



universität  
wien

# BACHELORARBEIT / BACHELOR'S THESIS

Titel der Bachelorarbeit / Title of the Bachelor's Thesis

„Resilient deployments with Istio“

verfasst von / submitted by

Ruslan Jelbuldin 01407036

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 Computer Science UG2002

Betreut von / Supervisor:

Amine El Malki BSc. MSc.  
Research Group Software Architecture

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Monolithic vs. Distributed Approach</b>	<b>4</b>
<b>3</b>	<b>Microservices</b>	<b>5</b>
3.1	Microservices & Containerization . . . . .	7
3.2	Microservices & Cloud Computing . . . . .	8
3.3	Microservices & Kubernetes . . . . .	9
<b>4</b>	<b>Service Meshes</b>	<b>10</b>
4.1	Consul Connect . . . . .	11
4.2	Istio . . . . .	11
4.3	Linkerd . . . . .	11
4.4	Comparison of Istio, Linkerd and Consul . . . . .	11
<b>5</b>	<b>Implementation</b>	<b>13</b>
5.1	Technology stack . . . . .	13
5.2	Distributed Calculation Cloud . . . . .	13
5.3	How to run . . . . .	17
<b>6</b>	<b>Evaluation</b>	<b>20</b>
6.1	Canary deployment with Istio as part of Routing . . . . .	21
6.2	Fault Injection . . . . .	22
6.3	Fault Injection - Timeout . . . . .	22
6.4	Fault Injection - HTTP 5XX . . . . .	25
6.5	Retry . . . . .	27
6.6	Outlier detection . . . . .	29
<b>7</b>	<b>Discussion</b>	<b>32</b>
<b>8</b>	<b>Conclusion</b>	<b>32</b>
<b>9</b>	<b>Future Work</b>	<b>33</b>

## Abstract

The purpose of this thesis is to demonstrate on a practical example, how service meshes could be integrated into a complex distributed system. We will briefly cover how did the modern software engineering came to the point, where such an approach became necessary and why. To do so, we will need to scratch and analyse the surface of the topic distributed systems, cloud computing, containerisation and container orchestration and understand how each of these concepts are related. This will consequently bring us to the understanding of the concept of service mesh and shed light on its purpose and application area. Thereafter, we will get acquainted with ISTIO (open source service mesh platform) and integrate it to the distributed system on a practical example. To sum it all up, we will analyse the results and observe how ISTIO is gathering metrics and discuss and analyse it.

**Keywords:** Microservices, Distributed Systemes, Software Architecture, Service Mesh, Cloud Computing, Docker, Kubernetes, ISTIO

# 1 Introduction

Since the first appearance of programming languages in the middle of 20<sup>th</sup> century and since the level of abstraction had started to increase, the process of building software systems started to gain in complexity. Nonetheless, these systems were still relatively simple when compared to modern super complex one's. At the beginning of software engineering times, there was no such thing as internet. Every built system was created to work locally on a single machine. Those machines had very limited data processing abilities. Nonetheless, this all was still sufficient enough for the tasks of that time.

With the lapse of time technologies evolved: machines became more advanced and were able to process larger amounts of data within shorter periods, new programming paradigms were introduced, which opened new doors for research and improvement. Programming languages started to slowly morph from 0, 1 and punch card gibberish to a text that could have been read by people. They became more high-level. Very slowly, the steepness of the learning curve began to decline, introducing more people to computer science. These factors influenced those small systems too and they consequently started to grow. Simple scripts turned into more complex and more abstract programming code. Tasks became more complex too due to increased power of computers.

In 1966 - 1967 the US computer scientist Allen Kay first introduced the concept of Object Oriented Programming at grad school. He mentioned that - "The big idea was to use encapsulated mini-computers in software which communicated via message passing rather than direct data sharing — to stop breaking down programs into separate "data structures" and "procedures"" [16]. Code pieces started to decouple into separate files, where each file had served it's own purpose. Although programs were split into file chunks, they still were running on one single machine and as a single instance and couldn't be accessed from elsewhere. It was one of the ceilings that computer science faced before invention of the internet and its worldwide spreading and overall integration.

But even before internet computer scientists were thinking on how to handle growing system complexity. The concept of Software Architectures, that started to emerge in early sixties was aimed to solve the problem. The idea was to systematize and cleverly decouple program components into an understandable structure that would make sense to anyone who is ever going to work with it. More complex systems started to appear. Development teams were growing. Depending on system application area, different architectures patterns were created with the laps of time. The most widely used are Layered (n-tier), Event Driven, Microkernel and with worldwide internet spreading - Microservices patterns. The first 3 were and still are great for their application area and each is actively used nowadays. However, back in time growing software complexity made it harder and harder to scale programs that were based on them. They were monolithic.

In nineties internet started it's world conquer and created new opportunities for software development and new challenges. It was spreading at an extreme pace and data transmission speed was increasing. In 2005 Dr. Peter Rogers first used the term "Web Micro Service". He described it as a way to split monolithic designs into multiple components/Processes. thereby making the codebase more granular and manageable [15]. As the time passed the pattern was widely adopted especially with appearance of cloud computing. Microservices and Cloud Computing became indivisible. However, the progress didn't stop and industry faced the new ceiling. Distributed systems were becoming more complex and it became harder to track data flow between services. It

became difficult to determine what has failed and where. Especially in containerized services in Kubernetes nodes. Demand for tracking tools appeared and one of these tools is ISTIO.

But let's first have a closer look at Architectures, containerization and orchestration concepts before diving into ISTIO and sum up why we ended up needing service meshes at all.

## 2 Monolithic vs. Distributed Approach

In modern software engineering Architectures could be split into 2 main types:

- Monolithic
- Distributed

Over time and with appearance of internet and cloud technologies some monolithic systems were replaced by distributed systems. This happened because some huge enterprise applications mainly relied on big number of features that were all part of a single program and failure of one part could cause failure of entire application, which could result in huge losses for the company. After 2010 big players started to migrate their legacy systems to cloud. However, this does not mean that monolithic approach is dead. It's alive and feeling well in smaller applications. On the other hand, big systems, like bank chains, government, military sector and IT giants shifted their sights on distributed approach, where microservices play bigger role.

Back in time applications were smaller and there were neither demand in decoupling services nor available means and infrastructure for building them. Therefore, monolithic apps were mostly dominating until the end of last century. Distributed systems, on the other hand, first appeared the mainstream stage in 1998, when the term Service-Oriented Architecture was first introduced. Not so long after, in 2005, the Microservice architecture, the ideological successor of SOA, entered the stage and became the backbone of modern cloud computing.

Comparison of Architectures	
Monolithic	Microservices
- Application is a single, integrated software instance	- Application is broken into modular components. This makes the application easier to understand, develop, test, and become more resilient to architecture erosion.[3] It can be considered as main advantage of microservices and this benefit is often argued in comparison to the complexity of monolithic architectures. [23]
- Application instance resides on a single server or VM	- Application can be distributed across the clouds and datacenter
- Updates to an application feature require reconfiguration of entire app	- Adding new features only requires those individual microservice to be updated
- Network services can be hardware based and configured specifically for the server	- Network services must be software-defined and run as a fabric for each microservice to connect to
-	- Scalability - Microservices can be implemented and deployed independently namely they run in independent processes and can be scaled, monitored and maintained independently. [5]
-	- Legacy migration: Microservices is considered as an effective solution for migrating existing monolithic software application.[17] [22]

### 3 Microservices

Microservices as an architectural style originate from Service-Oriented Architecture or SOA. The idea behind SOA is to create reusable and specialized components that work independently from one another. Key difference between two approaches is scope. SOA is enterprise oriented whereas Microservices' scope is application .

The above figure clearly shows us the scope difference between two approaches. In SOA we are speaking about multiple monolithic applications interconnected via enterprise network. The system itself is way bigger. Each application is a separate module that consists of various pieces that build it up as a single monolithic program. All these pieces are tightly coupled and entire application depends on the healthiness

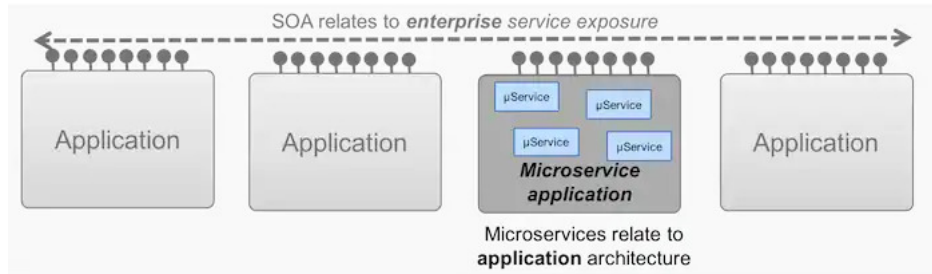


Figure 1: SOA vs. Microservice

of each piece. Because it can easily happen that if one fails, the entire application can collapse as a house of cards, but not necessarily.

Microservice application, on the other hand, has these pieces decoupled. This loose coupling isolates each piece and makes them independent even on technology stack level. For instance, if we have an IOT app that monitors weather measurements, we could have sensors software written in C, calculations module written in C++, messaging service in Java and UI could be an Electron/Web/Mobile app written in Flutter. Each of the services could be run from different location and doesn't require enterprise network. Interservice communication usually happens via exposed API endpoints. A common practice of deployment of microservices apps is containerization.

However, there are few moments to think about when considering microservices as architecture of choice. Although, it offers great scalability and encapsulation capabilities, it is considered to be way slower than monolithic approach. In monolithic apps, since each module is located and run on one machine, it has direct access to machines' computing power. Delays are minimal. Whereas, microservice based systems are distributed, hence could, theoretically, be run on different servers, that could even be located in different countries. The data between processes flows over the network with all the ensuing consequences like longer transfer time and possible security issues. Figure below sums up pros and cons of microservice architecture[14].

Advantages and disadvantages of Microservice Architecture	
PRO	CON
<ul style="list-style-type: none"> <li>- Versatile — microservices allow for the use of different technologies and languages</li> <li>- Easy to integrate and scale with third-party applications</li> <li>- Microservices can be deployed as needed, so they work well within agile methodologies</li> <li>- Solutions developed using microservice architecture allow for fast and continuous improvement of each functionality</li> <li>- Maintenance is simpler and cheaper — with microservices you can make improvements or amendments one module at a time, leaving the rest to function normally</li> <li>- The developer can take advantage of functionalities that have already been developed by third parties — you don't need to reinvent the wheel here, simply use what already exists and works</li> <li>- A modular project based on microservices evolves more naturally, it's an easy way to manage different developments, utilising the resources available, at the same time</li> </ul>	<ul style="list-style-type: none"> <li>- Because the components are distributed, global testing is more complicated</li> <li>- It's necessary to control the number of microservices that are being managed, since the more microservices that exist in one solution, the more difficult it is to manage and integrate them</li> <li>- Microservices require experienced developers with a very high level of expertise</li> <li>- Thorough version control is required</li> <li>- Microservice architecture can be expensive to implement due to licensing costs for third party apps</li> </ul>

### 3.1 Microservices & Containerization

Another important concept is containerization. One of the biggest problems with traditional development is that when we try to run our app on another machine, a high chance exists, that app will not run properly/at all because of version inconsistency of some packages, libraries, OS, programming languages, compilers and many many more other small things. Containerization solves this huge problem.

If we put it simple, containerization is packaging the source code of the app into a bare minimal OS setup with minimal set of libraries and dependencies that are required to launch the application. This package is called a container. Containers weigh less and are more resource-efficient than Virtual Machines, which makes it much faster to deploy and which also reduces overall costs including licensing. According to IBM's



latest survey<sup>1</sup> - "61% of container adopters reported using containers in 50% or more of the new applications they built during the previous two years; 64% of adopters expected 50% or more of their existing applications to be put into containers during the next two years." [10] The massive popularization of containerization began with introduction of Docker engine in 2013 which became industry standard. Other popular solutions are: LXC (Linux), Hyper-V and Windows Containers, Podman & others.

One of the most promoted conceptual advantages of containerization is that the code they contain could be "written once and run everywhere". It gives it the following advantages:

- improved development speed
- prevention of cloud vendor lock-in
- fault isolation
- ease of management
- simplified security and many more [10]

This already makes containers ideal companions for microservices. By building our entire distributed system on containers we can avoid database bottlenecks and enable Continuous Integration / Continuous Delivery (CI/CD) pipelines. As consequence system scalability is also greatly improved, since each service can be individually updated without influencing other ones. If necessary, we could even rewrite existing service using entirely different technology stack. Furthermore, containers are more secure, since they have less entry points that should be observed, making them easier to monitor.

## 3.2 Microservices & Cloud Computing

Cloud computing became integral part of modern software engineering and for a good reason. As already mentioned previously, with the lapse of time systems had become more complex and old-school server-client approach can not handle the growing codebases and data flow. Cloud computing brought application decoupling to the next level. Now we don't need expensive servers that will host our backend. We can just rent them and not only deploy our entire app there but also deliver entire storages, databases, servers, analytics etc. over the internet. Service providers like Google, Amazon and Microsoft provide developers with power of their data centers. Basically speaking, we delegate computation to these server providers. They provide us with "clouds" where we can manage our systems the way we want. Pricing is based on how much computing power is used. So it's relatively cheap to start. Clouds could be public, private and hybrid. Depending on how much control we need over our project we can choose over the following types<sup>2</sup> of cloud services:

- Infrastructure as a service (IaaS)
- Platform as a service (PaaS)
- Serverless computing
- Software as a service (SaaS)

Cloud computing caused the appearance of new software concept - Cloud native applications that entirely reside in clouds. Microservices wrapped in containers are naturally fitting into this concept, as they are made of independent modules, each of which can be deployed to the cloud and leverage its functionality and computation power.

---

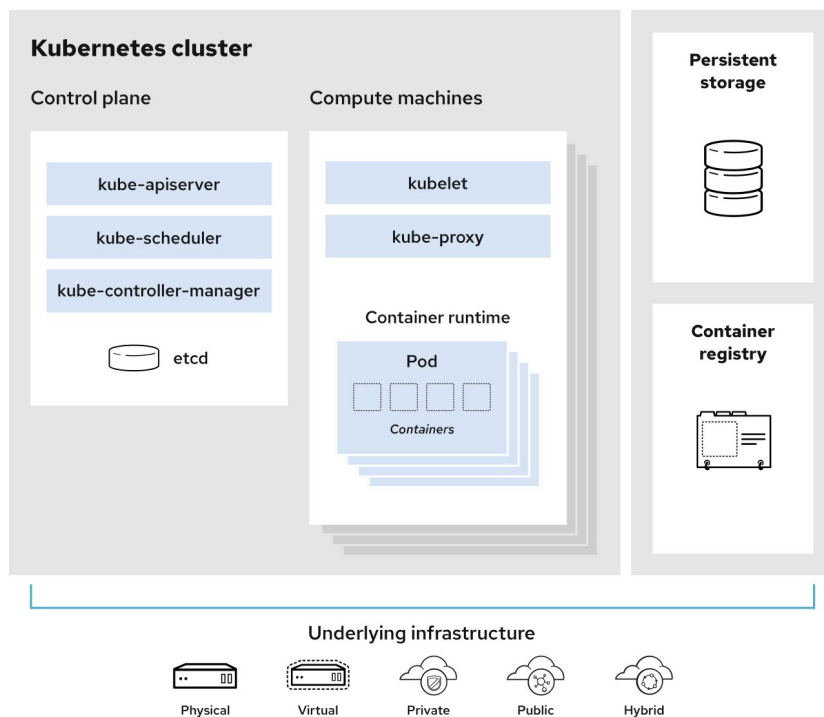
<sup>1</sup><https://www.ibm.com/downloads/cas/VG8KRPRM>

<sup>2</sup><https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/#benefits>

### 3.3 Microservices & Kubernetes

However, industry is not stopping there. The number of containers that are being deployed to cloud grows, and it's getting harder and harder to operate them. In 2014 Google introduced Kubernetes also known as k8s and "kube", an open-source container orchestration tool that manages clusters of Virtual machines and Schedules container deployment. It will take an entire thesis to describe Kubernetes in detail, but in a nutshell its main objective is to keep deployed containers up and running by automating many of the manual processes involved in deploying. Containers are being run in nodes that are organized in clusters. Nodes are the instances where our microservices are running. Clusters contain control pane that is responsible for node state management. Clusters can be used as development, testing and production environments.

Having containers with microservices deployed to Kubernetes nodes and grouped into appropriate clusters, it becomes relatively easy to test, scale and update each of the individual services. Moreover, Kubernetes is able to perform health-checks, failovers, networking, service discoveries, load balancing, container life cycle management and many other useful operations. This all makes Kubernetes play a giant role in CI-CD and a very useful tool for complex applications. Cloud platforms like Google, AWS, MS Azure provide access to Kubernetes. Red-Hat also provides an alternative called Open Shift. Below figure shows the example Kubernetes cluster. <sup>3</sup>



<sup>3</sup>[https://www.redhat.com/cms/managed-files/kubernetes\\_diagram-770x71700v2.svg?itok=Z6bFR9q1](https://www.redhat.com/cms/managed-files/kubernetes_diagram-770x71700v2.svg?itok=Z6bFR9q1) — v3 —

## 4 Service Meshes

A service mesh is a dedicated infrastructure layer that facilitates communication between (micro)services using a proxy.[13] This infrastructure level provides some benefits, such as providing observability into communications, secure connections or automating retries and backoff for failed requests.[1]

Even though Kubernetes helps to orchestrate containers, it is still hard to track how individual microservices interact with each other. Kubernetes can handle failovers, but it doesn't provide comprehensive tools for monitoring interservice communication. If we think about that, we can build microservices in such a way that this monitoring is possible, but the more complexer becomes communication, the harder it is to monitor it. Here is where service meshes come to rescue.

Basically a service mesh is abstraction to which we delegate monitoring duties. No new functionality is being added to service runtime. What's different about a service mesh is that it takes the logic governing service-to-service communication out of individual services and abstracts it to a layer of infrastructure[20].

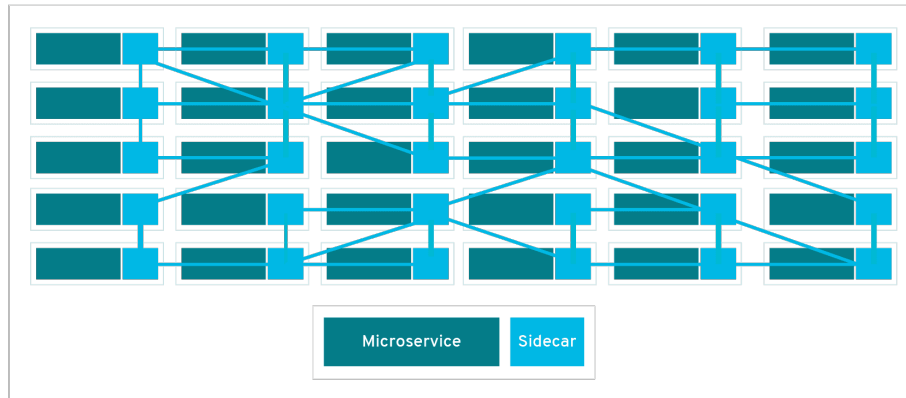


Figure 2: Proxy network [19]

We build it in to an app as an array of proxies. Requests are routed between microservices through proxies in their own infrastructure layer. These proxies are also called "sidecars" since they run not within but alongside microservices. All together these meshes form a mesh network. Every time a new service is added the new proxy instance is assigned to it. Proxies will monitor every aspect of service-to-service communication and create metrics report.

Istio is such a service mesh solution. It provides necessary functionality to manage traffic, security and observability of microservices. In below sections we will discuss how it works in greater detail on practical example.

There are three leading, open source solutions in the Kubernetes ecosystem for Service Mesh. As number of microservices grows, it creates challenges around figuring out how to enforce and standardize things like routing between multiple services/versions, authentication and authorization, encryption, and load balancing within a Kubernetes cluster. [18] Each service mesh solution has its benefits and drawbacks.

## 4.1 Consul Connect

Consul is a solution providing a full featured control plane with service discovery, configuration, and segmentation functionality. Each of these features can be used individually as needed, or they can be used together to build a full service mesh. [4] Consul Connect uses an agent installed on every node as a DaemonSet which communicates with the Envoy sidecar proxies that handles routing and forwarding of traffic. [18]

## 4.2 Istio

Istio is a kubernetes native solution, which is discussed in details in this thesis. Istio has separated its data and control planes by using a sidecar loaded proxy which caches information so that it does not need to go back to the control plane for every call. The control planes are pods that also run in the Kubernetes cluster, allowing for better resilience in the event that there is a failure of a single pod in any part of the service mesh.[18]

## 4.3 Linkerd

Linkerd is the 2nd popular service mesh on Kubernetes, which architecture mirrors Istio's closely, with an initial focus on simplicity instead of flexibility. Linkerd is a part of the Cloud Native Foundation (CNCF), which is the organization responsible for Kubernetes.[18]

## 4.4 Comparison of Istio, Linkerd and Consul




	 <b>Istio</b>	 <b>LINKERD</b> <b>Linkerd v2</b>	 <b>Consul</b>
<b>Supported Workloads</b>	Does it support both VMs-based applications and Kubernetes?		
<b>Workloads</b>	Kubernetes + VMs	Kubernetes only	Kubernetes + VMs
<b>Architecture</b>	The solution's architecture has implications on operation overhead.		
<b>Single point of failure</b>	No – uses sidecar per pod	No	No. But added complexity managing HA due to having to install the Consul server and its quorum operations, etc., vs. using the native K8s master primitives.
<b>Sidecar Proxy</b>	Yes (Envoy)	Yes	Yes (Envoy)
<b>Per-node agent</b>	No	No	Yes

Figure 3: Comparison by workloads and architecture [18]

Traffic Management			
Blue/Green Deployments	Yes	Yes	Yes
Circuit Breaking	Yes	No	Yes
Fault Injection	Yes	Yes	Yes
Rate Limiting	Yes	No	Yes

Figure 4: Comparison by traffic management [18]

Chaos Monkey-style Testing			
Traffic management features allow you to introduce delays or failures to some of the requests in order to improve the resiliency of your system and harden your operations			
Testing	Yes- you can configure services to delay or outright fail a certain percentage of requests	Limited	No
Observability			
In order to identify and troubleshoot incidents, you need distributed monitoring and tracing.			
Monitoring	Yes, with Prometheus	Yes, with Prometheus	Yes, with Prometheus
Distributed Tracing	Yes	Some	Yes

Figure 5: Comparison by testing and observability [18]

## 5 Implementation

### 5.1 Technology stack

Technologies used: Java 11, Gradle, Python 3.7, Docker, Kubernetes, Istio.

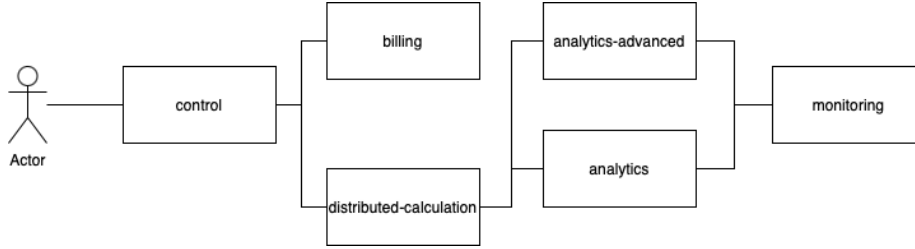


Figure 6: Distributed Calculation Cloud

### 5.2 Distributed Calculation Cloud

The goal of the study is testing of resilience features of Istio, understanding of how microservices can be managed and also how useful is this service mesh. To be able to test it, the distributed application was created and deployed with kubernetes and Istio. The distributed calculation cloud application is a simple version of Google Genomics or the LHC Computation Grid.[6] The solution provided in this paper allow users to perform "long" running calculations in asynchronous and multi-threaded fashion. To prevent the uneven and excessive utilization of calculation services a billing service is created [6], so users can create their bank accounts and pay for the requested calculations. As hundred of calculation and billing services can be deployed, the service repository was implemented. It is a main microservice which enable other services to simply ask which microservice supports a desired task.

1. Service Repository. The crucial microservice that contains information about all registered microservices, supports dynamically querying and registering any microservices and their endpoints. All other microservices know the main endpoint of the service repository MS. It is possible to register microservice's information such as description, endpoint path and its functionality. The service repository keeps all the information about registered microservices up-to-date. If there is a microservice that does not respond or not active anymore, it will be immediately removed from the list of registrations. It is also possible to query a list of microservices which support a requested job. Up to N Microservices' instances, that provide the same functionality can be registered. This crucial microservice was replaced with Istio features, since it supports service discovery, fault tolerance, retries etc. **This crucial microservice was implemented, but afterwards completely replaced with istio features, so there were no need in the service anymore.**
2. Distributed Calculation Service. This service simulates long running calculations such as, data analysis tasks or machine learning jobs. The service supports following calculations: addition of two numbers, calculation of two numbers,

calculation of prime and Fibonacci numbers. Each calculation operation has its own endpoint, that is registered at the service repository.

HTTP	Path	Request Body	Request Parameters	Consumes
POST	/add	decimalOne: Double, decimalTwo: Double	-	application/json
POST	/multiply	decimalOne: Double, decimalTwo: Double	-	application/json
POST	/prime	-	n	-
POST	/fibonacci	-	n	-
GET	/health	-	-	-
GET	/getWorkloadByType/{type}	-	-	-
POST	/fault	-	timeout: String, responseCode: Integer, frequency: Integer, shouldCalculate: Boolean	-

Figure 7: Distributed Calculation Service API

3. Billing Service. The microservice implements a simple billing service, where users can create their accounts, deposit them, get account information and pay for the requested calculations. A price for a calculation is based on the workload of each distributed calculation service instance and is changed dynamically, so if one of the calculation services is idle and the other one is loaded, the price for the last will be higher. For the sake of simplicity, a user can create/delete an account only with the first name, which is unique in this case.

HTTP	Path	Request Body	Request Parameters	Consumes
POST	/createAccount	-	user: String	-
POST	/depositAccount	-	user: String, amount: Double	-
POST	/chargeAccount	-	user: String, amount: Double	-
GET	/getAccountInfo	-	user: String	-
DELETE	/deleteAccount	-	user: String	-
GET	/health	-	-	-

Figure 8: Billing Service API

4. Service Control Center. The control center for the whole application and its main entry point. With this service users can request calculations, account information, deposit, charge and delete their accounts. It interacts with all

microservices in the system and all incoming requests are redirected to a corresponding MS. The logic for calculation price is also implemented here. In Figure 6 is shown the communication between microservices after calculation is requested.

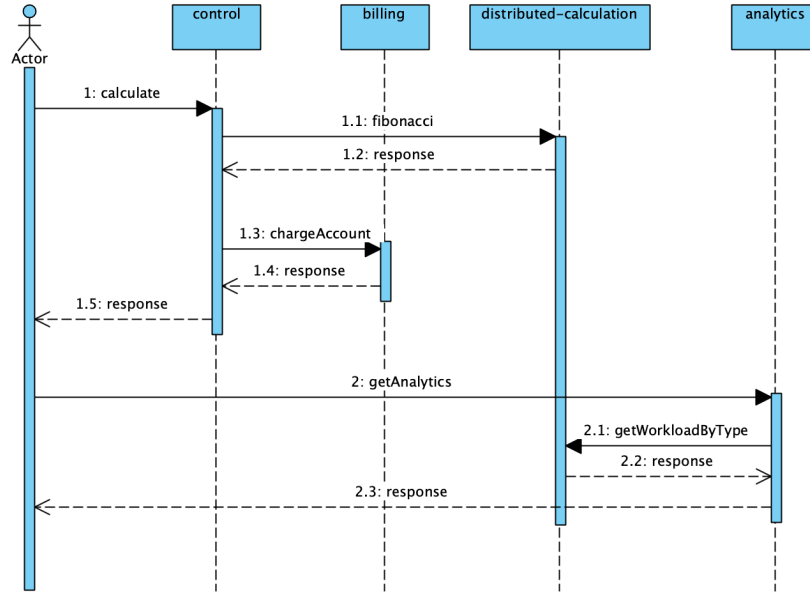


Figure 9: Successful calculation flow

- 1. A user requested a calculation via control panel. The requirement includes the description about calculation type, price that the user is able to pay.
  - 1.1 Control panel sends a calculation request to distributed calculation service.
  - 1.2 The result of calculation is returned to control panel.
  - 1.3 The User is charged.
  - 1.4 OK from the billing service
  - 1.5 The result of calculation is returned to the user.
  - 2. The user requested analytics for a specific calculation.
  - 2.1 The analytics service requests workload from calculation service.
  - 2.2 The result with workload is returned to analytics service.
  - 2.3 The result with analytics is returned back to the requestor.
5. Fake Behaviour Application. This application is connected to the control panel and replaces/simulates the frontend, from which customers can request the calculations, create and deposit their accounts.



HTTP	Path	Request Body	Request Parameters	Consumes
POST	/createAccount	-	user: String	-
POST	/depositAccount	-	user: String, amount: Double	-
POST	/chargeAccount	-	user: String, amount: Double	-
GET	/getAccountInfo	-	user: String	-
DELETE	/deleteAccount	-	user: String	-
POST	/calculate	user: String, calculationType: String, price: Double, firstNumber: Double, secondNumber: Double	-	-

Figure 10: Service Control Center API

6. Analytics. The service provides information about estimated calculation time and has a simple API. It collects the information about registered microservices from service-repository and calculates the estimated time for the requested calculation type. It can be used as backend for monitoring applications.

```
{
  "estimatedTime": 4.990354593777644,
  "function": "add"
}
```

Figure 11: Analytics service response

HTTP	Path	Request Body	Request Parameters	Consumes
GET	/getAnalytics	-	function: String	-
GET	/version	-	-	-

Figure 12: Analytics/Advanced Analytics API

7. Advanced Analytics. It is an improved version of analytics, that can additionally predict estimated price in euro and dollars for requested type of calculation. The service was created for testing canary deployments with Istio.

```
{  
  "estimatedPriceInDollar": 17.890592462954736,  
  "estimatedPriceInEur": 18.890592462954736,  
  "estimatedTime": 9.458302271030616,  
  "function": "add"  
}
```

Figure 13: Advanced Analytics service response

The implementation of services was a part of DSE (Distributed Systems Engineering) course, where I implemented the distributed calculation service. However I had completely rewritten all other microservices, so it is a different communication logic withing the whole application. Moreover, I decided to implement microservices in different programming languages such as Java and Python, to have a polyglot stack, what is a common practice in distributed systems. I also added additional functionalities and endpoints to each microservice, so it will be easier to test istio resiliency features:

1. The distributed calculation service was implemented in two versions, the first one with a standard set of endpoints and the second version has additional multiplication endpoint. It allows to demonstrate the canary deployment with help of Istio.
2. Additional endpoint for fault injection in the distributed calculation service was added. Following parameters are provided: timeout - set MS response timeout, responseCode - set response code, the ms will answer with, frequency - how often the MS will answer with the provided response code, shouldCalculate - switch on/off real calculations to make testing easier. If shouldCalculate parameter set to false, all calculation endpoints will return HTTP Status 200 without calculation result.
3. Health-checks were added to each MS.
4. New simple application for generating behaviour of users/clients was implemented.
5. New analytics service, that can predict estimated time of calculation.
6. New advanced analytics service, that can predict estimated time of calculation and price in euro and dollars.

### 5.3 How to run

There are following requirements for Linux and MacOS: Be sure that Istio supports the version of Kubernetes. The version of kubernetes - v.1.18.1. The version of Istio - 1.10.3. Minikube must be installed. At least 16GB of RAM, otherwise problems with running services. Deployment with Kubernetes and Istio:

- Clone the project from github:

```
git clone https://github.com/troublemaker92/resilient-microservices-with-istio.git
```

- Change directory:

```
cd resilient-microservices-with-istio
```

- start minikube:

```
minikube start
```

- Move to the Istio package directory.

```
cd istio/istio-1.10.3/
```

- Add the istioctl client to your path (Linux or macOS):

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

- Set default istio profile:

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

- Add a namespace label to instruct Istio to automatically inject Envoy sidecar proxies for future application deployments: [12]

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

- Deploy all applications with kubernetes:

```
kubectl apply -f k8s/service-repository-deployment.yaml;
kubectl apply -f k8s/calculation-deployment.yaml;
kubectl apply -f k8s/billing-deployment.yaml;
kubectl apply -f k8s/control-center-deployment.yaml;
kubectl apply -f k8s/analytics-deployment.yaml;
```

- Set environment variable for minikube ip and ingress port:

```
sh export_ingress_ip_port.sh
```

- Apply ingress gateway and virtual service

```
kubectl apply -f istio/ing_gateway.yaml;
kubectl apply -f istio/virt_svc.yaml;
```

- For monitoring be sure, that kiali, grafana and prometheus are installed.

- Be sure that all services are up and running:

```
kubectl get all -A
```

- Start the python simulation app:

```
python3 microservices/app-driver/calculation-driver.py
```

- Start simpilation:

```
http://localhost:8080/simulate?duration=180&ip=192.168.64.3&port=30439
```

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
default	service/analytics	ClusterIP	10.109.114.233	<none>	8080/TCP	50s
default	service/analytics-advanced	ClusterIP	10.96.196.93	<none>	8080/TCP	50s
default	service/billing	ClusterIP	10.98.179.28	<none>	8080/TCP	51s
default	service/control	NodePort	10.105.107.24	<none>	8080:30000/TCP	51s
default	service/distributed-calculation	NodePort	10.102.244.166	<none>	8080:30033/TCP	51s
default	service/kubernetet	ClusterIP	10.96.8.1	<none>	443/TCP	27d
istio-system	service/grafana	ClusterIP	10.105.112.68	<none>	3000/TCP	24d
istio-system	service/istio-egressgateway	ClusterIP	10.110.165.206	<none>	80/TCP,443/TCP	31d
istio-system	service/istio-ingressgateway	LoadBalancer	10.107.214.145	<pending>	15021:30009/TCP, 80:30439/TCP, 443:32457/TCP, 31400:30774/TCP, 15443:30504/TCP	31d
istio-system	service/istiod	ClusterIP	10.107.241.255	<none>	15010/TCP,15012/TCP,443/TCP,15014/TCP	31d
istio-system	service/kiali	ClusterIP	10.108.93.36	<none>	20061/TCP,9090/TCP	30d
istio-system	service/prometheus	ClusterIP	10.101.143.159	<none>	9090/TCP	30d
kube-system	service/kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP,53/TCP,5153/TCP	32d

Figure 14: Services

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
default	deployment.apps/analytics	1/1	1	1	50s
default	deployment.apps/analytics-advanced	1/1	1	1	50s
default	deployment.apps/billing	1/1	1	1	51s
default	deployment.apps/control	1/1	1	1	51s
default	deployment.apps/distributed-calculation-1	1/1	1	1	51s
default	deployment.apps/distributed-calculation-2	1/1	1	1	51s
istio-system	deployment.apps/grafana	1/1	1	1	24d
istio-system	deployment.apps/istio-egressgateway	1/1	1	1	31d
istio-system	deployment.apps/istio-ingressgateway	1/1	1	1	31d
istio-system	deployment.apps/istiod	1/1	1	1	31d
istio-system	deployment.apps/kiali	1/1	1	1	30d
istio-system	deployment.apps/prometheus	1/1	1	1	30d
kube-system	deployment.apps/coredns	1/1	1	1	32d

Figure 15: Deployments

Important is, that all the services are up and running. From the default namespace we can see, that all deployments are there and available. Be sure, that all needed services from istio-system namespace are also available. Ingressgateway allows you to define entry points into the mesh that all incoming traffic flows through[9]. Egress gateway is a symmetrical concept; it defines exit points from the mesh. Egress gateways allow you to apply Istio features, for example, monitoring and route rules, to traffic exiting the mesh. [9] Grafana, kiali and prometheus are needed for visualization.

```
jelbuldinr87@jelbuldinr87 git/resilient-microservices-with-istio main • kubectl get all -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	pod/analytics-7b9485c488-lqjf6	2/2	Running	0	50s
default	pod/analytics-advanced-57bd4dbbf9-schtr	2/2	Running	0	50s
default	pod/billing-6bddb9c7df-7c9vg	2/2	Running	0	51s
default	pod/control-75b8f687-6hmtw	2/2	Running	0	51s
default	pod/distributed-calculation-1-58d69bbb8-5gl9k	2/2	Running	0	51s
default	pod/distributed-calculation-2-5dbbb595dc-lkrmm	2/2	Running	0	50s
istio-system	pod/grafana-556f8998cd-rwm89	1/1	Running	26	24d
istio-system	pod/istio-egressgateway-7bfdcc9d86-p2wnk	1/1	Running	20	31d
istio-system	pod/istio-ingressgateway-565bfd4d-8jnv	1/1	Running	20	31d
istio-system	pod/istiod-5c6d886bd8-hzgzk	1/1	Running	20	31d
istio-system	pod/kiali-85c8cdd5b5-2lkm2	1/1	Running	56	30d
istio-system	pod/prometheus-69f7f4d689-pd68d	2/2	Running	76	30d

Figure 16: All pods are up and running.

## 6 Evaluation

Before testing Istio features, a normal communication between microservices is presented in the figure 17.

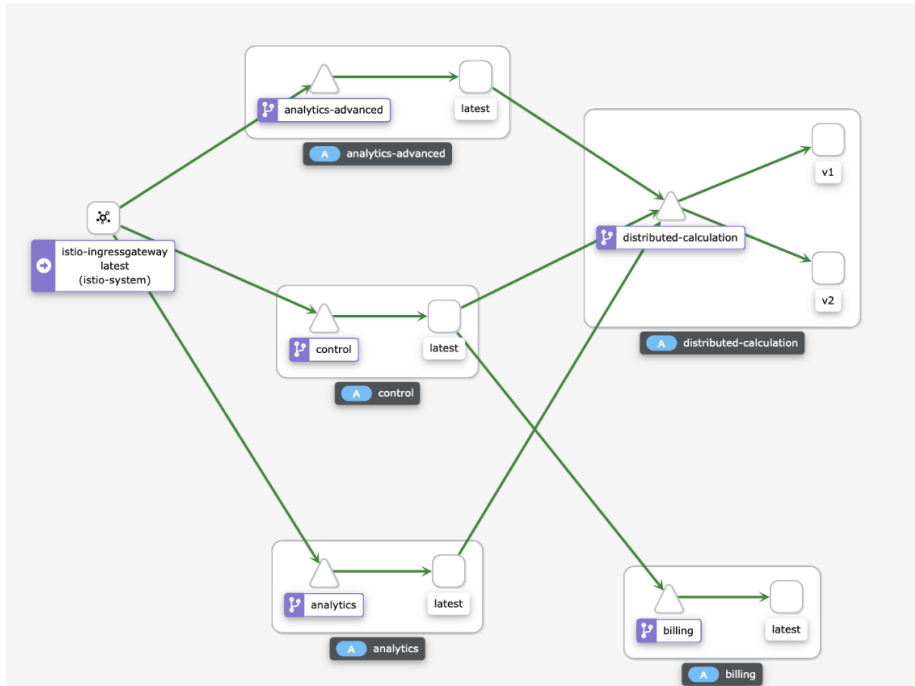


Figure 17: Kiali - All services work correctly.

## 6.1 Canary deployment with Istio as part of Routing

A canary is a deployment strategy that releases an application or service incrementally to a subset of users [2]. A canary deployment has the lowest risk compared to other release strategies, because it can be controlled very easily. With Istio can be configured, that N-percent of the traffic should be forwarded to one or another service. In our case we have analytics service, that predicts the estimated time of calculations and a new version of the analytics, that can predict the price of calculation in dollars and euro. To test this feature in production 10 percent of all traffic will be redirected to the new advanced analytics service and 90 percent to the old one. In case of failure the rollback scenario is pretty simple, in the istio config file 100 % of requests should be directed to the old service and 0 % to the new one. To apply the deployment do the following:

- Apply canary deployment

```
kubectl apply -f istio/virt_svc_10_90_canary.yaml
```

After it is applied, we can see from the Kiali, that only 10 percent of requests are forwarded to the new service and the rest to the old one.

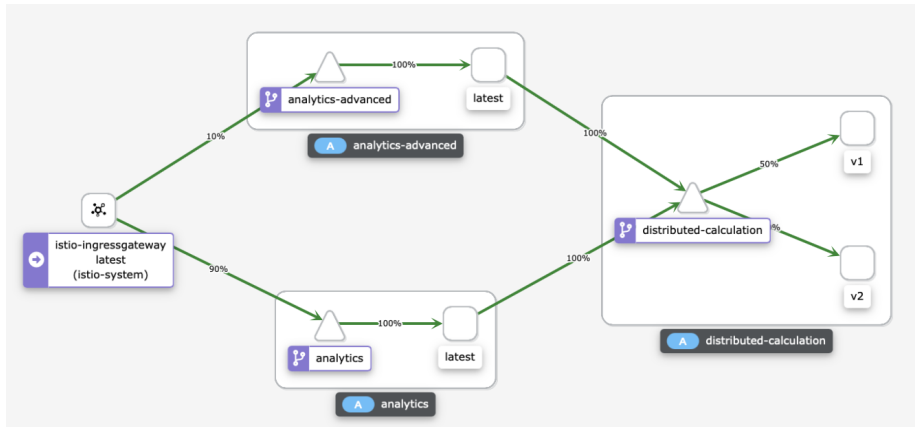


Figure 18: Requests distribution between analytics and advanced analytics services.

```

- match:
  - uri:
    | prefix: "/getAnalytics"
    route:
  - destination:
    | host: analytics.default.svc.cluster.local
    | port:
    | | number: 8080
    weight: 90
  - destination:
    | host: analytics-advanced.default.svc.cluster.local
    | port:
    | | number: 8080
    weight: 10

```

Figure 19: Canary deployment configuration.

## 6.2 Fault Injection

While architecting the cloud calculation application, the whole system was designed for resiliency. The application can fail somewhere, so it was designed with failure in mind. Intentionally additional endpoints were added to distributed calculation service such a /fault for testing Istio resiliency features, where such features are provided. Chaos Testing is a practice to intentionally introduce failures into your system to test the resiliency and recovery of microservices architecture. The Mean Time to Recovery (MTTR) needs to be minimized in modern day architectures. Hence, it is beneficial to validate different failure scenarios ahead of time and to take the necessary steps to stabilize the system and make it more resilient.[7] With Istio it is possible to inject failures at application layer, it can be different HTTP-Errors or delays. It is even possible to create a fault injection rule to delay traffic coming from the specific user. [11]

## 6.3 Fault Injection - Timeout

The calculation service has a 15 seconds hard-coded connection timeout for any calls. With this timeout the end-to-end flow should work normally, but with delay. It can be seen from Postman pictures, where the normal response time before fault injection was < 100ms and after > 15s. From the provided config file we can see that the timeout was set to 100 % of requests. The timeout was set for the distributed calculation MS, and since it is a central service in the ecosystem, the control center will not be able to communicate with it in a normal way. We can see on the Kiali graph after timeout injection, that there are no errors, but access time from control center to distributed calculation is 15 seconds after injection, what can be seen from postman pictures.

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: distributed-calculation
spec:
  hosts:
  - distributed-calculation.default.svc.cluster.local
  http:
  - fault:
      delay:
        fixedDelay: 10s
        percentage:
          value: 100
      route:
      - destination:
          host: distributed-calculation.default.svc.cluster.local

```

Figure 20: Fault injection - Timeout configuration

- Apply fault injection with delay using istio

```
kubectl apply -f istio/fault_injection_calculation_delay.yaml
```

ms4\_getAccountInfo Examples

GET 192.168.64.3:30439/getAccountInfo?user=ruslan

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> user	ruslan	

Body Cookies Headers (5) Test Results ⌕ Status: 200 OK Time: 62 ms

Pretty Raw Preview Visualize JSON ≡

```

1 {
2   "balance": 9260,
3   "name": "ruslan"
4 }

```

Figure 21: Postman - Normal response time.



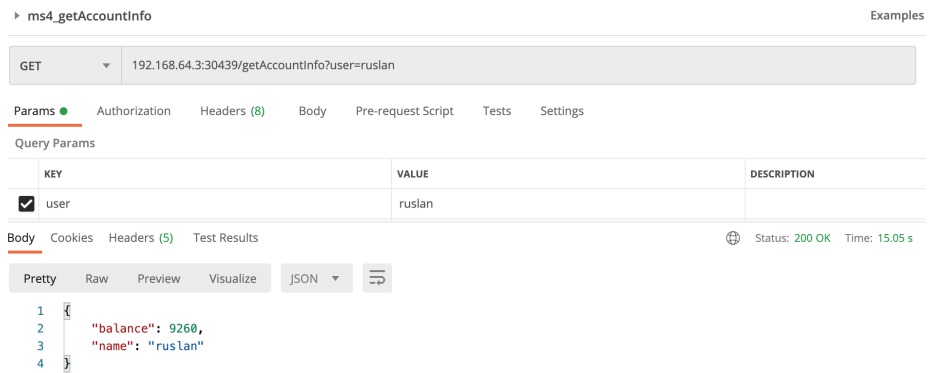


Figure 22: Postman - Increased response time after fault injection.

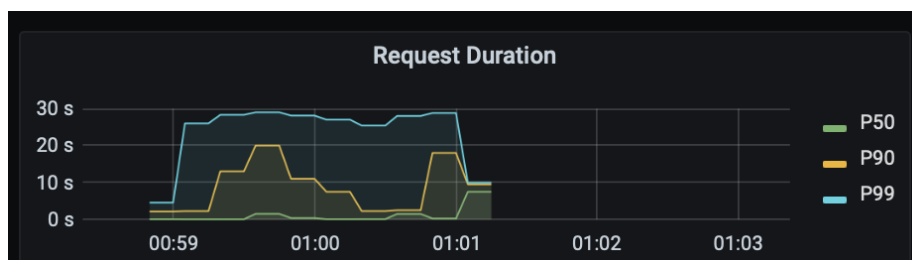


Figure 23: Grafana - Increased request duration after fault injection.

## 6.4 Fault Injection - HTTP 5XX

- Apply fault injection with HTTP 500

```
kubectl apply -f istio/fault_injection_calculation_502_50.yaml
```

On the Kiali graph is shown the expected results . The control center can access the distributed calculation service only partially. For the half of requests it is failed. Right after injecting the http 500 for 50% of requests, we can see on grafana dashboard, that the success rate is 91%, what means, that compared to the normal success rate of 100 % it decreases. The success rate for analytics and advanced analytics services is approx. 50 %, which means, that the fault injection works correctly.

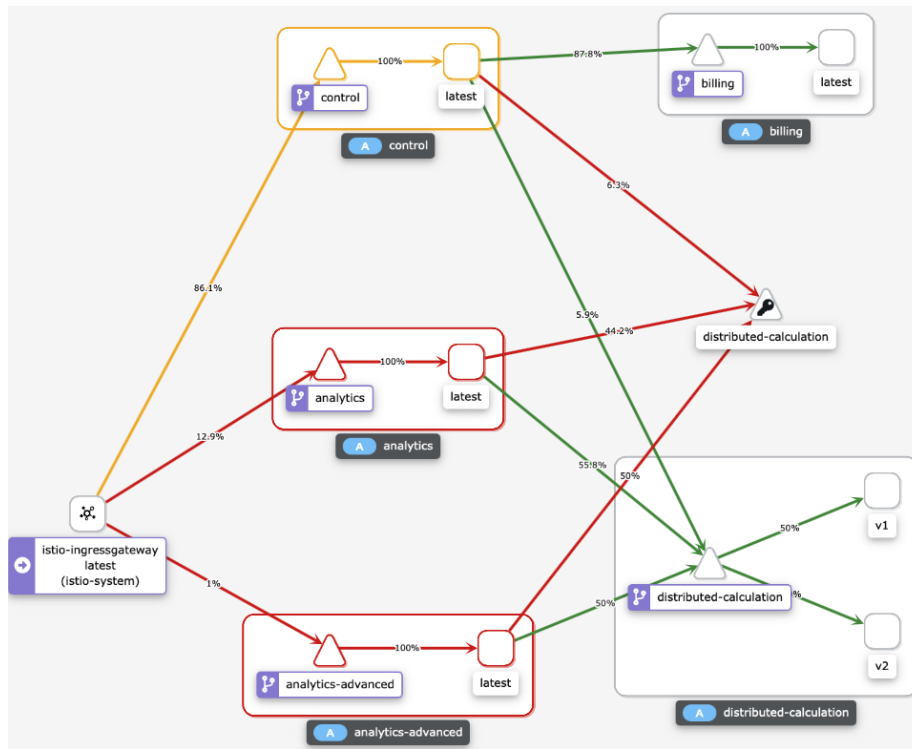


Figure 24: Kiali graph with fault injection - HTTP 500

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
-	unknown.unknown	-	261.68 µs	471.03 µs	4.10 ms	-
billing.default.svc.cluster.local	billing.default	18.51 ops/s	5.36 ms	11.45 ms	24.42 ms	100.00%
control.default.svc.cluster.local	control.default	14.71 ops/s	21.69 ms	71.75 ms	25.02 s	91.94%
distributed-calculation.default.svc.cluster.local	distributed-calculation-1.default	1.49 ops/s	25.02 ms	25.11 s	29.51 s	100.00%
distributed-calculation.default.svc.cluster.local	distributed-calculation-2.default	1.31 ops/s	8.46 ms	1.00 s	27.00 s	100.00%
analytics.default.svc.cluster.local	analytics.default	1.24 ops/s	29.25 ms	58.75 ms	99.62 ms	52.34%
analytics-advanced.default.svc.cluster.local	analytics-advanced.default	0.16 ops/s	33.93 ms	49.64 ms	94.50 ms	58.33%

Figure 25: Grafana - Fault injection - HTTP 500

```

apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: distributed-calculation
spec:
  hosts:
  - distributed-calculation.default.svc.cluster.local
  http:
  - fault:
      abort:
        httpStatus: 500
        percentage:
          value: 50
      route:
      - destination:
          host: distributed-calculation.default.svc.cluster.local

```

Figure 26: Fault injection configuration - HTTP 500

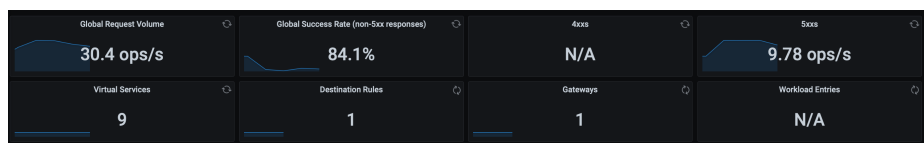


Figure 27: Grafana - success rate with fault injection

## 6.5 Retry

For testing retry feature, do the following:

- Before testing delete all deployments:

```
sh delete_all.sh
```

- Start testing retry feature with a script:

```
sh retry_test.sh
```

Script description (figure 29):

- Row 1-2: Set environment variables for minikube ip and ingress port. Since the ip address is not static, it is better to use it as environment variable.
- Row 3-8: Apply all kubernetes and istio deployments needed for testing of retry feature.
- Row 10: Waiting 20 seconds till all the deployments(pods, services) are up and running.
- Row 11: Fault injection: The calculation service will answer with HTTP 502 with 50 percent chance.
- Row 13-14: Create an account in the billing service and deposit it.
- Row 15: Send 20 requests to the calculation service via control panel.
- Row 18: Fetch http trace from the calculation service.

For sake of ease in this test only one calculation service is applied. A HTTP fault is injected, so the service responses with 502 with 50 percent chance. After that 100 requests were sent to the calculation service via control panel before and after istio retry injection. All requests were sent with help of postman runner, where a request is considered to be failed, if the service not responds with HTTP 200. In figure 29 is a postman collection run is shown, where 54 requests were failed and 46 succeeded and the same result we can see in figure 30 on the left side from Kiali dashboard. After Istio retry configuration was applied, the success rate in postman was 100% (see figure 31), whilst in the Kiali dashboard we can see, that success rate was not changed, what is shown in figure 30 on the right side. Kiali dashboard shows all inbound/outbound requests, which means, that the calculation service still responds with HTTP 502, but the final result in postman with 100% success rate demonstrates, that retry feature works as expected. Executing a script "retry test.sh", 20 requests are sent to the calculation service and in the end the http trace of the service is requested. From the trace we can see, that the calculation service accepted more than 20 requests, which means, that the retry config works as expected. An example of http trace can be found here: `/logs/calculation_http_trace_retry_20_35.json`. In this file we can see, that the service responded 15x times with HTTP 502 and the rest with HTTP 200.

```

1 export MINIKUBE_IP=$(minikube ip)
2 export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
3 kubectl apply -f k8s/billing-deployment.yaml
4 kubectl apply -f k8s/control-center-deployment.yaml
5 kubectl apply -f k8s/retry_calculation-deployment.yaml
6 kubectl apply -f istio/ing_gateway.yaml
7 kubectl apply -f istio/virt_svc.yaml
8 kubectl apply -f istio/retry.yaml
9 echo "Waiting 20 seconds till all services are up and running..."
10 sleep 20;
11 curl --location --request POST "http://$MINIKUBE_IP:30033/fault?timeout=0&responseCode=502&frequency=2&shouldCalculate=false"
12 printf "\n";
13 curl --location --request POST "http://$MINIKUBE_IP:$INGRESS_PORT/createAccount?user=ruslan" --data-raw ''
14 curl --location --request POST "http://$MINIKUBE_IP:$INGRESS_PORT/depositAccount?user=ruslan&amount=150000"
15 for i in {1..20}; do sleep 0.1; curl --location --request POST "http://$MINIKUBE_IP:$INGRESS_PORT/calculate2" --header 'Content-Type: application/json'
16 --data-raw '{"user": "ruslan","calculationType": "prime","price": 15,"firstNumber": 120,"secondNumber": 200}'; done
17 printf "\n";
18 echo "Result: "
19 curl --location --request GET "http://$MINIKUBE_IP:30033/actuator/httptrace"

```

Figure 28: Retry configuration.

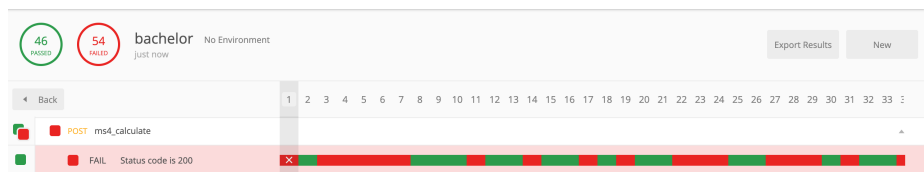


Figure 29: Postman - Successful/Failed Requests.

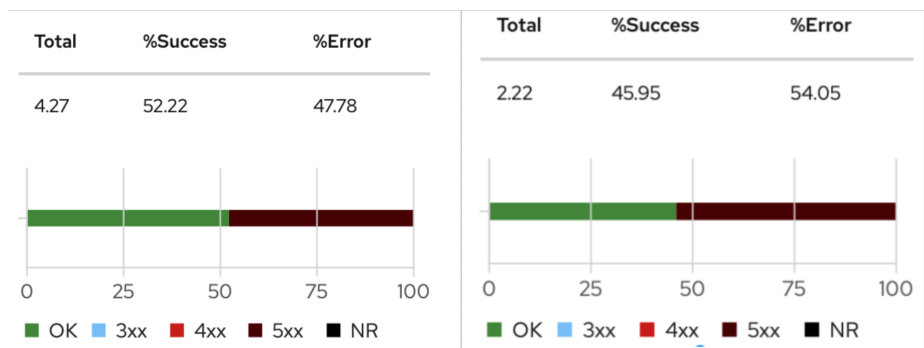


Figure 30: Kiali - Inboud requests before/after istio retry is applied.

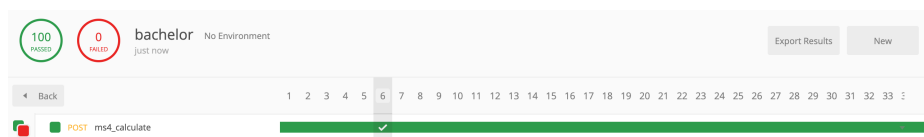


Figure 31: Postman - After istio retry is applied.

## 6.6 Outlier detection

Outlier detection - is an istio resiliency strategy that is used to detect unusual behaviour of a service and if a host is unhealthy, it will be evicted from the list. This feature of Istio can track the status of each service and check metrics like consecutive errors and latency associated with service calls. If it finds outliers, it will automatically evict them. [8] The following features for outlier detection provided in Istio:

1. BaseEjectionTime - The maximum ejection duration for a host. For example, in the config provided below the host will be ejected for 60 seconds before it is evaluated again for processing requests. [8]
2. ConsecutiveErrors - Number of errors before a host is ejected from the connection pool. For example, if you have 3 consecutive errors while interacting with a service, Istio will mark the pod as unhealthy.[8]
3. Interval - The time interval for ejection analysis. For example, the service dependencies are verified every 5 seconds.[8]
4. MaxEjectionPercent - The max percent of hosts that can be ejected from the load balanced pool. For example, setting this field to 100 implies that any unhealthy pods throwing consecutive errors can be ejected and the request will be rerouted to the healthy pods.[8]

For testing outlier detection feature, do the following:

- Before testing delete all deployments:

```
sh delete_all.sh
```

- Start testing outlier detection feature with a script:

```
sh outlier_detection_test.sh
```

In the provided script all needed deployments are applied, after that HTTP 500 is injected for calculation service v1. In figure 33,34 is shown that v1 is unhealthy with 0% success rate, so other services have issues connecting to it. We can also see that requests are distributed between v1 and v2, but after a minute the v2 is ejected from the loadbalancing, what is shown in figure 35. Outlier detection was activated and the v1 of the calculation service was evicted from the load balancing pool for 1 minute, what increased the overall availability of the service. [8]

```

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: distributed-calculation-outlier
spec:
  host: distributed-calculation.default.svc.cluster.local
  trafficPolicy:
    outlierDetection:
      baseEjectionTime: 60s
      consecutiveErrors: 3
      interval: 20s
      maxEjectionPercent: 100

```

Figure 32: Outlier Detection configuration.

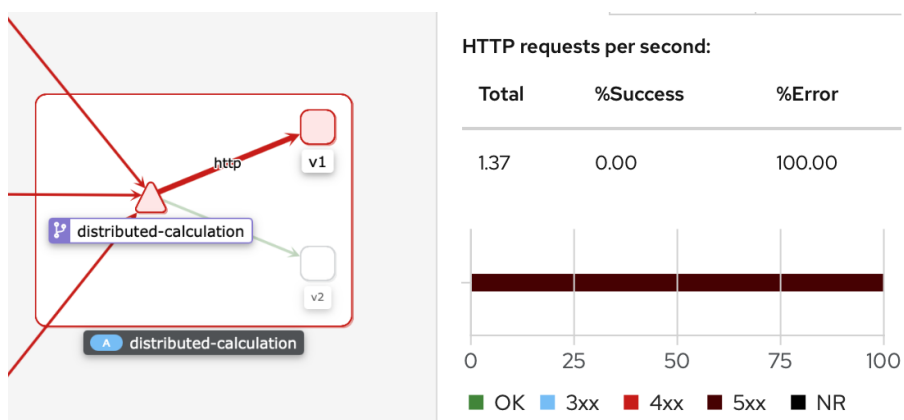


Figure 33: Kiali - Distributed calculation v1 is unhealthy.

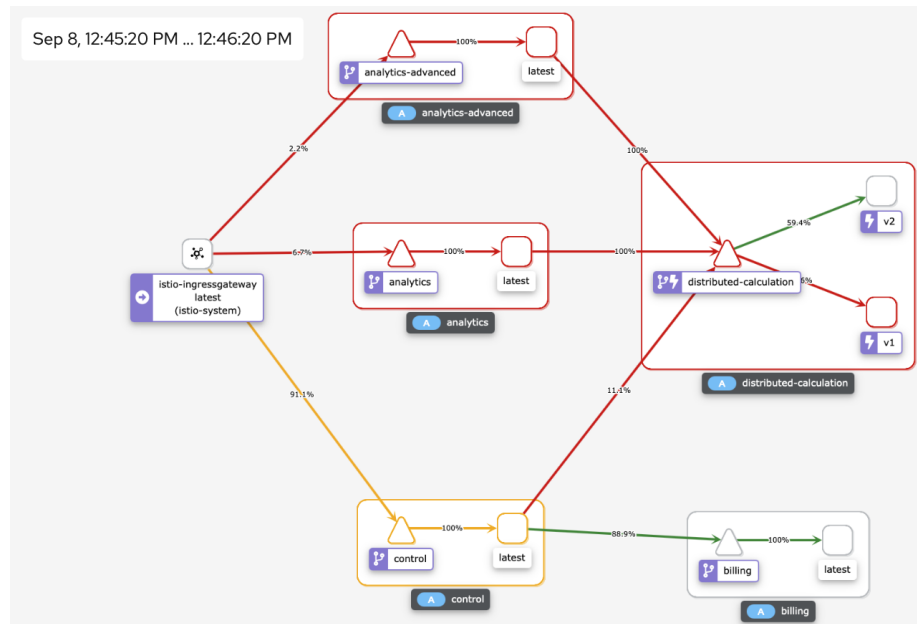


Figure 34: Kiali - Distributed calculation v1 before ejection.

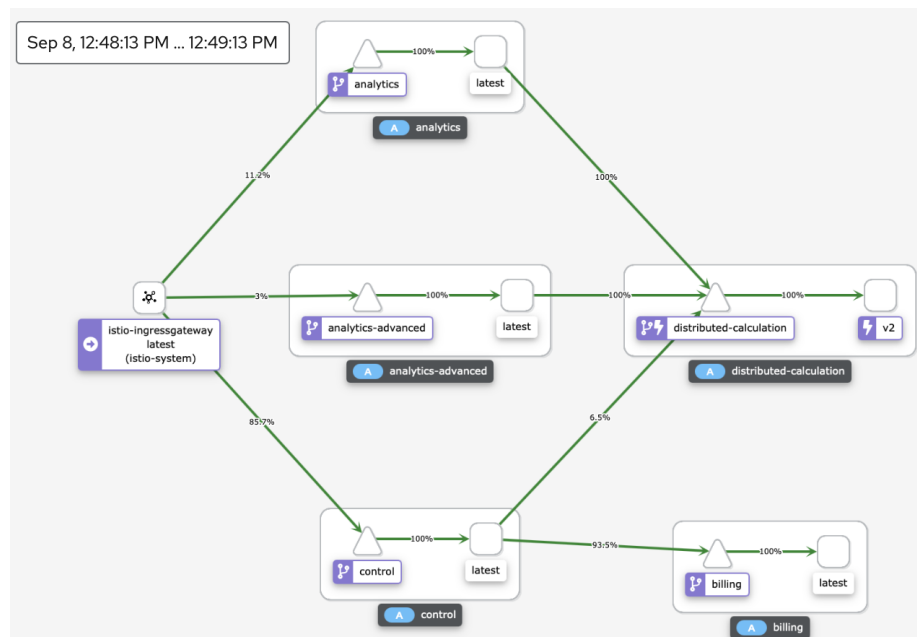


Figure 35: Kiali - Distributed calculation v1 ejected.



## 7 Discussion

For people, who never used Istio would be interesting to learn and try out all provided features. It was not easy to test some Istio mechanisms for me, cause in the beginning the applications was designed in a traditional fashion. As example, there is a retry implementation in calculation service written on Java with Spring framework. If a request is failed, there are 3 attempts with interval of 100 milliseconds. The Spring Cloud Netflix Ribbon library was used for this purposes. While designing the applications for the bachelor thesis I wanted it to be fault tolerant and it worked as expected, but this approach has some disadvantages:

1. It should be a part of application and for each endpoint it should be implemented additionally. There is no possibility to change the retry policy globally.
2. There are different implementations for each programming language and the library should be up to date. But for microservice approach, especially in polyglot stack it is not a good solution.
3. If any changes are needed, it is not enough to change the application configuration file and restart the service. Adaption of existing code is needed, which can be difficult in some cases.

```
@Service
public class RetryService {

    @Autowired
    private IFeignClient iFeignClient;

    @Retryable(value = {Exception.class}, maxAttempts = 3, backoff = @Backoff(100))
    public String retryWhenError(URI uri) {
        return iFeignClient.health(uri);
    }
}
```

Figure 36: Retry with Spring Netflix Ribbon

But with Istio it can be configured on a global level without changing the source code. The initial implementation with service repository MS made health checks of every registered service in 100ms interval, if one of the services was not reachable it was ejected from the list of registered services. Each registered microservice checked the status of the service repository, and if there were no response within 2 minutes the service was switched off. One crucial microservice and the additional logic implemented in each microservice was replaced by Istio features, what makes this technology very useful.

## 8 Conclusion

Different open source solutions such as Linkerd, Istio and Consul in the Kubernetes ecosystem for service meshes were discussed in the bachelor's thesis. Each of them has its own pluses and minuses. We have also discussed the difference between monolithic and microservice architecture. However, the goal of the thesis was deploying distributed application with Kubernetes and testing the resiliency features of Istio.

Using any of the above mentioned service meshes will put your DevOps teams in a better position to thrive as they develop and maintain more and more microservices. [18] From the demonstrated tests we can see, that Istio provides a lot of useful features. As part of the thesis the distributed calculation service was implemented and deployed with Kubernetes and Istio. Various features such as fault injection, retries, canary deployment and outlier detection were tested and expected results of the testing was introduced above. Istio is a new technology, which is great, but not mature. There are a lot of open bugs there. Distributed calculation service contains at most 10 simple microservices connected with each other, but sometimes 16 GB of RAM was not enough to handle it, what can be a problem in big projects.

## 9 Future Work

In this bachelor thesis simple, basic Istio resiliency features were tested and documented. In the future work more resiliency features can be tested and discussed. As mentioned above there are at least 3 open source solutions of service meshes with its features, pros and cons. A future work could contain a practical comparison of service meshes by following parameters:

- Traffic Management
- Monitoring and Observability
- Architecture
- Testing

It is interesting to see how Istio will be implemented and integrated in existing production solutions to understand the difference between theory and practice.

The community is focused on continuing to make Istio easy to use and as transparent as possible, with little or zero configuration [21]. Istio is promising and in my opinion will be widely used in the near future.

## References

- [1] CALCOTE, L., AND BUTCHER, Z. *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. O'Reilly Media, 2019.
- [2] <https://harness.io/blog/continuous-verification/blue-green-canary-deployment-strategies/>.
- [3] CHEN, L. Microservices: Architecting for continuous delivery and devops. In *2018 IEEE International Conference on Software Architecture (ICSA)* (2018), pp. 39–397.
- [4] <https://www.consul.io/docs/intro>.
- [5] DRAGONI, N., LANESE, I., LARSEN, S. T., MAZZARA, M., MUSTAFIN, R., AND SAFINA, L. Microservices: How to make your application scale. In *Perspectives of System Informatics* (Cham, 2018), A. K. Petrenko and A. Voronkov, Eds., Springer International Publishing, pp. 95–104.
- [6]
- [7] <https://dzone.com/articles/chaos-testing-your-microservices-with-istio>.
- [8] <https://dzone.com/articles/istio-circuit-breaker-with-outlier-detection>.
- [9] <https://istio.io/latest/docs/tasks/traffic-management/egress/egress-gateway/>.
- [10] <https://www.ibm.com/cloud/learn/containerization>.
- [11] <https://istio.io/latest/docs/tasks/traffic-management/fault-injection/>.
- [12] <https://istio.io/latest/docs/setup/getting-started/>.
- [13] KHATRI, A., KHATRI, V., NIRMAL, D., PIRAHESH, H., AND HERNESSE, E. *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing, 2020.
- [14] <https://medium.com/@goodrebels/to-go-or-not-to-go-micro-the-pros-and-cons-of-microservices-7967418ff06> (accessed 15.08.2020).
- [15] <https://medium.com/microservicegeeks/an-introduction-to-microservices-a3a7e2297ee0>.
- [16] <https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f>: :text=%E2%80%9CObject%2DOriented%20Programming%E2%80%9D%20(,his%20Sketchpad%20
- [17] NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*, 1st ed. O'Reilly Media, February 2015.
- [18] <https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/>.
- [19] <https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh>.
- [20] <https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh> (accessed 15.08.2020).
- [21] <https://sdtimes.com/softwaredev/guest-view-5-reasons-to-be-excited-about-istios-future/>.
- [22] WOLFF, E. *Microservices: Flexible Software Architecture*. Always learning. Addison-Wesley, 2017.
- [23] YOUSIF, M. Microservices. *IEEE Cloud Computing* 3, 5 (2016), 4–5.