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

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 well 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^{th} 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 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”" [4]. 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 worldewide 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 [3]. 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
- 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

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 [1].

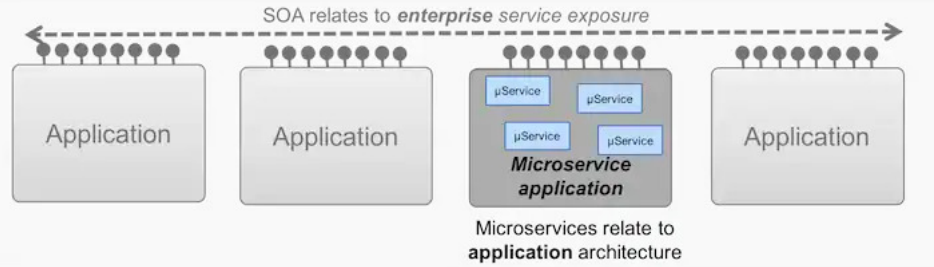


Figure 1: SOA vs. Microservice

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 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 peace 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[2].

[1] Link to IBM image

Adavantages and disadvantages of Microservice Architecture	
PRO	CON
- Versatile — microservices allow for the use of different technologies and languages	- Because the components are distributed, global testing is more complicated
- Easy to integrate and scale with third-party applications	- 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
- Microservices can be deployed as needed, so they work well within agile methodologies	- Microservices require experienced developers with a very high level of expertise
- Solutions developed using microservice architecture allow for fast and continuous improvement of each functionality	- Thorough version control is required
- Maintenance is simpler and cheaper — with microservices you can make improvements or amendments one module at a time, leaving the rest to function normally	- Microservice architecture can be expensive to implement due to licensing costs for third party apps
- 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	
- 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	

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[2] - "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." [1] 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 [1]

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[3] of cloud services:

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

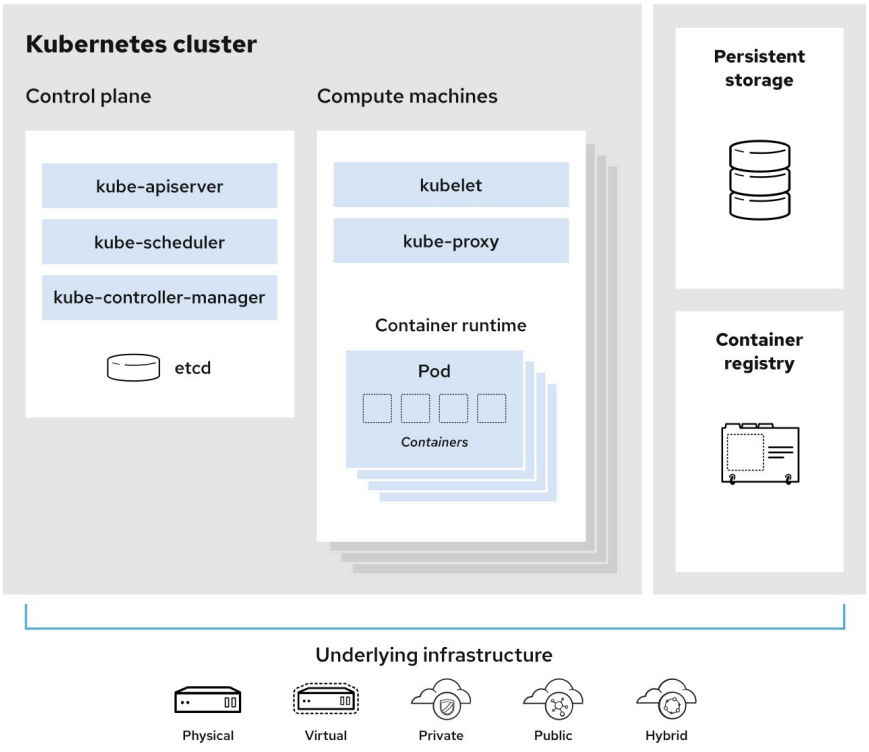
Cloud computing caused the appearance of new software concept - Cloud native applications that entirely reside in clouds. Microservices wrapped in containers are naturally fitting into this concept, as they are made of independent modules, each of which can be deployed to the cloud and leverage its functionality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

[2] https://www.ibm.com/downloads/cas/VG8KRPRM
[3] 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. [4]



[4]Link to Red-Hat

4 Service Meshes & ISTIO

Even though Kubernetes helps to orchestrate containers, it is still hard to track how individual microservices interact with each other. Kuberentes 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[5].

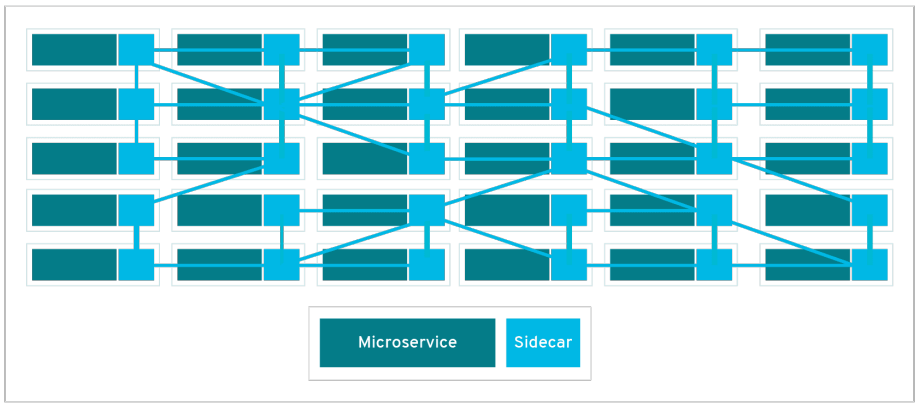


Figure 2: Proxy network[5]

We build it in to an app as an array of proxies. Requests are routed between microservices through proxies in their own infrastructure layer. This proxies are also called "sidecars" since they run not within but alongside microservices. All together this 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.

5 Implementation

5.1 Technology stack

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

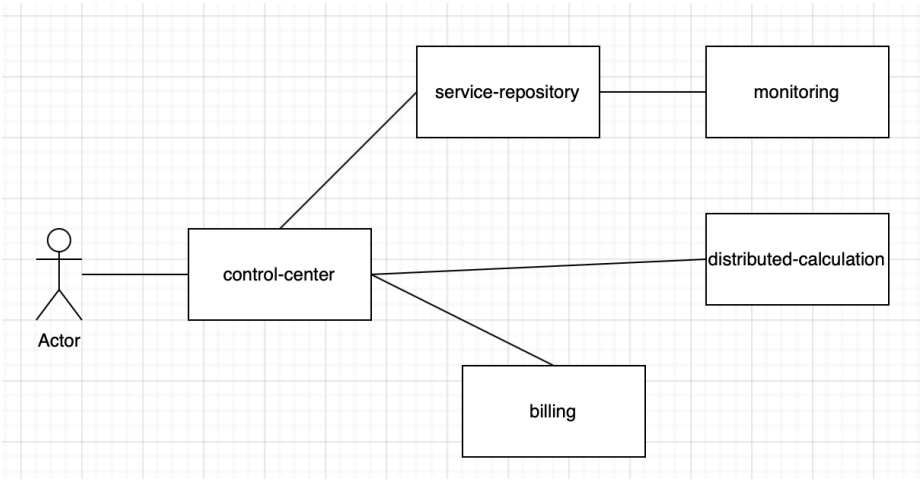


Figure 3: Distributed Calculation Cloud

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. (link-dse) 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 (link-dse), 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. For example 3 calculation services can be registered, so a user can choose which one is better for his/her purposes.

HTTP	Path	Request Body	Request Parameters	Consumes
POST	/register	msName: String, msPort: Integer, msDescription: String, msFunction: String, msType: String	-	application/json
GET	/ getRegisteredServices	-	-	
GET	/ findServiceByFunction	-	function	
GET	/getMetrics	-	-	

Figure 4: Service Repository API

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.
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.
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.
    - 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 center requested a list of microservices and its endpoints, which are active and support the required calculation. The information about billing service was also requested.
    - 1.2 All requested information was provided by 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	/timeout	-	n	-
POST	/responseCode	-	n	-

Figure 5: Distributed Calculation Service API

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 6: Billing Service API

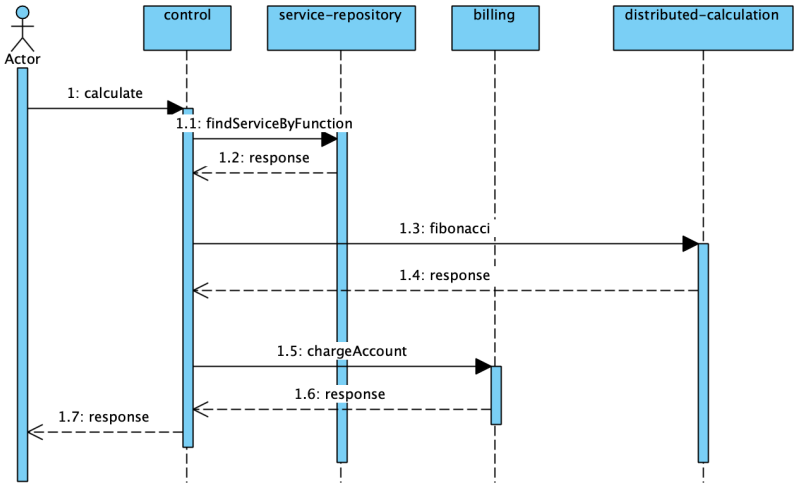


Figure 7: Successful calculation flow

- 1.3 Control panel sends a calculation request to distributed calculation service.
- 1.4 The result of calculation is returned to control panel.
- 1.5 The User is charged.
- 1.6 OK from the billing service
- 1.7 The result of calculation is returned to the user.

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.
6. Service Dashboard. The service provide a minimalistic website, where a user can monitor the service availability and workload.

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 endpoints in the distributed calculation service were added: Now it is possible to set a response code, with which the MS should answer and also it is possible to set a response timeout for the microservice.

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 8: Service Control Center API

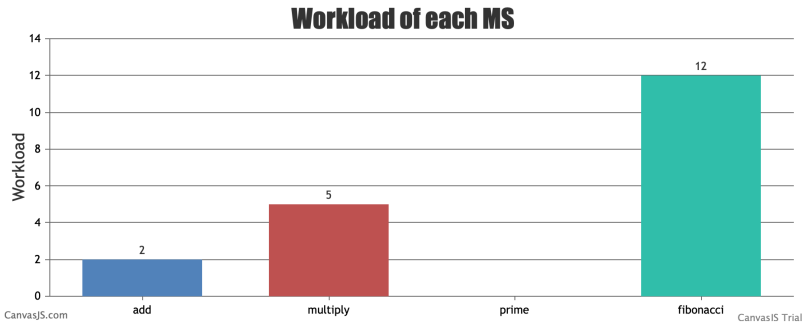


Figure 9: Service Control Center API

3. Health-checks were added to each MS.
4. New simple application for generating behaviour of users/clients was implemented.

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: (link-sitio https://istio.io/latest/docs/setup/getting-started/)

  ```
  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-1-deployment.yaml;
  kubectl apply -f k8s/billing-deployment.yaml;
  kubectl apply -f k8s/control-center-deployment.yaml;
  ```

- Set the ingress ports:

  ```
  export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{
  export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jso
  ```

6 Evaluation

6.1 Canary deployment with Istio

Done. Needs to be described. Screenshots from Kiali, Grafana.

6.2 Fault Injection

Done. Needs to be described. Screenshots from Kiali, Grafana.

6.3 Retries with Istio

Done. Needs to be described. Screenshots from Kiali, Grafana.

6.4 Circuit Breaker

Done. Needs to be described. Screenshots from Kiali, Grafana.

References

[1] https://www.ibm.com/cloud/learn/containerization (accessed 15.08.2020).

[2] https://medium.com/@goodrebels/to-go-or-not-to-go-micro-the-pros-and-cons-of-microservices-7967418ff06 (accessed 15.08.2020).

[3] https://medium.com/microservicegeeks/an-introduction-to-microservices-a3a7e2297ee0 (accessed 14.08.2020).

[4] https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f: :text=%E2%80%9CObject%2DOriented%20Programming%E2%80%9D%20(,his%20Sketchpad%20T (accessed 14.08.2020).

[5] https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh (accessed 15.08.2020).