient strong failure detecting and failure recovery mechanisms are required. Istio provides some strategies that can help to maintain necessary amount of resiliency [20]:

1. Load balancing
2. Rate limiting
3. Circuit breaking
4. Timeouts and retries

In scope of this case study the focus will be on the first 3 and on reviewing resilient version deployment via canary approach.

Monitoring tools are great addition for traffic observability of our microservices. By utilizing them we can visualize traffic flow and see where problems occur. Grafana, Kiali, Prometheus, Prometheus and Jaeger are great tools that allow us to observe data.

1. **Prometheus** is monitoring system and time series database for gathering metrics
2. **Grafana** is graphical representation of Prometheus data. It has customizable dashboards.
3. **Kiali** gives us the view of the data flow over our entire mesh.
4. **Jaeger** for distributed tracing

Without further ado let us go through our project setup and studying Istio features.

3 Project overview and setup

In sections below we will go step by step over our test project, its setup and how to utilize Istio configs to simulate real environment situations on a smaller scale. This simplicity will allow us to get general idea of how Istio features work, what they do and how affect the entire system. Each case will be analyzed and compared to real situations in big environments. Each config will be discussed and it will be reviewed how they affect the system with the help of above mentioned monitoring tools. In the end we will sum everything up and discuss how these Istio techniques can be used in conjunction with other techniques that were not covered in details. So let's start.

3.1 Microservices

Our case study will be based on a small internet of things (IOT) app based on processing weather data of 2 types: air temperature and weather condition (sunny, cloudy etc.). The app consists of 3 main microservices that are responsible for the main data flow. There are 2 more microservices (frontend and CLI), but for the scope of our research main 3 will do. Main microservices are:

1. **Weather simulation (WS) service** is responsible for collecting (read simulating) weather data. Has two modes: air temperature and weather condition.
2. **Message Queue (MQ)** serves as an intermediary between IOT and other services. Other services can subscribe to it in order to fetch messages relevant to as soon as these messages arrive in the MQ.
3. **Machine Learning (ML)** MS is subscribed for weather messages in MQ. It processes weather data and returns weather predictions. Can also detect weather anomalies.
4. **GUI app** (irrelevant) is subscribed to all types of messages and serves as weather monitoring interface for weather station staff.
5. **CLI app** (irrelevant) accepts commands for MQ debugging.

All microservices are Spring Boot Java applications. The program is just a simulation of real world weather sensors, which means that all data is randomly generated and used only as foundation for our Istio playground. Machine learning predictions are also based on simple average calculation and random percentage for anomalies. Message queue is a simplified broker service that serves as binding between ML and WS.

3.2 Docker & Kubernetes

Each MS is compiled into a JAR file and wrapped in a separate docker container. For Docker environment simple *openjdk:11* will be used. Docker files will look as simple as:

```
FROM openjdk:11

WORKDIR /usr/app/iot

COPY ./ ./
CMD ["java", "-jar", "simulation.jar"]
```

Also, each MS is deployed to my personal Dockerhub for easier Kubernetes configuration. For local Kubernetes setup Minikube is used. It has everything that is required for deploying our app and testing Istio. I am using *hyperkit* driver, since I had problems setting up Istio with MacOS native *darwinx64* driver. According to my research this problem is known and is currently being fixed.

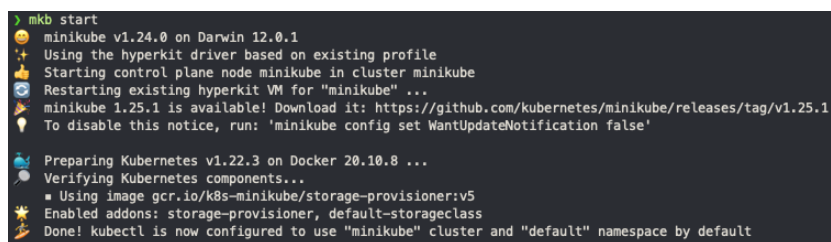
3.3 Setting up Kubernetes & Istio

The project can run on any system (Linux, MacOS, Windows) with recommended minimum of 8 GB RAM. Software requirements:

- Docker (Version for local my setup v20.10.12)
- Minikube (Version for my local setup v1.24.0)
- Postman (Version for my local setup v9.13.2) or curl

After above requirements are met we can proceed with the initial setup:

1. Open the terminal and run `$ minikube start` command. This will start Minikube with default driver for your system and create docker image where our Kubernetes will reside. Successful start should show something like this.



```
> mkb start
minikube v1.24.0 on Darwin 12.0.1
Using the hyperkit driver based on existing profile
Starting control plane node minikube in cluster minikube
Restarting existing hyperkit VM for "minikube" ...
minikube 1.25.1 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.25.1
To disable this notice, run: 'minikube config set WantUpdateNotification false'

Preparing Kubernetes v1.22.3 on Docker 20.10.8 ...
Verifying Kubernetes components...
  ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
  ■ Enabled addons: storage-provisioner, default-storageclass
Done! kubectll is now configured to use "minikube" cluster and "default" namespace by default
```

Figure 8: Successful *minikube start*. (*mkb* is a custom alias for minikube)

To make sure that Kubernetes is up and running execute `$ kubectl version` command. You should see something like that:

```
Client Version: version.Info{Major:"1", Minor:"22",
GitVersion:"v1.22.4", GitCommit:"b695d79d4f967c403a96986
f1750a35eb75e75f1", GitTreeState:"clean", BuildDate:
"2021-11-17T15:41:42Z", GoVersion:"go1.16.10", Compiler:
"gc", Platform:"darwin/amd64"}
Server Version: ...
```

2. Now let's install Istio using projects' getting started guide.¹ This will guide us through downloading and configuring Istio demo setup on the system. For now, the guide needs to be followed until part 2 of *Install Istio* section. In this part namespace label is being added, to instruct Istio to automatically inject Envoy sidecar proxies when we later deploy our application by executing command:

```
$ kubectl label namespace default istio-injection=enabled
```

Without it, Envoy sidecar proxies for each service must be added individually.

3. Lastly, we need to install monitoring tools. To do so, navigate to istio folder and run: `$ kubectl apply -f samples/addons`. This will deploy Kiali, Prometheus, Grafana and some other helpful tools to our cluster.

Now let us make sure that everything was installed correctly by running `$ kubectl get all -A`. In deployment section under `istio-system` we should see our monitoring tools listed.

NAMESPACE	NAME	READY ..
...		
istio-system	deployment.apps/grafana	1/1
istio-system	deployment.apps/istio-egressgateway	1/1
istio-system	deployment.apps/istiod	1/1
istio-system	deployment.apps/jaeger	1/1
istio-system	deployment.apps/kiali	1/1
istio-system	deployment.apps/prometheus	1/1
...		

¹<https://istio.io/latest/docs/setup/getting-started/>

3.4 Deploying microservices

At this point we have our microservices ready and containerized, and our Kubernetes cluster up and running. It is time to deploy our weather simulation IOT app. To do so we need to apply our kubernetes config files. Config files are similar for all 3 microservices and contain settings for Service and Deployment, where service serves as a network gate for our pods. Deployment config will fetch jar file from dockerhub which will start Spring boot app within our cluster. By default, we will have only one replica of each pod. We will increase number of replicas further in load balancing section. To deploy microservices perform following steps:

1. Clone the project from repository ²
2. Navigate to `../configs/IOT/` in terminal
3. Execute `$ kubectl apply -f .` (including dot)

This will deploy all 3 microservices to our cluster. Deployment may take some time. Execute `$ kubectl get pods` to make sure that everything was deployed correctly. The output should look like that:

NAME	READY	STATUS	...
machine-learning-v1-5db8b8b6b-lwpcj	2/2	Running	
machine-learning-v2-6468565977-5lx69	2/2	Running	...
message-queue-5d848945c8-wz2vd	2/2	Running	
simulation-7ff94c55cb-2vmxq	2/2	Running	...

As we can see on the screenshot machine-learning has two instances v1 and v2. This is done on purpose to demonstrate canary deployment in one of the future sections. Also note 2/2 status in READY column by every pod. This indicates that our pod has something attached to them, namely our Envoy proxy sidecar. If we would deploy these configs without installing istio the status would be 1/1. This means that we did everything correct. Alongside with service logs, we now have opportunity to view sidecar logs by just adding `istio-proxy` to our standard log command, but more on this in further sections.

3.5 Deploying custom ingress gateway

By default kubernetes Ingress is responsible for incoming traffic. Istio, however, adds up to this concept and provides extensive customization and flexibility to further fine tune how we want our network traffic to behave. This concept is called Istio Gateway [9]. By default Istio deploys `istio-ingressgateway` which is the global entrance to our mesh of sidecars. We can configure mesh-wide rules and policies for incoming traffic. We can restrict URLs, limit incoming connections and etc. However, we can also deploy our custom gateways for

²<https://github.com/watty888/bachelor-project>

specific service or group of services. For our case we will define such a gateway to allow traffic only to the specific URL of a specific service. To do so we will add a virtual service that specifies the URL and path to service in our cluster. This gateway will also help us to analyze Isitos' rate limiting capabilities. To apply gateway and virtual service navigate to `../configs` directory and execute `$ kubectl apply -f gateway.yml`.

Now, to make an API call to the simulation service, we don't need to access its port anymore. Instead, we need to do it through the ingress gateway port and then specify URL. To access ingress port execute:

```
$ (kubectl -n istio-system get service istio-ingressgateway
-o\newline jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
```

Output of this command is the port we need. Now let's make our first API call. **Host** is our cluster ip. To obtain it run `$ minikube ip`. **Port** is the ingress port obtained earlier above. **Endpoint** is `api/v1/performSimulation?duration=60000&type=weather&threads=1`.

Let's now make a POST call in Postman. To test if everything works, change duration to few milliseconds to make sure that request goes through (status: 200 OK) and endpoint is reachable. Then change it back to 1 minute and run the call again. While request is running let's start Kiali dashboard:

1. In terminal navigate to istio folder
2. Execute `$ istioctl dashboard kiali` and navigate to Graph

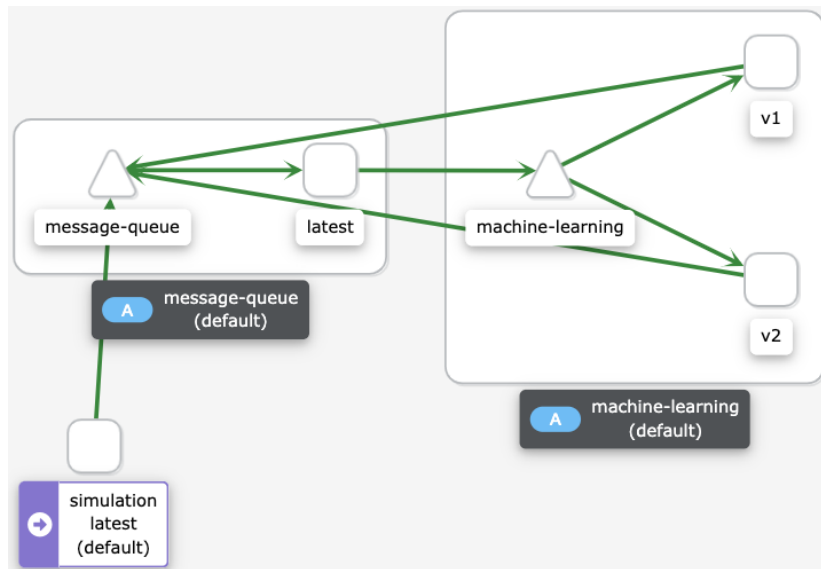


Figure 9: If you see this, then everything is configured correctly

4 Case study

Now that the system is configured, we can proceed with pure Istio stuff. In this section we will apply different Istio configurations and see how they affect the system. Many of the features that we are going to test, as was mentioned in the introduction section of the paper, are also achievable with pure Kubernetes. However, as we will soon see service meshes didn't appear out of nowhere. It happened purely because of limitation that Kubernetes has when it comes to monitoring microservices, gathering metrics and fine tuning some of the security aspects.

We will configure and study following features one by one:

- Load balancing
- Rate limiting
- Circuit breaking
- Canary deployment

Without further ado, let us proceed.

4.1 Load balancing

Load balancing is one of the key issues of microservice based architectures. The need for it appears as the system scales. We need it to properly distribute traffic between our nodes in Kubernetes. But what if we need manage traffic between microservices.

Istio utilizes Envoy's load balancing mechanism based on Destination Rules. Destination rules work in conjunction with virtual services. Let's cite official Istio docu on this ones: "*You can think of virtual services as how you route your traffic to a given destination, and then you use destination rules to configure what happens to traffic for that destination.*" [21]. That is exactly what we need now, except that virtual service is not required for this demos purpose. Default ingress VS will be used automatically.

Let's see what we will be testing in the first place. The POST request to the simulation service will spam our message queue with messages with corresponding labels. This will trigger machine learning service to collect these messages. We are going to load balance this message collection. Namely we will scale our machine learning service so that more than one services subscribe to these messages and see how the traffic will be organized among replicas.

Note that there are two versions of Machine Learning service v1 and v2. Let's scale v1 to 3 replicas and forget about v2 for now:

```
$ kubectl scale deployments/machine-learning-v1 --replicas=3
```

We should now see 5 pods of v1 if we run `kubectl get pods`.

NAME	READY	STATUS	...
machine-learning-v1-5db8b8b6b-f6q7f	2/2	Running	
machine-learning-v1-5db8b8b6b-k75m7	2/2	Running	
machine-learning-v1-5db8b8b6b-qrmzh	2/2	Running	...
machine-learning-v2-6468565977-ppx54	2/2	Running	
message-queue-5d848945c8-sq2jf	2/2	Running	
simulation-7ff94c55cb-ptdf2	2/2	Running	...

We are going to apply destination rule to v1 that defines simple load balancing policies. By default there are 3 policies, but custom ones can also be set up for more specific cases. For now we just need to concentrate on the standard ones, which are:

- ROUND_ROBIN
- RANDOM

To test the policies let us first open 3 parallel terminal tabs and run tracing logs for each of the v1 replicas sidecar proxies:

```
$ kubectl logs --follow machine-learning-v1-5db8b8b6b-<pod-id>
istio-proxy
```

Thereafter, let's load Machine Learning service by starting simulation for one minute in postman:

```
POST http://<minikube-ip>:<ingress-port>/api/v1/performSimulation?duration=
60000&type=weather&threads=1
```

4.1.1 Round Robin

Round robin is the default load balancing policy in istio. Which means, that we don't need to configure anything for it. It is already there.

Now let's observe. We will notice that data started to flow as soon as we switch to our terminal tabs after sending POST request. What we will also note, is that each tab receives connection in a consecutive order. This means that our round robin is working.

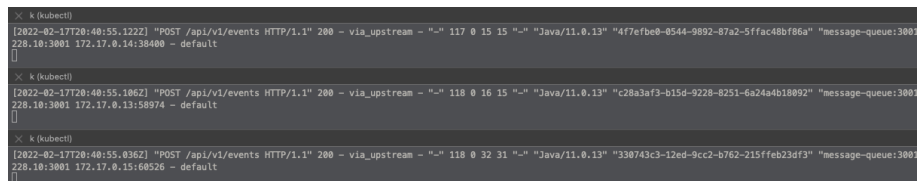
Figure 10: Round Robin request order: 2→3→1→2→3→1→2→3

4.1.2 Random

Random load balancing policy is pretty self-explanatory. Replicas will send receive weather data for processing in a random order. It's often use to evenly distribute requests when load is high. We first need to apply random policy to v1. To do so run from the root folder:

```
$ kubectl apply -f configs/load-balancing/random.yaml
```

Now we can run simulation again and observe logs. We will note now that there is no order anymore in how requests come.



```
k (kubectl)
[2022-02-17T20:40:55.122Z] "POST /api/v1/events HTTP/1.1" 200 - via_upstream - "-" 117 0 15 15 "-" "Java/11.0.13" "4f7efbe8-8544-9892-87a2-5ffac48bf86a" "message-queue:3801"
228.18:3801 172.17.0.14:38400 - default

k (kubectl)
[2022-02-17T20:40:55.186Z] "POST /api/v1/events HTTP/1.1" 200 - via_upstream - "-" 118 0 16 15 "-" "Java/11.0.13" "c28a3af3-b15d-9228-8251-6a24a4b18892" "message-queue:3801"
228.18:3801 172.17.0.13:58974 - default

k (kubectl)
[2022-02-17T20:40:55.836Z] "POST /api/v1/events HTTP/1.1" 200 - via_upstream - "-" 118 0 32 31 "-" "Java/11.0.13" "330743c3-12ed-9cc2-b762-215fdeb23df3" "message-queue:3801"
228.18:3801 172.17.0.15:68526 - default
```

Figure 11: Random request order: 2→1→1→2→3→1→2→2

We can now clean up our configs and proceed to the next part

```
$ kubectl delete -f config/load-balancing
```

4.2 Rate limiting

Rate limiting is a great security technique that can prevent different attack types like brute force, DoS/DDoS, web scrapping and etc. It does so by limiting amount of requests the service can accept. For example in Instagram and Twitter any third-party application that integrates them, can only refresh to look for new tweets or messages a certain amount of times per hour [22]. This makes sense, as otherwise their servers would simply fall apart from the amount of requests per second.

Kubernetes already supports rate limit on node basis, but with Istio we can fine tune it on the service level. In Istio rate limit is controlled our gateway and proxies. There are 2 types of rate limiting in Istio - **global** and **local**. Global rate limit is applied to the entire mesh and restricts connection from the main ingress gateway. Local, on the other hand, is being applied on the local proxy level of specific service. Hence, we can, for example, limit amount of requests our machine learning can handle and save it from overload.

We will now apply each of the strategies to our system and see how it behaves. Let's start with the global strategy.

4.2.1 Global

As mentioned above global rate limiting controls the requests that come from outside of our mesh. Let's set up situation where our system only accepts certain amount of requests per minute to prevent it from hacker attacks. Configuration will proceed in four steps [19]:

1. Applying config map. This is the main config that has instructions to limit amount of requests to Machine learning service (1 per minute)
2. Deploying global rate limit service which implements Envoy's rate limit service protocol based redis but adapted to our needs³.
3. Applying an EnvoyFilter to the ingressgateway to enable global rate limiting using Envoy's global rate limit filter.
4. Applying another EnvoyFilter to the ingressgateway that defines the route configuration on which to rate limit.

Let's apply all 4 configs by executing:

```
$ kubectl apply -f configs/ratelimit/global .
```

Now, run the simulation several times for few milliseconds and start observing:

```
$ POST http://<minikube-ip>:<ingress-port>/api/v1/performSimulation?duration=10&type=weather&threads=1
```

Lastly let's observe istio-ingressgateway log and see what happens:

```
$ kubectl logs pod/istio-ingressgateway-78f69bd5db-dq5vt -n istio-system
----

[2022-02-21T00:47:08.438Z] "POST /api/v1/performSimulation?duration=1&type=weather&threads=1 HTTP/1.1" 200 - via_upstream - "-" 0 2 1387 1331 "172.17.0.1" "PostmanRuntime/7.29.0" "140fd23f-0516-95e8-ac90-79fd440fc20a" "192.168.64.9:30046" "172.17.0.8:3001" outbound|3001||simulation.default.svc.cluster.local 172.17.0.4:53802 172.17.0.4:8080 172.17.0.1:34480 - -
[2022-02-21T00:47:13.654Z] "POST /api/v1/performSimulation?duration=1&type=weather&threads=1 HTTP/1.1" 429 - request_rate_limited - "-" 0 0 4 - "172.17.0.1" "PostmanRuntime/7.29.0" "c204d442-db4d-9b37-97b1-baf9f5e89438" "192.168.64.9:30046" "-" outbound|3001||simulation.default.svc.cluster.local - 172.17.0.4:8080 172.17.0.1:34480 - -
[2022-02-21T00:47:14.576Z] "POST /api/v1/performSimulation?duration=1&type=weather&threads=1 HTTP/1.1" 429 - request_rate_limited
```

³<https://github.com/istio/istio/blob/release-1.13/samples/ratelimit/rate-limit-service.yaml>

```
- "-" 0 0 3 - "172.17.0.1" "PostmanRuntime/7.29.0" "821d1e99-3f27-9b13-bce6-a61020d0a18b" "192.168.64.9:30046" "-" outbound|3001|simulation.default.svc.cluster.local - 172.17.0.4:8080 172.17.0.1:34480 - - [2022-02-21T00:47:15.273Z] "POST /api/v1/performSimulation?duration=1&type=weather&threads=1 HTTP/1.1" 429 - request_rate_limited - "-" 0 0 2 - "172.17.0.1" "PostmanRuntime/7.29.0" "b9da7bd7-69bd-9e5f-97bc-083b50ea47d8" "192.168.64.9:30046" "-" outbound|3001|simulation.default.svc.cluster.local - 172.17.0.4:8080 172.17.0.1:34480 - -
```

As we can see only the first request was a 200 Ok. All the others were blocked because of our 1 request per second policy.

4.2.2 Local

To configure local rate limit on proxy we need to configure an Envoy filter, that enables rate limit for the Machine learning service, which is done by patching the `envoy.filters.http.local_rate_limit` envoy filter into the HTTP connection filter chain. The local rate limit filter's token bucket is configured to allow 10 requests/min. Let's remove previous filters and apply new one:

```
$ kubectl apply -f configs/ratelimit/local .
```

This time we'll need Grafana to better observe internal traffic.

```
$ istioctl dashboard grafana
```

Let's start our simulation like before and check Machine Learning service log:

```
$ kubectl logs service/machine-learning -n default istio-proxy
----
[2022-02-21T01:01:43.323Z] "POST /api/v1/events HTTP/1.1" 429 - local_rate_limited - "-" 0 18 0 - "-" "Java/11.0.13" "f0c4a877-5245-98e2-ad94-a6a343872ddb" "machine-learning:3001" "-" inbound|3001|| - 172.17.0.6:3001 172.17.0.7:51612 outbound_3001_..machine-learning.default.svc.cluster.local - [2022-02-21T01:01:43.309Z] "POST /api/v1/events HTTP/1.1" 200 - via_upstream - "-" 118 0 19 19 "-" "Java/11.0.13" "0fb88228-fa6e-9df6-a93a-1df2fb110cac" "message-queue:3001" "172.17.0.7:3001" outbound|3001||message-queue.default.svc.cluster.local 172.17.0.6:42744 10.109.126.133:3001 172.17.0.6:59572 - default [2022-02-21T01:01:43.307Z] "POST /api/v1/events HTTP/1.1" 200 - via_upstream - "-" 117 0 24 24 "-" "Java/11.0.13" "d3363e76-c5c5-9487-809e-590c89afeb76" "message-queue:3001" "172.17.0.7:3001" outbound|3001||message-queue.default.svc.cluster.local 172.17.0.6:42302 10.109.126.133:3001 172.17.0.6:59570 - default [2022-02-21T01:01:43.357Z] "POST /api/v1/events HTTP/1.1" 200 - via_upstream - "-" 116 0 17 16 "-" "Java/11.0.13"
```

```

"2d35ed7a-3828-98d5-bfa9-d70b53d94e14" "message-queue:3001"
"172.17.0.7:3001" outbound|3001||message-queue.default.svc.cluster.local
172.17.0.6:42302 10.109.126.133:3001 172.17.0.6:59570 - default
[2022-02-21T01:01:45.581Z] "POST /api/v1/events HTTP/1.1" 429 -
local_rate_limited - "-" 0 18 0 - "-" "Java/11.0.13"
"221af186-b92c-9f32-a6e2-75a89a35e3f2" "machine-learning:3001"
"-" inbound|3001|| - 172.17.0.6:3001 172.17.0.7:51652 outbound_
.3001_..machine-learning.default.svc.cluster.local -
[2022-02-21T01:01:51.658Z] "POST /api/v1/events HTTP/1.1" 429 -
local_rate_limited - "-" 0 18 0 - "-" "Java/11.0.13"
"7be56935-2046-9cd6-87f0-bc7af5e55436" "machine-learning:3001"
"-" inbound|3001|| - 172.17.0.6:3001 172.17.0.7:51740
outbound_..3001_..machine-learning.default.svc.cluster.local -

```

From log we can clearly see that our local rate limit is working, since we get our 429s and a `local_rate_limited` message. Let's now check Grafana.

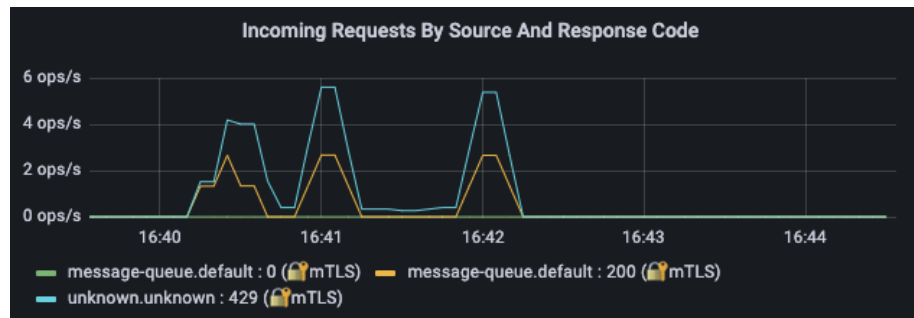


Figure 12: Incoming request into Machine Learning service

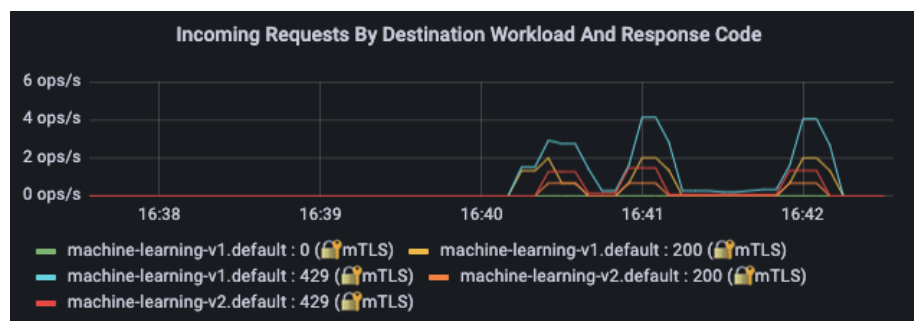


Figure 13: Requests distributed between v1 and v2

On figure 12 we can see requests with status 200 and 429, that were sent to the machine learning service. Figure 13 shows distribution of this requests among pods of the ML service.

4.3 Canary Deployment

Canary deployment is one of the commonly used deployment strategies. In Kubernetes and Istio it is a way to introduce new features gradually. Let's say we added a new feature to our Machine Learning microservice, but we are not yet confident enough that it will work as expected. We decide to release version 2 of the service, but make it slowly. We limit the amount of traffic that version 2 accepts. Doing so will, first of all, allow us to always have stable version 1 for rollback if something goes wrong. As times goes and version 2 proves to be stable, we can increase the amount of traffic towards it and slowly discontinue version 1.

This can be achieved in kubernetes on global, as well as on microservice level. But for microservices it is quickly becomes a tedious task as soon as system becomes big enough, since we will need to take our replicas into consideration. That's where Istio comes to rescue. We can easily control traffic flow between versions of a service by utilizing our sidecar proxies.

In the beginning we deployed 2 versions of Machine Learning service. Let us assume that v1 is our current version and v2 is the one that we want to roll out. First of all we need to delete previous configs.

```
$ kubectl delete -f configs/ratelimit/local .
```

Now let's setup a simple canary deployment using Istio features, namely Virtual Service and Destination Rule. We first need to specify our host service, which is `machine-learning`. Thereafter, we specify 2 destinations. One for each version. By editing the `weight` property we can decide the load for each version. For our test we set 90 for v1 and 10 for v2 respectively. Our Virtual Service should look as simple as:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: machine-learning
spec:
  hosts:
    - machine-learning
  http:
    - route:
        - destination:
            host: machine-learning
            subset: v1
          weight: 90
        - destination:
            host: machine-learning
            subset: v2
          weight: 10
```

Here we defined the host services and specified which versions of it should be affected and with which load.

Next, we are going to define our Destination Rule:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: machine-learning
spec:
  host: machine-learning
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
```

For convenience both of configs should be added into one file and applied:

```
$ kubectl apply -f configs/canary.yaml
```

We can now start our simulation for at least few minutes and see what happens. Run Simulation in Postman run and open Kiali dashboard and go to Graph:

```
POST http://<minikube-ip>:<ingress-port>/v1/performSimulation?
duration=100000&type=weather&threads=1
```

```
$ istioctl dashboard kiali
```

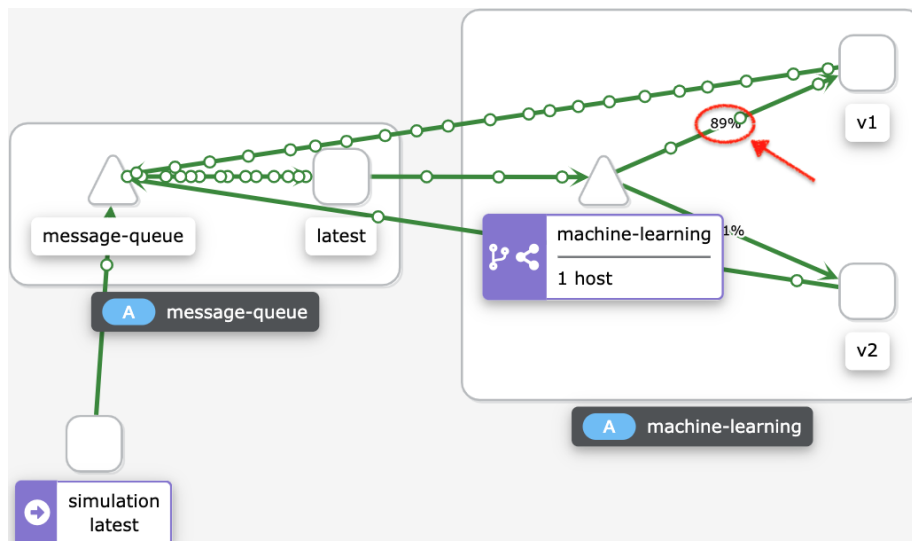


Figure 14: Making sure that our canary settings are working

In the right part it can be seen, that traffic distribution between versions of Machine Learning service is 89 and 11 percents respectively. Due to small fluctuations this values may vary slightly. According to observations the amplitude is around 5-7%. In grafana traffic distribution will look like that:

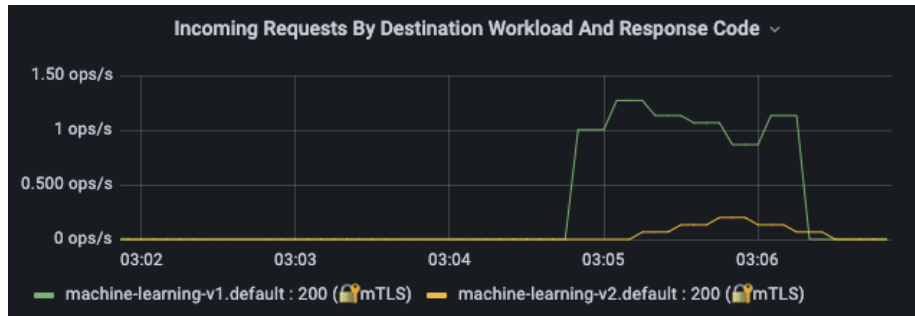


Figure 15: Traffic distribution between versions

Now clean up configs to prepare to the last concept review.

```
$ kubectl delete -f configs/canary.yaml
```

4.4 Circuit Breaking

Circuit Breaking is a popular pattern used in microservice based systems for resiliency. Microservices interact with each other and share data and some of them depend on the others in different ways and grades. It often happens that some service can get overwhelmed by request from one services, so that other services can never reach them. Which wastes resources and can lead to failures or even cascade failures. The job of circuit breaking is to detect such potential failures.

In istio, setting up circuit breaking is as simple as writing of one config. But to test the config it will be necessary to access services inside of the service mesh. To do so we will utilize a simple load-testing client Fortio [8], which is already included in the Istio folder.

```
$ kubectl apply -f samples/httpbin/sample-client/
fortio-deploy.yaml
```

To make sure it worked execute `$ kubectl get pods`. Fortio should be in the list and have status Running.

NAME	READY	STATUS	...
fortio-deploy-687945c6dc-twtvx	2/2	Running	...
...

In this example we will be overloading the Message Queue service to test circuit breaking. First let's prepare our config:

```

apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: mq-circuit-breaker
spec:
  host: message-queue
  trafficPolicy:
    connectionPool:
      http:
        http1MaxPendingRequests: 10
        maxRequestsPerConnection: 3
      tcp:
        maxConnections: 1

```

In this config the number of maximum pending requests is set to 10, which means that every further pending request will be blocked. Also, each connection may handle up to 3 requests. Max connection is the number of consecutive connections to the service. Every excess connection will be considered pending. Now let's apply our config and start testing.

```
$ apply -f configs/circuit-breaker.yaml
```

Next login to fortio pod to be able to send requests through it. We can immediately test if it works:

```

$ export FORTIO_POD=$(kubectl get pods -l app=fortio -o
'jsonpath={.items[0].metadata.name}')

$ kubectl exec "$FORTIO_POD" -c fortio -- /usr/bin/fortio
curl -quiet http://message-queue:3001/api/v1/observations

```

This will produce the following output:

```

HTTP/1.1 200 OK
content-length: 0
date: Sun, 20 Feb 2022 18:10:18 GMT
x-envoy-upstream-service-time: 5
server: envoy

```

which means that fortio is configured correctly. Now let's run the test case. The service will be called with 20 concurrent connections (-c 20) and send 20 requests (-n 20):

```

23:07:47 I logger.go:127> Log level is now 3 Warning (was 2 Info)
Fortio 1.17.1 running at 0 queries per second, 2->2 procs,
for 20 calls: http://message-queue:3001/api/v1/observations
Starting at max qps with 20 thread(s) [gomax 2] for exactly
20 calls (1 per thread + 0)
23:07:47 W http_client.go:806> [16] Non ok http code 503 (HTTP/1.1 503)

```

```

23:07:47 W http_client.go:806> [15] Non ok http code 503 (HTTP/1.1 503)
23:07:48 W http_client.go:806> [7] Non ok http code 503 (HTTP/1.1 503)
23:07:48 W http_client.go:806> [8] Non ok http code 503 (HTTP/1.1 503)
23:07:48 W http_client.go:806> [9] Non ok http code 503 (HTTP/1.1 503)
23:07:48 W http_client.go:806> [6] Non ok http code 503 (HTTP/1.1 503)
23:07:48 W http_client.go:806> [11] Non ok http code 503 (HTTP/1.1 503)
23:07:48 W http_client.go:806> [3] Non ok http code 503 (HTTP/1.1 503)
Ended after 60.266846ms : 20 calls. qps=331.86
Aggregated Function Time : count 20 avg 0.026513552 +/- 0.01763
min 0.004410098 max 0.059791383 sum 0.530271044
# range, mid point, percentile, count
>= 0.0044101 <= 0.005 , 0.00470505 , 10.00, 2
> 0.007 <= 0.008 , 0.0075 , 15.00, 1
> 0.009 <= 0.01 , 0.0095 , 20.00, 1
> 0.011 <= 0.012 , 0.0115 , 25.00, 1
> 0.012 <= 0.014 , 0.013 , 35.00, 2
> 0.014 <= 0.016 , 0.015 , 45.00, 2
> 0.02 <= 0.025 , 0.0225 , 55.00, 2
> 0.025 <= 0.03 , 0.0275 , 60.00, 1
> 0.03 <= 0.035 , 0.0325 , 70.00, 2
> 0.035 <= 0.04 , 0.0375 , 75.00, 1
> 0.04 <= 0.045 , 0.0425 , 80.00, 1
> 0.045 <= 0.05 , 0.0475 , 85.00, 1
> 0.05 <= 0.0597914 , 0.0548957 , 100.00, 3
# target 50% 0.0225
# target 75% 0.04
# target 90% 0.0532638
# target 99% 0.0591386
# target 99.9% 0.0597261
Sockets used: 20 (for perfect keepalive, would be 20)
Jitter: false
Code 200 : 12 (60.0 %)
Code 503 : 8 (40.0 %)
Response Header Sizes : count 20 avg 75 +/- 61.24 min 0 max 125 sum 1500
Response Body/Total Sizes : count 20 avg 171.4 +/- 56.83 min 125 max 241 sum 3428
All done 20 calls (plus 0 warmup) 26.514 ms avg, 331.9 qps

```

From this log it can be observed that out of 20 requests only 12 were successful and 8 were blocked by our circuit breaker, which means our configuration works. One can also specifically check the number of requests that were flagged for circuit breaking by querying the proxy:

```

$ kubectl exec "$FORTIO_POD" -c istio-proxy -- pilot-agent request GET
stats | grep message-queue | grep pending
----
cluster.outbound|3001||message-queue.default.svc.cluster.local
.circuit_breakers.default.remaining_pending: 10

```

```

cluster.outbound|3001||message-queue.default.svc.cluster.local
.circuit_breakers.default.rq_pending_open: 0
cluster.outbound|3001||message-queue.default.svc.cluster.local
.circuit_breakers.high.rq_pending_open: 0
cluster.outbound|3001||message-queue.default.svc.cluster.local
.upstream_rq_pending_active: 0
cluster.outbound|3001||message-queue.default.svc.cluster.local
.upstream_rq_pending_failure_eject: 0
cluster.outbound|3001||message-queue.default.svc.cluster.local
.upstream_rq_pending_overflow: 1021
cluster.outbound|3001||message-queue.default.svc.cluster.local
.upstream_rq_pending_total: 3371

```

As can be seen from the log 10 requests were flagged.
Let's now clean configs and proceed to summary:

```

$ kubectl delete -f circuit-breaker.yaml
$ kubectl delete -f gateway.yaml
$ kubectl delete -f IOT/

```

5 Summary

As can be seen from the above tests, Istio is a pretty comfortable tool for use and also a very powerful one. Although, our example was based on a small microservice system, similar configurations could easily be applied to a much bigger scale and with even deeper fine tuning. As was already mentioned almost all of the problems that were covered can be solved either on Kubernetes, or on microservices level, but will require way more man hours and hence more costs. The above examples illustrate, that with simple configs one can configure complex behaviours without almost any system overloads, as well as maintaining good level of resiliency.

Let us sum up what was learned through this study:

- How cloud based applications are being built in modern days
- What are service meshes and how do they help software engineers to save time and money
- How to use Istios' load balancing feature to manage traffic between our replicas in a more convenient way
- How to apply two rate limit strategies: global for the entire mesh and local for the exact service
- How the canary deployment can be realized using Istios' virtual services and destination rules
- How circuit breaking can prevent services from cascade failures and increase resiliency

6 Conclusion

Each of the described Istio features, especially when combined in a proper manner, can give developers exceptional tool set for solving modern day Microservice and Cloud Computing problems. And we didn't even cover other features and concepts. What have been covered is just a slice of traffic management features and a little bit of security. Istio has other powerful concepts like: ⁴

- **Security** - Authorization and Authentication
- **Observability** - Telemetry and Monitoring features
- **Extensibility** - Web Assembly plugin system

Istio is a very young technology and evolves constantly. Each major update brings new features that further extend developers' arsenal. It is the logical consequence of software engineering's evolution. The systems get larger and more complex, hence new techniques that handle this complexities come to life. Such changes require automation, scalability and continuous delivery. These are the most valuable aspects that have to be considered while evolving. [14]

In the beginning, microservices were a great alternative to the hard to scale monolith architecture. However, with the lapse of time, even they did not escape the latter's fate. And here's a good example: Such a complex network itself

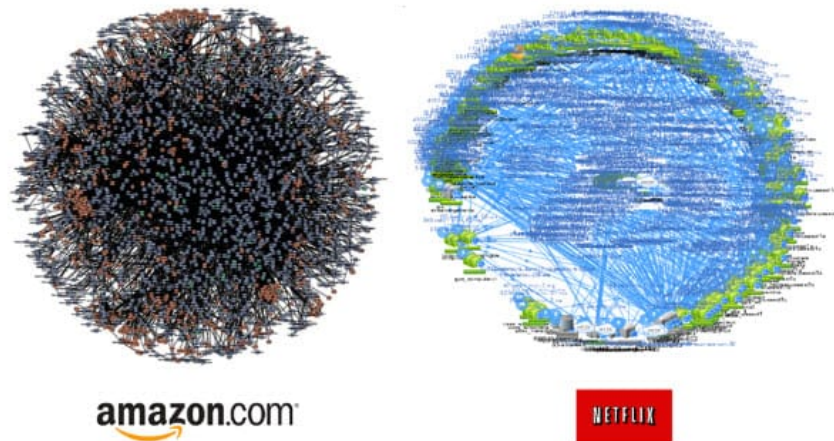


Figure 16: Microservice Death Star by Amazon and Netflix [4]

requires some sort of management and we are not speaking about Kubernetes here. The higher level of control is required. A good analogy would be the complexity of programming language levels (from low to high). Each level is the logical result, the evolution, the higher level of abstraction above it's predecessor.

⁴<https://istio.io/latest/docs/concepts/>

Each has more high level control and less low level. Like C has more control over memory than Java, Kubernetes has more control over microservices than Istio. But, in conjunction with one another they make it possible to build more complex applications.

7 Future work

If I would do further research on the topic, I would definitely love to try service meshes on a real life customer deployed system, whether it is a new setup or some maintenance project. It would be nice and useful to see, how these features function on a way larger scale, to collect real life metrics, see security features in action etc. As was mentioned in conclusion it is really intriguing where the time will bring us. Little hardware development stagnation is over and technologies are rapidly improving again. It is only the matter of time until we will come to the point, where the next complexity level jump occurs. And I wonder where this changes will bring us. Who knows, maybe with the lapse of time even service meshes will become so complex that the new solution will have to appear to manage it. It would be really interesting to theorize on this matter.

References

- [1] <https://www.ibm.com/cloud/blog/four-architecture-choices-for-application-development> (accessed on 22.01.2022).
- [2] <https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-cluster> (accessed on 27.01.2022).
- [3] <https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html> (accessed 23.01.2022).
- [4] <https://dzone.com/articles/navigating-the-microservice-deathstar-with-deployh>.
- [5] <https://aws.amazon.com/devops/what-is-devops/> (accessed on 25.01.2022).
- [6] <https://www.docker.com/resources/what-containers> (accessed on 25.01.2022).
- [7] <https://www.envoyproxy.io/> (accessed on 27.01.2022).
- [8] <https://github.com/fortio/fortio> (accessed on 14.02.2022).
- [9] <https://istio.io/latest/docs/tasks/traffic-management/ingress/ingress-control/> (accessed on 05.02.2022).
- [10] <https://cloud.google.com/learn/what-are-containers> (accessed on 27.01.2022).
- [11] <https://istio.io/latest/about/service-mesh/> (accessed on 22.01.2022).
- [12] <https://istio.io/latest/about/service-mesh/#how-it-works> (accessed on 28.01.2022).
- [13] KOSCHEL, A., BERTRAM, M., BISCHOF, R., SCHULZE, K., SCHAAF, M., AND ASTROVA, I. A look at service meshes. In *2021 12th International Conference on Information, Intelligence, Systems Applications (IISA)* (2021), pp. 1–8.
- [14] KRATZKE, N., AND QUINT, P.-C. Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software* 126 (01 2017), 1–16.
- [15] <https://kubernetes.io/docs/concepts/overview/components> (accessed on 27.01.2022).
- [16] <https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh> (accessed on 28.01.2022).

- [17] MILLER, S., SIEMS, T., AND DEBROY, V. Kubernetes for cloud container orchestration versus containers as a service (caas): Practical insights. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2021), pp. 407–408.
- [18] <https://www.redhat.com/en/topics/containers/what-is-container-orchestration> (accessed on 27.01.2022).
- [19] <https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit/> (accessed on 10.02.2022).
- [20] <https://istio.tetratelabs.io/blog/istio-resiliency-patterns/> (accessed on 02.02.2022).
- [21] <https://istio.io/latest/docs/concepts/traffic-management/#destination-rules> (accessed on 05.02.2022).
- [22] <https://www.cloudflare.com/learning/bots/what-is-rate-limiting/> (accessed on 10.02.2022).
- [23] VILLAMIZAR, M., GARCÉS, O., CASTRO, H., VERANO, M., SALAMANCA, L., CASALLAS, R., AND GIL, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (2015), pp. 583–590.