universität wien

# BACHELORARBEIT / BACHELOR'S THESIS

Titel der Bachelorarbeit / Title of the Bachelor's Thesis

„Observability and Resiliency in Mircroservices using Istio service mesh“

verfasst von / submitted by

Hakob Harutyunyan 01408703

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

Bachelor of Science

Wien, 2021 / Vienna, 2021

| | |
|---|---|
| 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 of Computer Science |
| Betreut von / Supervisor: | BSc. MSc. Amine El Malki |

# Contents

**Abstract**

Microservices architecture is an architectural pattern that allows separating a monolithic application into a set of loosely coupled, independent, interconnected services, each having its scope of responsibility. A growing base of microservices in a distributed system means a growing amount of interactions between them and increasing requirements. It becomes a challenge to manage the infrastructure across all of the abstractions the microservices are deployed on, as well as the communication between them. Managing the traffic across the infrastructure abstraction layers microservices deployed on, as well as applying security and observability layers on it becomes a big challenge for the developers, especially taking into account that those components have to be implemented on the client side of each microservice.

Service meshes provide a dedicated infrastructure layer for handling service-to-service communication, as well as applying and configuring security and observability patterns. With a growing number of businesses adopting cloud-native approach for the development of their distributed systems, service meshes gain higher relevancy and significance. One of the implementations of a service mesh is Istio.

In this paper, we will focus on a discussion and case-study demonstration of Istio's features and concepts, the ways it can be used to apply patterns that improve such quality attributes of a microservices architecture as resilience and observability and also discuss cloud-native development.

# 1 Reliable microservice architecture

In the context of microservices architecture resiliency, high availability and observability are among the terms one hears quite often. And that is obvious, since dealing with unexpected failures, or identifying bottlenecks or recovering from them in order to have zero downtime becomes harder in a microservices based environment. The reason for the increased complexity often resides in the multiple sources of failures: multiple instances of multiple microservices, unreliable network, infrastructure issues.

## 1.1 Resiliency

In a cloud-native distributed environment with numerous microservices and a big load of traffic it should always be assumed that partial failures will certainly occur at some point. The ability of a system or a microservice to recover from partial failures refers to resiliency. The resiliency of microservice-based applications depends on how well they handle intercommunication over an unreliable network[3].

The patterns belonging to the resiliency area are:

- Load balancing[1] is a core component that can help to increase the throughput and lower the latency. A load balancer can be used to distribute the load between instances of a given service and provide better availability as a result.
- Timeout[8] is a pattern that makes clients avoid long/unbounded waiting times for a response from a server, i.e. hanging requests that can lead to e.g. unresponsive UI by failing the within the set timeout. The famous "fail fast" paradigm of microservices architecture makes bugs and system vulnerabilities show up faster and timeout is a great pattern consolidating this paradigm. In more sophisticated scenarios in the case of a timeout, the request is not simply failed, but there is also a fallback request to another service or a retry operation.
- Retry pattern[2] is useful in cases when there are several instances of a downstream service, but one of them either takes unusually long to process (failed, if the timeout is used) the inbound request, or there was an exception in it. After the upstream service becomes aware of the failure, it sends another request to the service and depending on its own client-side retry configuration or load-balancer this request would go to another instance of the service, which has a higher chance of successfully processing the request.

[1] quite often load balancing is included in the resilience patterns, however in my opinion it is fairer to say that it improves availability and not resiliency, since load balancing by itself does not act in the processes of recovering from partial failures, but participates in a running distributed system. But since our goal is to demonstrate Istio's capabilities in the case study section, it makes sense to include them.

- Circuit breaker[9] is a pattern resembling an electrical circuit breaker, which trips for some time when encountering a number of failures. For the configured duration each consecutive request fails without invoking the remote calls. After that duration, the circuit breaker starts allowing requests and checks whether they succeeded or failed. If they fail as well, then the circuit breaker trips again.

## 1.2 Observability

Observability is another key component of a reliable and healthy system. Observability in microservices is a component that enables development teams to easily identify and troubleshoot potential issues/bottlenecks or inspect the behavior within a scope of an individual microservice, as well as a network of microservices as a whole. There are quite a lot of patterns and tools facilitating observability, which allow developers to implement relevant observability patterns. There is no single way to do it, but there is always a need for a certain degree of observability especially if we are working in a production environment with dozens of microservices deployed.

Among others, there are such observability patterns as:

- Metrics

  Metrics represent quantitative measures about different components not only of any given application, but also of the infrastructure, meaning e.g. container runtime and Kubernetes metrics. Unlike the health check API metrics bring a passive way of observation, whereas health checking is about regular checks whether an application is in a running (up) state or failed (down) state. There are not many qualitative conclusions that can be made of a simple health check and a developer would usually need to investigate further the reasons of a failure by accessing logs, etc. Adding metrics scraping of meaningful data about the system would allow developers and system architects to plan up- or downscaling or decide to split a microservice into separate microservices. Thus, metrics are not only helpful for detecting failure sources, but also for improving the system architecture and availability.

- Distributed tracing

  Cloud native approach, orchestrator tools' ability of automatic horizontal up and downscaling made the distributed nature of microservices architecture more complex to comprehend for the developers. Thankfully there have been several tools developed in response to the need of observing the requests as they propagate through the distributed environment. We achieve full end-to-end visibility by introducing distributed tracing to the picture. Distributed tracing does not only facilitate observability, but also resilience, since we can easily identify the source of a failed response, instead of going through the application logs.

- Health check API

  Health check is a proactive method of testing a deployed running service. A service should expose a health check API endpoint, which gets called by a monitoring service (e.g. Prometheus), load balancer, or service registry. The health information can be monitored by or passed to the orchestrator's (Kubernetes) cluster and based on that the orchestrator tool might decide to restart the pod. Kubernetes has liveness and readiness probes that address health check endpoints of services[4].

# 2 Cloud-native

There is no clear definition for the term "cloud-native", since the consensus about the characteristics of the cloud-native applications is still growing in the IT industry[5] and is not fixed yet. The terms cloud-native application and microservices are often used interchangeably by mistake. Microservices are just one of the architectural patterns that can be used when building cloud-native applications. Cloud-native is a term that encompasses various tools, techniques and methodologies of utilizing cloud services for dynamic and agile microservice application development. The applications designed with the cloud native focus leverage the advantages of Platform as a Service (PaaS) infrastructure meaning the automation features like scaling, resizing and automated recovery. Cloud makes it easier to implement automation in the infrastructure, as well as those components that sit above it (e.g., firewalls and storage). Although it may seem like an expensive investment, it will pay off in the long run. This and other aspects of planning a cloud native architecture and development need a systematic approach. An industry-wide accepted methodology for the development of cloud based applications is the 12-Factor Application.

## 2.1 The 12-Factor application principles

The set of 12 best practices provides a sophisticated and established way of developing a cloud-native application. Some of the factors might be considered obvious and usual best practices, but they all together form the cloud-native approach. Following is the list of 12 factors[7]:

1. Code base

   A cloud-native application is stored in a single code base, that will be deployed multiple times and is stored in a version control system. Test, staging and production environments should be in place.

2. Dependencies

   This principle of 12-Factor App states that storing external artifacts, such as .jar files together with the source code in the source control systems should be avoided, i.e. they should be isolated from the source code. Those artifacts should be referenced in the dependencies manifest. In the case

of Java there are tools like Gradle and Maven - dependency management tools which fit perfectly for this purpose.

3. Backing services

   Backing resources such as databases should be treated as attached resources. It should be possible to change one database to another with a configuration change.

4. Config

   The application might have multiple deployments (test, staging, production) and each of them might require a connection to different systems with different credentials. Important is that the configurations to the systems are externalized into the environments. A common way of doing this is storing the security-sensitive configurations in environment variables.

5. Build, release, run

   There should be a strict separation between build, release and run phases, meaning that no configuration change is executed in the run phase. Every time a configuration change is needed, a new release is built, released and run.

6. Process

   The application should consist of one or more stateless processes. It should not rely on a process being available from one request to another. If a client makes 20 requests, the assumption is that each request is handled by a separate stateless process. The server-side state associated with a client can be stored in an external datastore, in case of a follow up request, there is no impact on the client.

7. Port binding

   HTTP applications that expose services via port bindings should not rely on being installed into a web container. They should have a HTTP server dependency, that opens a port during startup.

8. Concurrency

   The application should be able to scale different parts independently if needed. A good example of such a situation would be 2 different services running behind separate load balancers. It would be possible to scale up/down each of them without affecting the rest of the services.

9. Disposability

   The application should be disposable, i.e. the process should quickly start, shut down and cope with termination. This makes sure the application scales out well and provides resiliency for unexpected failures, since the process can be quickly restarted from the last release.

10. Dev/prod parity

    There should be no difference between development and staging environments. Although that is sometimes hard to achieve, the goal should be to always try to minimize the difference, since a good part of problems come up because of those differences. This can be achieved by using the same tools in both deployment contexts, effective collaboration inside of the team, configured CI/CD pipelines with the focus on avoiding the disparities.

11. Logs

    The application should write logs in the process output instead of writing to the file system. The execution environment will forward the process output to the final destination for storage and viewing.

12. Admin processes

    Admin processes should run as one-off processes as part of application deployment. It has to be ensured that the scripts for e.g. database migrations are automated and not executed manually before the release of a new application version.

# 3 Service mesh

A service mesh is an infrastructure layer that controls the communication between services in a distributed system. It solves various challenges in the security, traffic control, telemetry functional areas linked with microservice architecture. Those challenges are not news to the IT industry and they used to be and still are successfully implemented in the application code of separate services. However, it is not trivial to implement and manage them, since it is more error-prone and requires additional effort from the teams to properly test their implementations. Service meshes provide a variety of features that allow you to manage network traffic between your applications. With Istio SREs or developers can extend the routing rules to route traffic based on application versions. This feature can be useful in A/B testing pattern where the endpoint the request will be routed to depends on the client's subnet. The big advantage of applying configurations within a service mesh is that there is no need for the redeployment of the services, since the configurations are applied on the go without interfering with the application code. Thus, service mesh allows development teams to concentrate on the business logic and forget about infrastructure concerns to some extent.

# 4 Comparison of service mesh implementations

## 4.1 Linkerd

Linkerd[6] is a service mesh that is managed by the Cloud Native Computing Foundation. It is one of the first service mesh implementations, developed by two ex-Twitter engineers. It is the second most popular service mesh after Istio. In the second iteration of Linkerd, it has been enhanced to provide more advantages that a service mesh can offer. It still lacks in the feature set compared to Istio, although it has a more gentle learning curve and provides more out of the box configurations for its features. It is not Envoy proxy-based, but instead uses a more lightweight and simple "micro-proxy" - Linkerd2-proxy. Kubernetes is a prerequisite for using Linkerd, i.e. it does not support VMs.

| Advantages | Disadvantages |
|---|---|
| Very lightweight | No ingress gateway included |
| Active community base | Lacks delay injection, circuit breaking |
| Very easy setup | Does not have the same proxy extensability as Envoy based meshes |

Table 1: Consul connect advantages and disadvantages

## 4.2 Consul

Consul connect also uses Envoy proxies on the data plane level. The control plane leverages Consul's KV store (used for dynamic configuration, feature flagging, coordination, leader election, etc.). Like Istio it injects Envoy proxies to

the application pods' containers. Consul connect provides features like level-seven traffic routing, load balancing, security capability, such as basic mTLS. It is possible to have Consul integrated with Vault (secrets and encryption management tool), which can be used as a certificate authority for the cluster.

| Advantages | Disatvantages |
|---|---|
| Support for all kind of virtual machines besides Kubernetes | Poor documentation, mostly redirecting to Envoy proxy documentation |
| Strong enterprise community | Requires effort to set up observability capabilities: dashboards, metrics. |
| Extremely lightweight | |

Table 2: Consul connect advantages and disadvantages

# 5 Istio

Istio allows developers to manage a distributed network of deployed services with no changes in the source code. A single exception from this rule is the distributed tracing feature of Istio. In order to have working distributed tracing there are a few code changes needed, which will be demonstrated in our case study. Istio is platform-independent, so it can run in several environments like:

- Kubernetes
- Cloud
- On-premise
- Mesos

Istio service mesh is logically split into two modules:

- Control plane
- Data plane

Istio takes advantage of the Envoy proxies deployed together with each pod and intercepts all inbound and outbound traffic. The set of the Envoy proxies comprises the data plane of Istio. Control plane is comprised of a single binary called Istiod which provides service discovery, configuration and certificate management. The rules that Envoy proxies (data plane) must follow while routing the traffic are injected by the Istiod component (control plane)[1].

Historically the control plane contained several components deployed independently (Galley, Pilot, Mixer and Citadel), thus, complying with a microservice architecture. Since version 1.5 was released in March 2020 the control plane architecture was consolidated to a single Istiod component forming a monolith architecture. This step simplified the usage of Istio in general improving the istioctl CLI tool, integration with CI systems, automated mTLS configuration. The following figure provides a simplified view of the architecture of the latest Istio version.

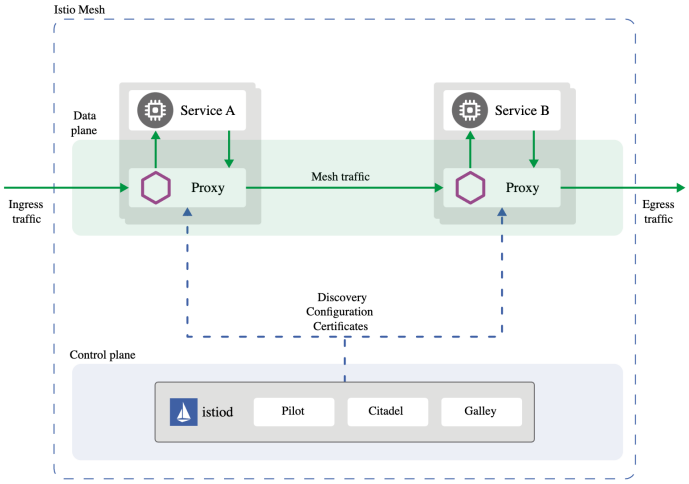


Figure 1: Istio architecture[1]

# 6 Case study

## 6.1 Application based on microservices architecture

A simple microservices application was developed to demonstrate a set of features of Istio service mesh. The application consists of 5 services:

- Bank client app - simulates a backend of a client app for a mobile banking app
- User validator service - simulates a service that validates a payload of a user registration
- Customer rating service - simulates a service that rates the registered user based on registration details
- Master data service - simulates a service that persists the data in the database
- SMS notification service - simulates a service that sends out an SMS notification to a phone number user has entered during registration

There is a single flow of registration implemented. The communication between the services in case of a successful registration flow is shown on the UML sequence diagram below.

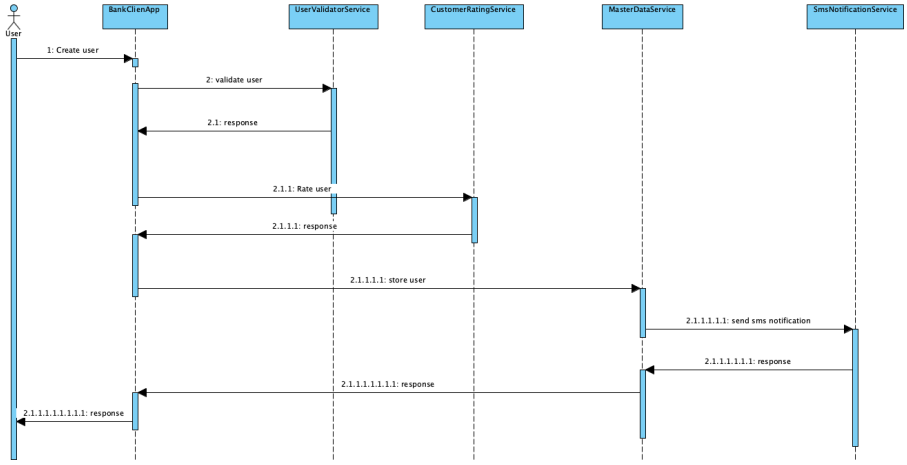


Figure 2: UML sequence diagram of user registration flow

## 6.2 Technology stack

The technlogy stack used for the application implementation includes and the case study:

- Java 11
- Spring Boot
- H2 Embedded database
- Maven
- Docker
- Kubernetes
- Minikube
- Istio

## 6.3 Deployment

The microservices are deployed in a local Kubernetes cluster using Minikube[2] - a cross-platform tool that allows running a lightweight Kubenetes cluster. It creates a VM on the local machine with a single node. Minikube requires a

[2]https://minikube.sigs.k8s.io/docs/

container or virtual machine manager installed for the VM to run and kubectl to interact with the cluster.

Start minikube with:

```
$ minikube start
```

In the deployments.yaml file the deployments and services are defined that will be created in the cluster. Each service's deployment configuration is defined by a similar principle: 1 pod and 1 service per deployment.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bank-client-app
  labels:
    app: bank-client-app
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: bank-client-app
      version: v1
  template:
    metadata:
      labels:
        app: bank-client-app
        version: v1
    spec:
      containers:
        - name: svc
          image: hakob21/bank-client-app:latest
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: bank-client-app
  labels:
    app: bank-client-app
spec:
  ports:
    - port: 80
      targetPort: 8080
      protocol: TCP
  selector:
    app: bank-client-app
```

Figure 3: Snippet of the YAML file with the deployment and service configurations

When applied with

```
$ kubectl apply -f deployments.yaml
```

and checking afterwards with

```
$ kubectl get all
```

it can be confirmed that all the pods and services are up and running

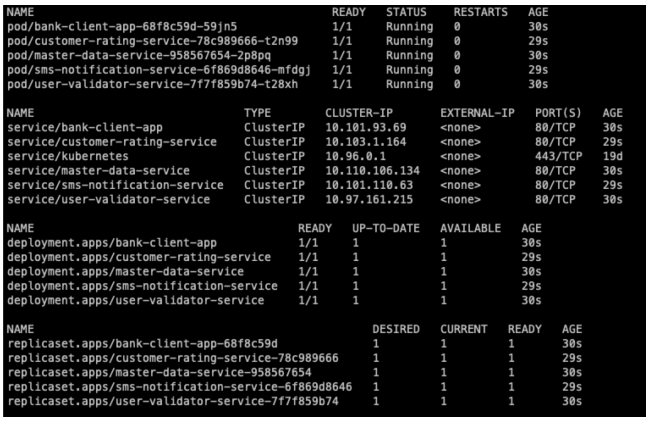


Figure 4: Running pods and services inside Kubernetes cluster

## 6.4 Installing Istio

Istio can be downloaded by

```
$ curl -L https://istio.io/downloadIstio | sh -
```

add istioctl to the path

```
$ export PATH=$PWD/bin:$PATH
```

At this point, Istio can be applied to the Kubernetes cluster via

```
$ istioctl install --set profile=demo -y
```

and with

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

Istio will be instructed to automatically inject Envoy sidecar proxies for each application deployment.

To check if the setup is active we can delete and reapply the deployments and service of the deployments.yaml file

```
$ kubectl delete -f deployments.yaml
```

```
$ kubectl apply -f deployments.yaml
```

Checking the deployments, pods and services

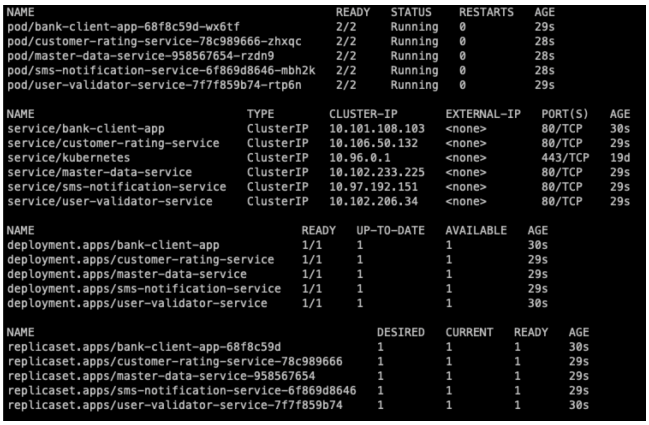


Figure 5: Running pods and services inside Kubernetes cluster after installing Istio

As we can see the difference in the deployments with and without Istio is that now there are 2 pods running in each container

## 6.5 Ingress gateway

Before moving to the main part of the case study, i.e. demonstrating observability and resiliency features of Istio, we would configure an Istio ingress gateway in order to send HTTP requests to the service APIs via Postman or curl commands. It will allow external traffic into the cluster and serve as a load balancer.

Although it is possible to configure an Ingress gateway using solely Kubernetes, it is preferable to have an Istio Ingress gateway, because this gateway is yet another Envoy proxy therefore it can be configured to use Istio's traffic splitting, security, or other features. Also, it will send out metrics, which can be viewed in the observability tools.

By default Istio deploys an istio-ingressgateway service, which is of type LoadBalancer with a public IP address to the mesh. We have to note it is running inside of a Minikube cluster on our machine, so it will not have an external IP address specified if we check it with the following command:

```
1   $ kubectl get svc istio-ingressgateway -n istio-system
```

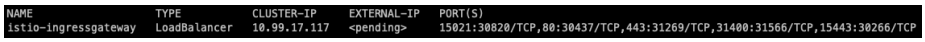


Figure 6: Ingress gateway external IP is pending

In order to create a route to the gateway from our machine we have to run

```
1   $ minikube tunnel
```

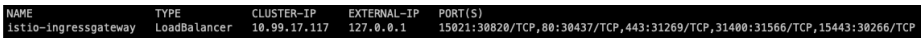


Figure 7: Ingress gateway external IP is known

Now that we have the entry point to our cluster, we can try to access it with curl command performing a GET request

```
1   $ curl -X GET http://localhost/
```

We receive the following response


Figure 8: Gateway routing failure

The reason for this message is that our gateway does not have any routing rules in place and does not redirect the incoming request to any service.

We create a gateway resource and a virtual service with the routing rules for the gateway.


Figure 9: Gateway resource

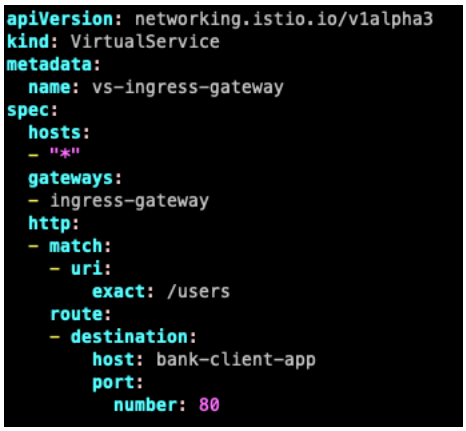


Figure 10: Virtual service for routing

The routing rules in the virtual service match any host requesting "/users" endpoint to the target bank-client-app Kubernetes service from where it gets caught by our application and the registration flow starts. Eventually, we get the JSON response with the created user details.

```
1   $ curl -X POST http://localhost/users -H 'Content-Type: application/json' -d '{"
        firstName": "FirstName1", "lastName": "LastName1", "phoneNumber":
        "06601111111", "citizenship": "Citizenship1"}'
```

```
~ curl -X POST http://localhost/users -H 'Content-Type: application/json' -d '{"firstName": "FirstName1", "lastName": "LastName1",
"phoneNumber": "06601111111", "citizenship": "Citizenship1"}'
{"firstName":"FirstName1","lastName":"LastName1","phoneNumber":"06601111111","citizenship":"Citizenship1","rating":1.0}%
```

Figure 11: Successful response to the POST request

Now that we are able to make requests to client service API through the Ingress Gateway, it is time to dive into Istio's capabilities for implementing resilience and observability features within our service mesh.

## 6.6 Observability

Istio's generated telemetry types are the following:

- Metrics - a set of metrics based on the four "gold signals" of monitoring (latency, traffic, errors, saturation) are generated by Istio. The metrics for the mesh control plane are also provided[1].

  Metrics can be visualized in Grafana[3] - multi-platform analytics and interactive visualization tool.

- Access logs - Istio provides a full record of each request flowing into a mesh and into the services through Envoy proxies. They provide an option to monitor and gain understanding about the behavior from the individual workload perspective.

- Distributed traces - distributed trace spans are generated by each Envoy proxy.

Istio integrates with a variety of visualization and scraping tools. The ones that will be used in the evaluation are:

- Grafana[4] is a tool that allows visualizing the metrics of services within the mesh on flexible dashboards, setting alerts on events, etc. It integrates with Prometheus to visualize the scraped data.

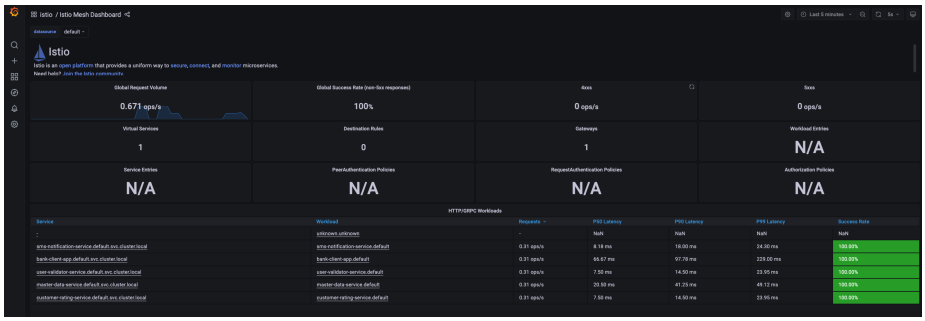


Figure 12: Grafana's Istio mesh dashboard created by default with Istio installation

- Prometheus[5] is a monitoring system and time series database. It allows querying metrics data about the service mesh.

- Kiali[6] - visualization tool for service to service communication. Provides detailed metrics and basic Grafana integration

[3] https://prometheus.io/
[4] https://grafana.com/
[5] https://prometheus.io/
[6] https://kiali.io/

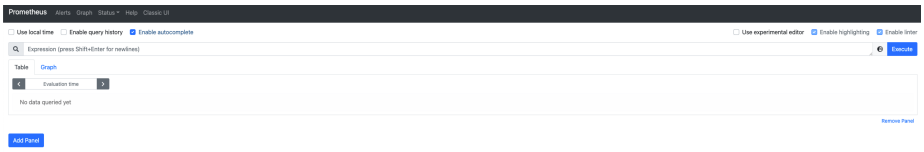


Figure 13: Prometheus main page

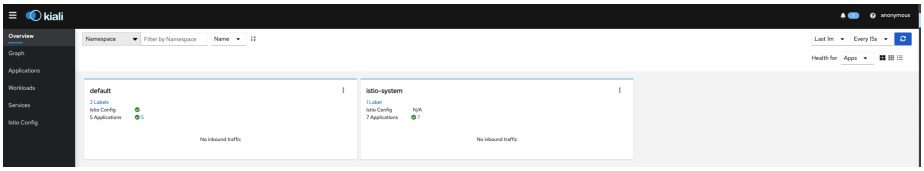


Figure 14: Kiali overview page

The Envoy proxies can automatically send spans to the tracing backend service (e.g. Jaeger), however in order to build meaningful full traces they need a bit fine-tuning on the applications' side. This is the only case where an intervention into application code is needed. Applications need to propagate the following headers from and to any request flowing throw them.

- x-request-id
- x-b3-traceid
- x-b3-spanid
- x-b3-parentspanid
- x-b3-sampled
- x-b3-flags
- x-ot-span-context

By default in the production mode only 1% of traces are recorded, or rather sent to Jaeger, or any other tracing backend service. But since we set demo profile during Istio installation, it will record all of our requests by default. The following figure shows a basic trace view of a request in Jaeger.

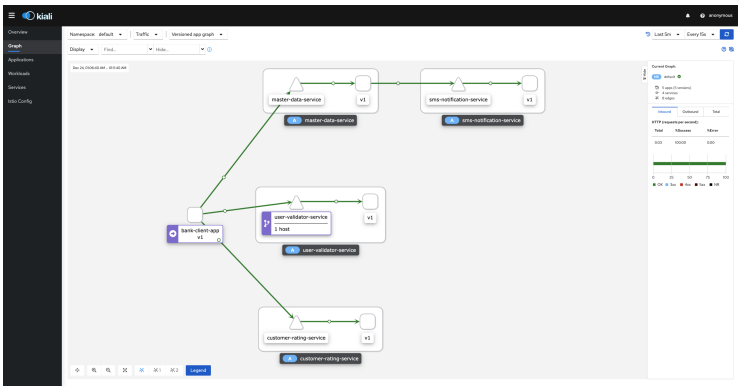


Figure 15: Kiali's visualization of mesh communication

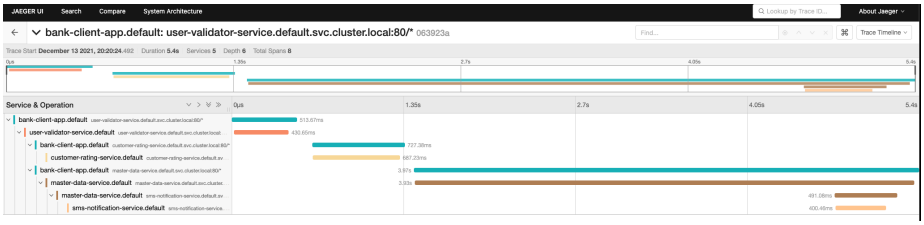


Figure 16: Trace of a request in Jaeger UI

## 6.7 Resiliency

### 6.7.1 Client-side load balancing

A straightforward way to implement load balancing capability is to have a centralized load balancing server that all of the clients can use to distribute the load across all of their systems. However, this method can be very costly and can often lead to a bottleneck, as well as become a single point of failure. A good way to improve overall performance and lower latency is distributing load balancing capabilities to clients. Istio's sidecar proxies provide the following client-side load balancing algorithms out of the box:

- ROUND_ROBIN

  Evenly distributes the load in a cyclic manner across all replicas

- RANDOM

  Evenly distributes the load across all replicas without order

- LEAST_CONN

  The least request load balancer uses an O(1) algorithm which selects two random healthy hosts and picks the host which has fewer active requests.

The load balancing algorithm in Istio's context is set using the DestinationRule configuration. We will set the LoadBalancer's algorithm to ROUND_ROBIN and check it using Jaeger's distributed traces. Another prerequisite for executing this scenario is having multiple pods of one of the services. We will configure the Deployment of user-validator-service to have 3 replicas instead of 1 it had before and apply the deployments.yaml file again.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-validator-service
  labels:
    app: user-validator-service
    version: v1
spec:
  replicas: 3
  selector:
    matchLabels:
      app: user-validator-service
      version: v1
  template:
    metadata:
      labels:
        app: user-validator-service
        version: v1
    spec:
      containers:
        - name: svc
          image: hakob21/user-validator-service:v1
          ports:
            - containerPort: 8080
          imagePullPolicy: Always
```

Figure 17: Deployment configuration of a service with 3 replicas

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: user-validator-service
spec:
  host: user-validator-service.default.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
```

Figure 18: Load balancing configuration in DestinationRule resource

We make 6 consecutive user registration requests in total to the bank-client-app. Ultimately we validate that the requests follow ROUND_ROBIN algorithm based on the pod IP addresses of the user-validator-service's span in Jager. The results are:

| | |
|---|---|
| Request 1 | 172.17.0.15 |
| Request 2 | 172.17.0.10 |
| Request 3 | 172.17.0.14 |
| Request 4 | 172.17.0.15 |
| Request 5 | 172.17.0.10 |
| Request 6 | 172.17.0.14 |

Table 3: Request number and IP address of a pod

Thus, we successfully tested that the ROUND_ROBIN algorithm is working.

### 6.7.2 Fault injection

The timeout pattern will be applied for the requests flowing from the bank-client-app to master-data-service. But before applying the timeout pattern we would need to inject an artificial delay to the communication between services. Istio has a fault injection feature, which is usually used in chaos testing - the process of breaking services on purpose in order to achieve exposed vulnerabilities of the system. Fault injection is performed in a VirtualService resource. In this case, we inject a 7-second long delay between master-data-service and sms-notification-service. We also have the option to set the percentage of inbound requests we want the delay to be applied to (100% in this case).

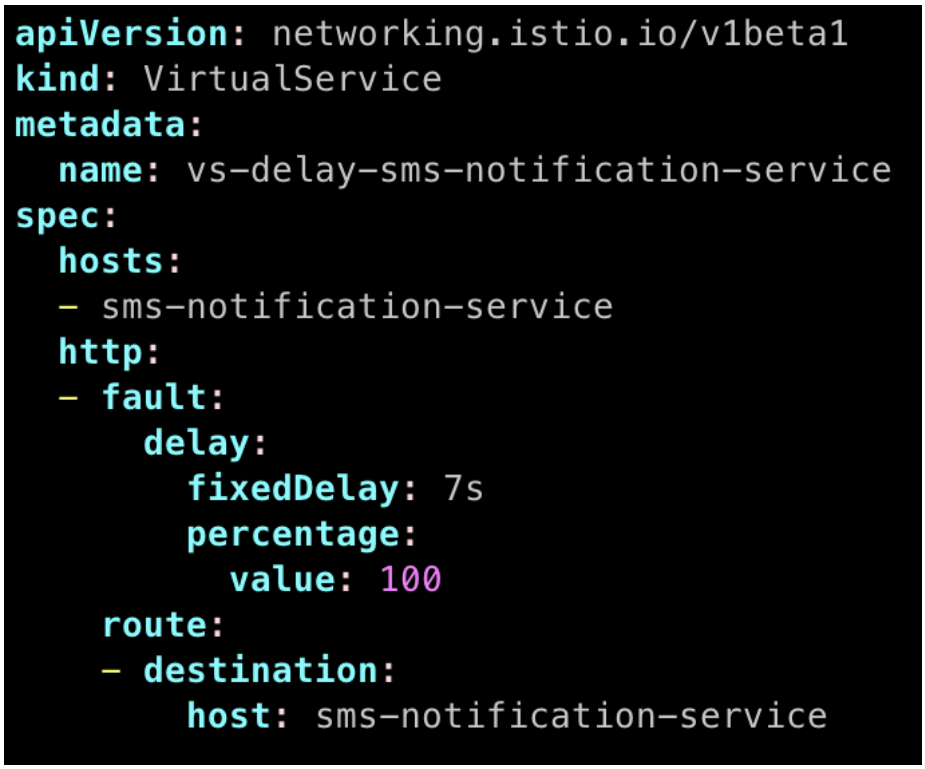


Figure 19: Istio fault injeciton configuration

### 6.7.3 Timeout

Having the artificial delay in place, we apply the second VirtualService resource with the timeout configuration.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-timeout-master-data-service
spec:
  hosts:
  - master-data-service
  http:
  - route:
    - destination:
        host: master-data-service
    timeout: 0.5s
```

Figure 20: Timeout configuration

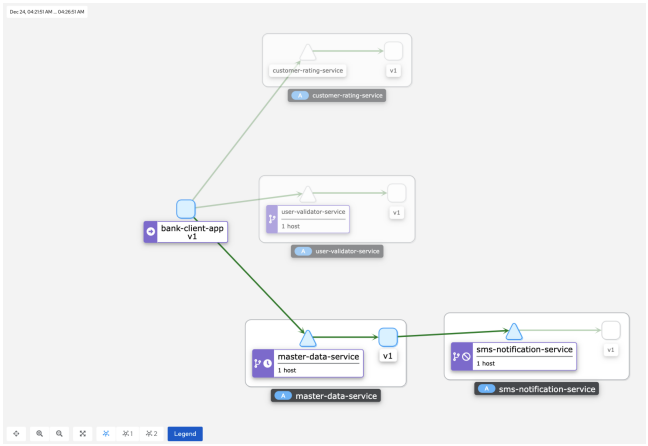


Figure 21: Kiali mesh visualization with timeout and fault injection icons indicated on services

To test if the timeout is in place, we would POST another user registration request to the BankClientApp API. We know that at some point of execution flow one of the requests hits the master-data-service and then sms-notification-service. Since the request to sms-notification-service is delayed, the sidecar Envoy proxy of master-data-service returns a response with 504 Gateway Timeout status code to the BankClientApp. For demonstration purposes, the bank-client-app is implemented in such a way that it will simply redirect this response to us. As we see in the lower right corner of Postman the status code is 504, which indicates a running timeout configuration.

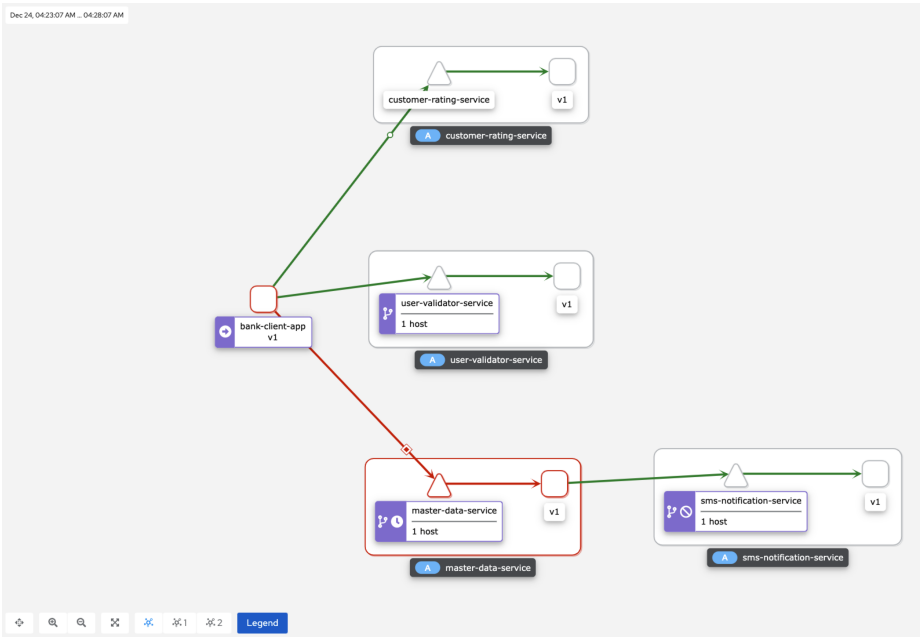


Figure 22: Kiali visualization of failed requests due to timeout

Figure 23: Response from user-validator-service in Postman

### 6.7.4 Retry

For the user-validator-service we have an exposed endpoint "/validate-possible-bad-gateway" which with 50% chance returns a 502 BAD GATEWAY HTTP response code to the client. The retry has the following configuration. The envoy proxy would make 5 attempts of hitting a successful response if it hits a 502 BAD GATEWAY HTTP response. We have also configured a 1-second timeout tolerance.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-retry-user-validator-service
spec:
  hosts:
  - user-validator-service
  http:
  - route:
    - destination:
        host: user-validator-service
    retries:
      attempts: 5
      perTryTimeout: 1s
      retryOn: gateway-error
```

Figure 24: Retry configuration

The request trace can be checked in Jaeger. As we see there have been 2 additional attempts after the original one has failed and the last attempt was responded by an HTTP status code 200

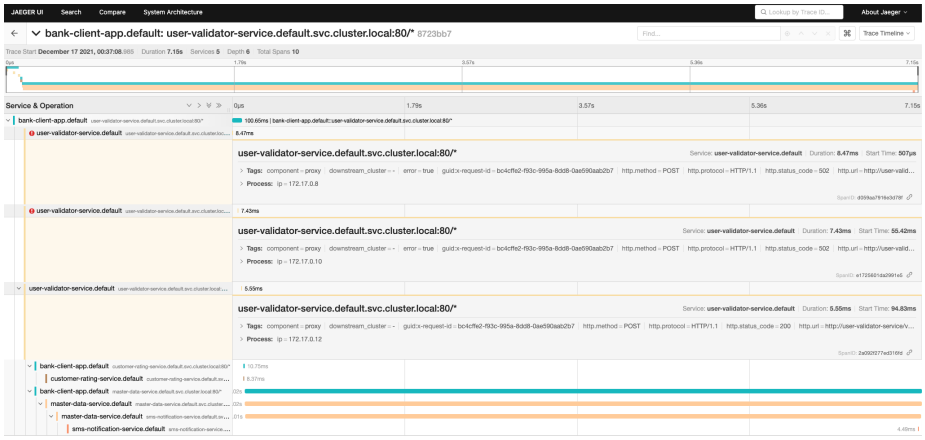


Figure 25: Request traces in Jaeger

### 6.7.5 Circuit breaker

The circuit breaker is configured in a DestinationRule resource of Istio. The rules for the circuit breaker are specified in the trafficPolicy section. There are two configuration components for the circuit breaker DestinationRule - connectionPool and outlierDetection. In ConnectionPool section the maximum number of pending requests, retries and timeouts can be set. The requests exceeding the set threshold will be tripped by the circuit breaker. In outlierDetection we can set a threshold for the number of failed requests. If the threshold is exceeded for a given pod, then that pod is ejected from the connection pool. Also, we can set the minimum ejection duration and the percentage of pods we allow to be ejected.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: dr-user-validator-service
spec:
  host: user-validator-service
  trafficPolicy:
    connectionPool:
      http:
        http1MaxPendingRequests: 1
    outlierDetection:
      consecutive5xxErrors: 3
      interval: 10s
      baseEjectionTime: 15s
      maxEjectionPercent: 100
```

Figure 26: Circuit breaker configuration

In order to test the connectionPool part of the circuit breaker configuration we will first execute two asynchronous requests to an endpoint of user-validator-service exposed for testing purposes with the following combination of curl commands.

```
$ curl -I "http://localhost/get-test" & curl -I "http://localhost/get-test"
```

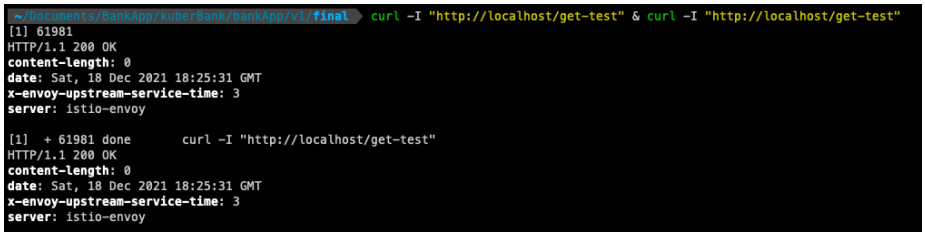


Figure 27: Request and headers of response from user-validator-service without circuit breaker

We see that both requests received 200 OK responses. Now we will apply the DestinationRule in order to configure the circuitBreaker and run both requests again.

```
$ curl -I "http://localhost/get-test" & curl -I "http://localhost/get-test"
```


Figure 28: Request and headers of response from user-validator-service with circuit breaker

As we see one of the requests was tripped by the circuit breaker, thus, proving that the circuit breaker is intact.

To test the outlierDetection part we would make four consecutive requests to the bank-client-app's "/users-bad-gateway" endpoint, which will make bank-client-app send a request to the user-validator-service's "/validate-bad-gateway" endpoint which always responds with 502 BAD GATEWAY error. The expected behavior is that after the third consecutive failed (502 bad gateway) response, Istio will eject the single user-validator-service pod, thus making all the consecutive requests fail with 503 SERVICE_UNAVAILABLE response.

This behavior can be validated by checking the logs of both services.

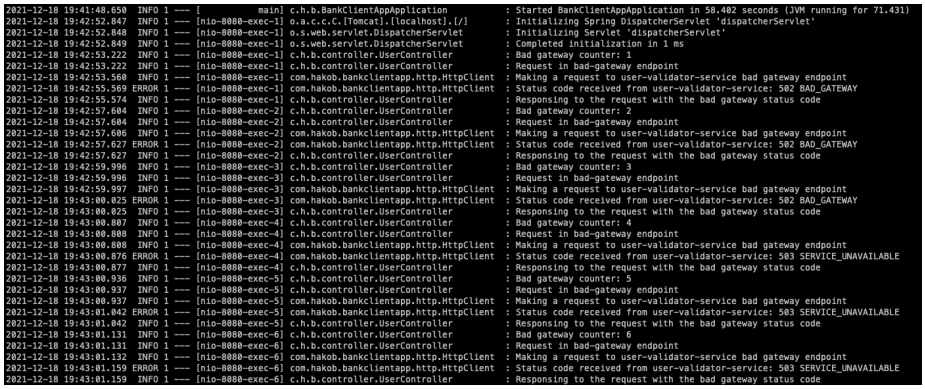


Figure 29: Logs of bank-client-app

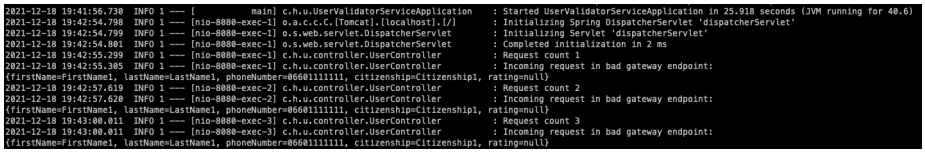


Figure 30: Logs of user-validator-service

# 7 Conclusion

The concept of a service mesh was introduced and discussed in this bachelor's thesis. The main focus was Istio and its resiliency and observability features, however, other competitor service mesh implementations exist in the market, which we compared without giving preference to any of them, since for various scenarios and organizations, there might be more sense in using a solution of another vendor. As discussed in the paper and demonstrated in the case study, the features and gentle learning curve of Istio (at least in the scope of core features for resiliency and observability) provides an attractive option for organizations to modernize, secure, monitor and manage communication for their emerging or existing microservice architecture based applications. Migrating to cloud-native can also be a good reason to start looking toward Istio, since the whole idea of service mesh - taking standard functionality of microservices down to the infrastructure layer, allowing developers to focus on business logic complies with the cloud-native principles quite well. There is no longer a need of integrating, maintaining and developing SDKs for the applications, taking into account different programming languages.

The future of Istio seems to be bright, based on the rapid pace of innovating, growing ecosystem of projects building on top of Istio and the highly active community thriving to make Istio easier to use and as transparent as possible[10].

# 8 Future work

Even though keeping the scope of the thesis to resilience and observability features of Istio allowed for a concise, yet detailed enough discussion, it should be noted that these are only a rather small subset of features provided. Specifically, the security area has a wide range of policies and interesting scenarios worth analyzing.

The application developed for the case study is a simple application serving only the purpose of evaluation of Istio's features. However, an application developed following the 12-Factor application rules could provide a better view of the benefits of the cloud-native approach and consequently of Istio.

The architectural details of Istio were also touched in the thesis, however a deeper analysis of Envoy proxies, xDS API could be a nice addition and provide an insight into the "engine" of Istio.

# References


[1] Istio architecture. https://istio.io/latest/docs/ops/deployment/architecture/.

[2] How to make services resilient in a microservices environment. https://dzone.com/articles/libraries-for-microservices-development, 2018 June.

[3] Building resilient microservices. https://conferences.oreilly.com/software-architecture/sa-ca-2019/public/schedule/detail/75217.html, 2019.

[4] Microservices monitoring with health checks using watchdog. https://medium.com/aspnetrun/microservices-monitoring-with-health-checks-using-watchdog-6b16fdae0349, 2020 April.

[5] Cloud native: A strategy for the future of business. https://www.cio.com/article/189118/cloud-native-a-strategy-for-the-future-of-business.html, 2021.

[6] Dury, G. A kubernetes service mesh comparison. https://www.toptal.com/kubernetes/service-mesh-comparison, 2019.

[7] Emily Jiang, Andrew McCright, J. A. *Practical Cloud-Native Java Development with MicroProfile*. Packt.

[8] Perikov, I. 5 patterns to make your microservice fault-tolerant. https://itnext.io/5-patterns-to-make-your-microservice-fault-tolerant-f3a1c73547b3.

[9] Pubudu, N. Design patterns for microservices — circuit breaker pattern. https://medium.com/nerd-for-tech/design-patterns-for-microservices-circuit-breaker-pattern-ba402a45aac2.

[10] Sun, L. Guest view: 5 reasons to be excited about istio's future. https://sdtimes.com/softwaredev/guest-view-5-reasons-to-be-excited-about-istios-future/, 2020.