Universidade de Aveiro

Portal for Services Mesh in Virtualized Environments

Alexandre Paiva (89908), Rafael Carvalho (93227), David Bicho (93215)

David Raposo (93395), Guilherme Lopes (93393)

Projeto em Engenharia de Computadores e Informática

Departamento de Eletrónica, Telecomunicações e Informática

June 12, 2023

Contents

List of Figures

Chapter 1

Introduction

In an era of global digitalisation, real-time management and oversight of operations across multiple locations has become crucial. Our project is designed to meet this demand, implementing a robust multi-cluster solution based on Kubernetes Kind and Istio Service Mesh.

The foundation of our project is a network of clusters, each representing a different country. Within these clusters, individual nodes simulate factories engaged in the production process. Echoing an Industry 4.0 environment, these factories continuously provide real-time updates on input materials, such as wood, and output products, like furniture, to a central Flask web portal.

The primary cluster hosts this Flask web portal, serving as the operational hub for monitoring each factory's activity. This centralized system communicates with all other clusters, assimilating their real-time data, offering an efficient and comprehensive overview.

The portal caters to various user roles, each with their designated authority. Each country's factory manager can access their respective factory's specific settings through unique login credentials. On the other hand, office personnel have wider access to data from all factories across all clusters, providing a comprehensive view of global operations.

The main objective of our portal is to offer a user-friendly interface for Kubernetes data, enhancing the accessibility and usability of the system. Although our project simulates an Industry 4.0 example for ease of understanding, the architecture incorporates robust microservices features.

The implementation of the Istio Service Mesh provides additional benefits such as improved traffic management, advanced load balancing, and service-to-service authentication. Its resilience and fault-tolerant design ensure system stability even in the event of factory failures, avoiding total system crashes and maintaining smooth operations.

Developing this sophisticated and innovative solution required a deep understanding and diligent application of a suite of technologies, including Kubernetes, Istio, and Flask. The task, although challenging, has culminated in a comprehensive system that has the potential to revolutionize factory management and oversight.

Chapter 2

The State of the Art

2.1 Service Mesh Provisioning Technologies

To delve deeper into the essence of a Service Mesh[2], we need to understand the complexities it seeks to mitigate. Our project, a robust multi-cluster solution leveraging Kubernetes and Istio, showcases the critical role of a Service Mesh. Within our system, it fosters seamless inter-cluster communication, facilitates real-time data management, and contributes significantly to the robustness and resilience of the entire setup.

At the heart of our system's architecture lies the microservices design principle. Each cluster, simulating a factory within a certain country, operates as an independent microservice. Such a design provides us with key advantages including: independent scalability, de-coupled release cycles, technology flexibility, and clear role assignment. This allows us to have granular control and flexibility in simulating the operations of individual factories.

One of the key pillars supporting the scalability and resilience of our system is the use of Istio, a powerful Service Mesh, in conjunction with Kubernetes (K8s). Istio's features and capabilities ensure that the services within our simulated factories stay agile, operational, and are able to adjust dynamically to varying capacity needs.

As our system continues to grow and evolve, it's inevitable that we'll encounter new challenges such as service discovery, traffic management, security & policy management, and observability. Istio has been specifically designed to tackle these problems effectively.

Istio brings immense value to our service discovery mechanism by maintaining a service registry, which effectively tracks the network locations of all available services. This greatly facilitates communication within the factory simulations, ensuring data integrity and timeliness.

In terms of traffic management, Istio's advanced features enable dynamic routing and load balancing, ensuring efficient communication between services. This is essential in our factory simulations, where real-time data from each factory needs to be routed efficiently to the central portal.

Security is a paramount concern in any microservices-based architecture. Istio excels in this domain by providing service-to-service communication in a secure manner through automatic mTLS encryption. This ensures robust protection for all inter-service interactions within our multi-cluster setup.

Another critical aspect managed by Istio is policy enforcement. Istio's policy enforce-

ment capability allows us to exercise control over how services interact with one another, providing us the ability to easily manage access and permissions.

Lastly, observability, facilitated by Istio's rich telemetry data, provides effective monitoring, tracing, and logging. This enables our system to detect and resolve potential issues in a timely manner, enhancing the overall system reliability.

Outsourcing these critical functionalities to the Istio Service Mesh allows our microservices to focus on their core tasks: simulating factory operations and providing real-time updates to the central portal. This separation of concerns makes our system flexible, scalable, and robust, equipping it to handle the intricacies of a multi-cluster factory simulation.

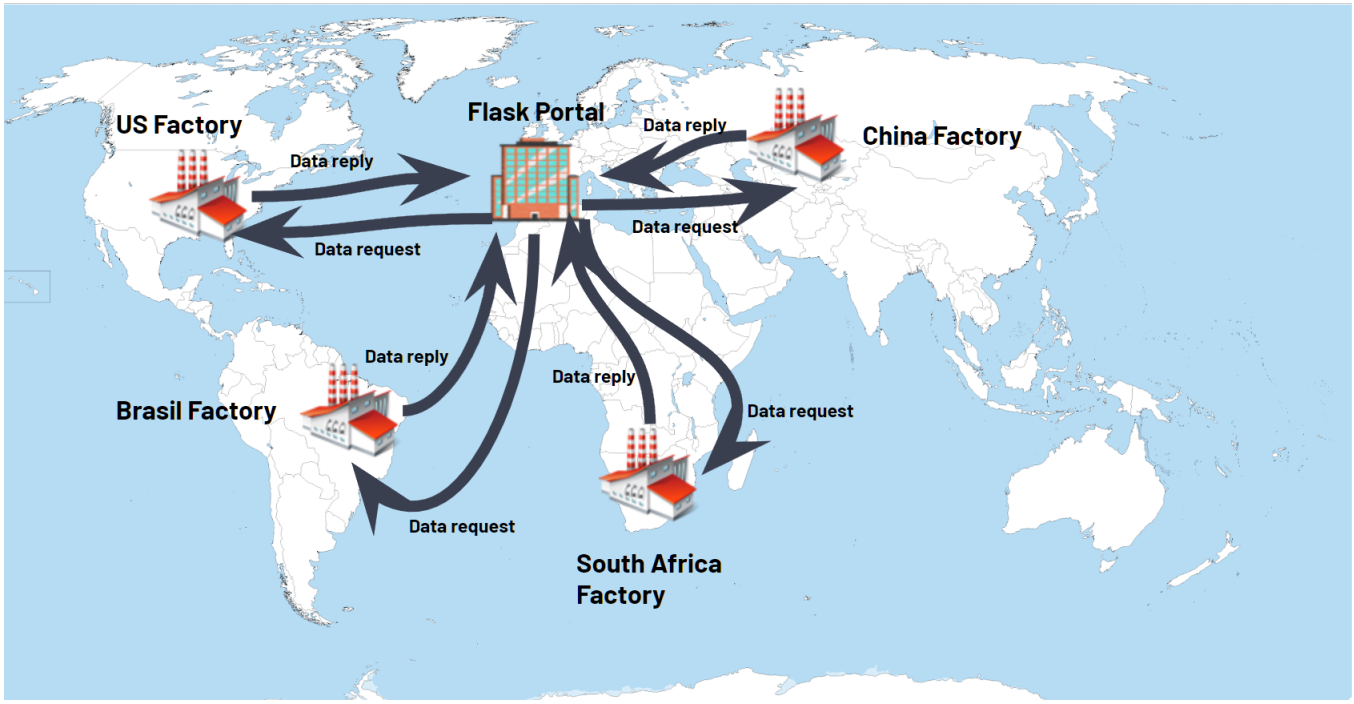

Figure 2.1: *Example of the flow of our multi-cluster project implementation. Our portal requests the data from the factories and the factories send that data through an API that is directed by the service Mesh (ISTIO)*

2.1.1 KongHQ

KongHQ [3] is a Kubernetes and Service Mesh provider, it powers trillions of API transactions for leading organizations globally through their end-to-end API platform. They are able to do this by using K8s and custom Service Meshes as part of their solutions. They also offer the ability to move from monolith to microservices. This is not what our project aims to do, move a monolith application to a microservices, but only implementing part of their services with Kubernetes, a Service Mesh (Istio) and while KongHQ implemented their own solutions our project aims to using open-source ones.

The **Kong Gateway** is a API gateway built for multi-cloud and hybrid, and optimized for microservices and distributed architectures. It's built on top of a lightweight proxy to deliver unparalleled latency, performance and scalability for all microservice applications regardless of where they run. It allows the ability to exercise granular control over all traffic with Kong's plugin architecture. This is similar to what our project aims to implement.
The Kong Gateway allows is built on top of a lightweight proxy to deliver unparalleled latency, performance and scalability for all microservice applications regardless of where they run. It allows the user to exercise granular control over the traffic with Kong's plugin architecture

KongHQ is not exactly what is going to be done in our project open-source technologies were used unlike KongHQ who have their own technologies developed and a wider array of services to provide that is more than the goal of our project. They do provide a full solution of what our project aims for but with less customization.
A huge disadvantage is that they do not allow manually handling of the containers and the configuration of the cluster and service mesh. Despite simplifying things for anyone who wishes to use Kubernetes or Service Meshes it does not meet the goal of our project. KongHQ also does not allow us to access the source code to understand the technology behind it.

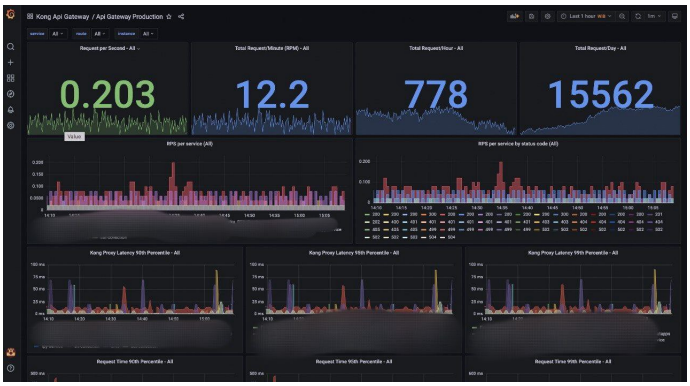

Figure 2.2: *Kong Gateway Dashboard*

2.1.2 Red Hat OpenShift

Red Hat OpenShift[4] is a cloud computing platform as a service (PaaS) developed by Red Hat. It's based on the Kubernetes container orchestration platform, and it provides several additional tools and features designed to help organizations develop and deploy applications more quickly and easily.

OpenShift includes a web-based console for managing applications and a command-line interface (CLI) for managing the platform itself. It also provides support for a range of programming languages and frameworks, including Java, Node.js, Ruby, and Python.

OpenShift is a partial solution of what our system provides, missing customization, and the ability for the user to control everything through a single portal.

Similar to KongHQ, OpenShift provides services related to Service Meshes and Kubernetes but does not allow manually handling of the containers and the configuration of the cluster, while it simplifies life for anyone who wishes to use Kubernetes or Service Meshes and is a big plus when it comes to facing competition in the business market it does not meet the goal of our project which is understanding how the deployment is done.

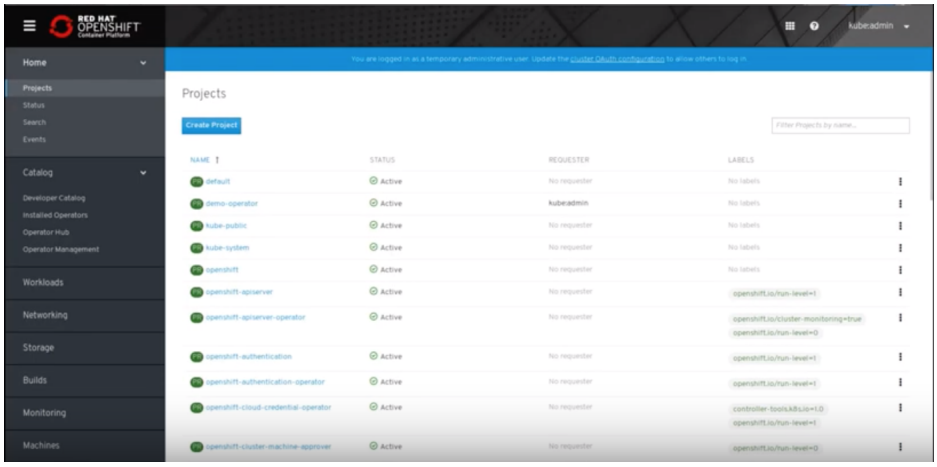

Figure 2.3: *Red Hat OpenShift Dashboard*

2.1.3 AWS App Mesh

AWS App Mesh [5] provides application-level networking so the users services can communicate across multiple types of compute infrastructure.

AWS App Mesh provides consistent visibility and network traffic controls and helping deliver secure services by acting as a Service Mesh. App Mesh removes the need to update application code to change how monitoring data is collected or traffic is routed between services.

This service is part of our solution, it would substitute the Istio Service Mesh and the custom Kubernetes Cluster deployment.

Again, the App Mesh they provide is their own custom Service Mesh, that most certainly achieves the same goal as the open source technology that our system is using (Istio). They do not expose the source code or allow direct and fine control over the deployment and everything is facilitated for the end-user, which is perfect for end-users but does not meet the goals of understanding how a service mesh is deployed and used.

Figure 2.4: *AWS App Mesh Dashboard*

2.1.4 IBM

IBM [6] deliver technology lifecycle services for IBM Systems products as well as open source and leading vendor systems and software.

With **IBM** and our project theme in mind it's best to be focusing on their Cloud Kubernetes Service, where it's possible to deploy secure, highly available clusters in a native Kubernetes experience. They allow to implement the Istio Service Mesh just like our system is doing in our project. It allows to deploy a scalable web app, monitor it and analyze logs and allow it to be deployed continuously using a CI/CD pipeline.
Similar to the above providers, IBM provides service mesh and Kubernetes cloud solutions but do not allow full control over them which, again, regarding our project does not meet the goals of understanding and learning.

Also, IBM provides the same partial solution as AWS mentioned above, their solution would substitute the Istio Service Mesh and the custom Kubernetes Cluster deployment.

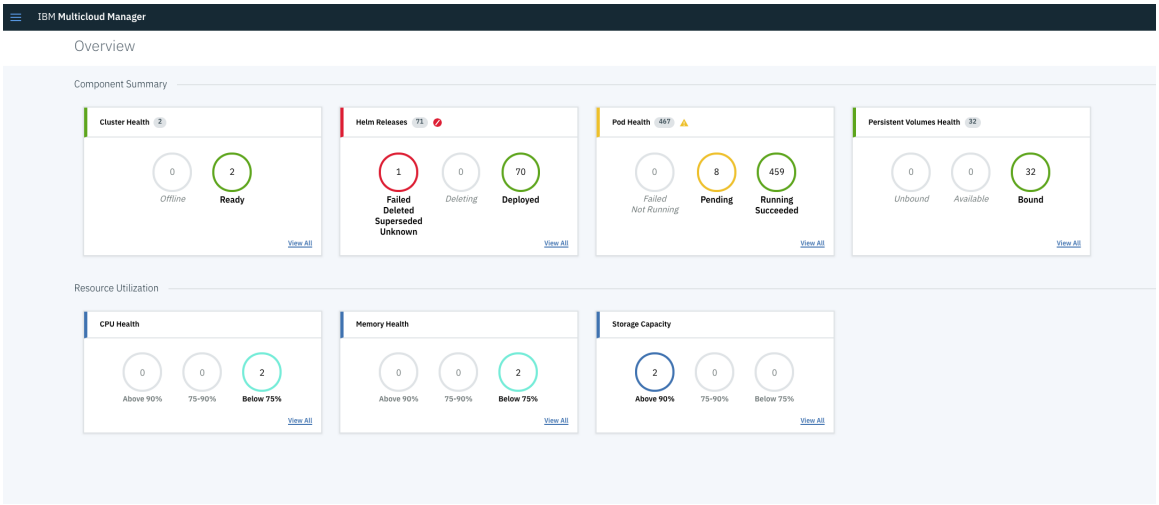

Figure 2.5: *IBM Cloud Kubernetes Dashboard*

2.2 Spotify

Spotify[7] is a large well known audio-streaming platform that has millions of daily active users.

2.2.1 Challenge

From early stages Spotify recognized the importance of containers and deploying microservices in a cloud native environment, for that reason Spotify adopted Helios[8] (which is a Docker orchestration platform for deploying and managing containers across an entire fleet of servers) to manage the containerized environments. However, Helios could not keep up with the exponential growth of Spotify.

2.2.2 Solution: Kubernetes

By late 2017, for Spotify it became clear that having a small team working on the features was just not as efficient as adopting something that was supported by a much bigger community.
So in 2019 they migrated to Kubernetes and increased their performance instantly.
Kubernetes was also a more feature-rich than Helios. Plus, they wanted to benefit from added velocity and reduced cost, and align with the rest of the industry on best practices and tools. At the same time, their team wanted to contribute its expertise and influence in the flourishing Kubernetes community. The migration, which would happen in parallel with Helios running, could go smoothly because Kubernetes can fit very nicely as a complement and now as a replacement to Helios.

2.2.3 Conclusion

With this update it allowed the Spotify team to not only improve their operational metrics, such as lead time, deployment frequency, time to resolution, and operational load, but also provided them with the necessary support to deploy the technology.
With Kubernetes, not only was the Spotify team able to scale their topline operational metrics, lead time, deployment frequency, time to resolution, and operational load but were also able to get all the help to deploy the technology.
The Kubernetes team was proactive in providing support and almost everybody was available to help the Spotify team with the technical know-how and expertise required while deploying a particular service. This served as a source of resurrection and motivation for the Spotify team to further scale-up their operational efficiency.
The Spotify team leveraged several Kubernetes APIs and other extendable functionalities to aid and interface with their legacy infrastructure. The integrations were hassle-free and easy for the Spotify team. Because of saving so much time with Kubernetes, the Spotify team was able to experiment with solutions such as gRPC[9] (A high performance, open source universal RPC framework) and Envoy[10] (An open source edge and service proxy, designed for cloud-native applications).

2.3 AutoTrader UK

Auto Trader UK[11] began in 1977 as the premier automotive market magazine in the United Kingdom. When it pivoted to an online presence near the end of the 20th century, it grew to become the UK's largest digital automotive marketplace.

2.3.1 Challenge

Changing requirements precipitated Auto Trader UK's migration to containerized applications using Istio as a service mesh. One of the most pressing reasons was the recent focus on GDPR[**?**] (General Data Protection Regulation).
AutoTrader wasn't satisfied with just typical perimeter security. It aspired to also encrypt all traffic between microservices, even those in the same local network, using mutual-TLS. The effort felt significant for a primarily custom-built on-premises private cloud infrastructure operating at Auto Trader's large scale.

2.3.2 Solution: Istio and Kubernetes

The Auto Trader UK Platform team worked on a proof of concept implementation of mTLS[12] (mutual TLS) for the on-premises private cloud. As expected, implementation was a laborious task. They decided to experiment with a container-based solution that could leverage a service mesh like Istio to manage mTLS for a key end-to-end slice of their microservice architecture. AutoTrader didn't have ambition to build and manage Kubernetes themselves, so they decided to run their experiment on Googles Kubernet Engine[13] which built and managed Kubernetes for them.
The container experiment was a success. Implementing encryption was taking weeks of effort on the private cloud but just days in the containerized project. The migration path to containerized services was clear for them.

2.3.3 Results: Confidence and Observability

Istio gave Auto Trader UK the confidence to deploy more and more applications to the public cloud. Istio allowed them to consider services in aggregates instead of as just individual instances.
With increased observability, they had a new way to both manage and think about infrastructure. They suddenly had insights into performance and security.
Istio was also helping them discover existing bugs that had been there all along, unnoticed. By fixing small memory leaks and small bugs in existing applications, they were able to bring significant performance improvements to their overall architecture.

2.4 Deployment Technologies

2.4.1 Kubernetes

Kubernetes[14], or K8s, is a production-grade open-source container orchestration tool developed by Google to help manage the containerized/dockerized applications supporting multiple deployment environments like On-premise, cloud, or virtual machines.
K8s automates the deployment of containerized images and helps the same to scale horizontally to support high level of application availability.

Each container image in itself is a Microservice that needs to be managed and scaled efficiently with less overhead, this demand to handle thousands and thousands of containers became a tedious task for the organization.
This problem led to the evolution of K8s as one of the popular container orchestration tools because it makes the management of containers less challenging as it's needed to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start and it can end up with dozens, even thousands of containers over time. These containers need to be deployed, managed and updated.
It will be very difficult to do all things manually, so Kubernetes is the solution. Kubernetes provides the user with a framework to run distributed systems resiliently. It takes care of scaling and fail-over for the application, provides deployment patterns, and more.

- Package up the app and let something else manage it for us.
- Not worry about the containers management.
- App Containers can't live with single server deployment.
- Inter-host communication of containers.
- Deploying and updating software at scale.
- Better management through modularity.
- Auto healing.
- Logging and Monitoring.

The four key concepts to understand the architecture of Kubernetes are:

- **Nodes -** Nodes are the machines on Kubernetes. They can be either a physical or a virtual machine.
- **Pods -** Pods are composed of a group of one or more containers, the shared storage for them and their options.
- **Deployments -** Deployments are used to make updates on Pods. We can use them to bring up new Pods, change the image version of a container and even recreate the previous state if something goes wrong.

- **Services -** Service consist of a set of Pods and a policy that defines the access control. It is responsible for managing, deploying and operating containers integrates easily with other services.

Knowing all this raises the question of 'What can Kubernetes do?' and the answer is:

- **Service discovery and load balancing -** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Scaling -** Auto-scaling: Automatically change the number of running containers, based on CPU utilization or other application-provided metrics.
- **Manual scaling -** Manually scale the number of running containers through a command or the interface.
- **Self-healing -** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to a user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Storage orchestration -** Kubernetes allows to automatically mount the desired storage system, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks -** The desired state for the deployed containers can be described using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, it's possible to automate Kubernetes to create new containers for the deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing -** Provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. Tell Kubernetes how much CPU and memory each container needs. Kubernetes can fit containers onto nodes to make the best use of resources.
- **Secret and configuration management -** Kubernetes allows to store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. Our system can deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

Knowing this, in our project Kubernetes was deployed as a platform for managing containerized workloads and services, to maintain and handle our containers and clusters.

Kubernetes Command Line Tool (kubectl)

Kubernetes provides a command line tool for communicating with a Kubernetes cluster's control plane, using the Kubernetes API.
This tool is named kubectl. For configuration, kubectl looks for a file named config in the '/.kube' directory. Below on the Bash Figure2.12 an example bash file is show that will create the '/.kube' directory and copy the config files to it.

Master Node

The master node provides connections to the user interface (UI) and executes the control plane service, in our system case it runs Istio's control plane.
The control plane services manages and operates the Kubernetes cluster. It also has the API server for the cluster.[15]

The API server manages all administrative tasks in the master node. The API server receives users' REST calls.

Worker Node

The worker node is the runtime for the application. Through containerization and microservices, the applications run out of pods. Pods are managed on worker nodes, where they consume computing resources such as memory, storage, networking.

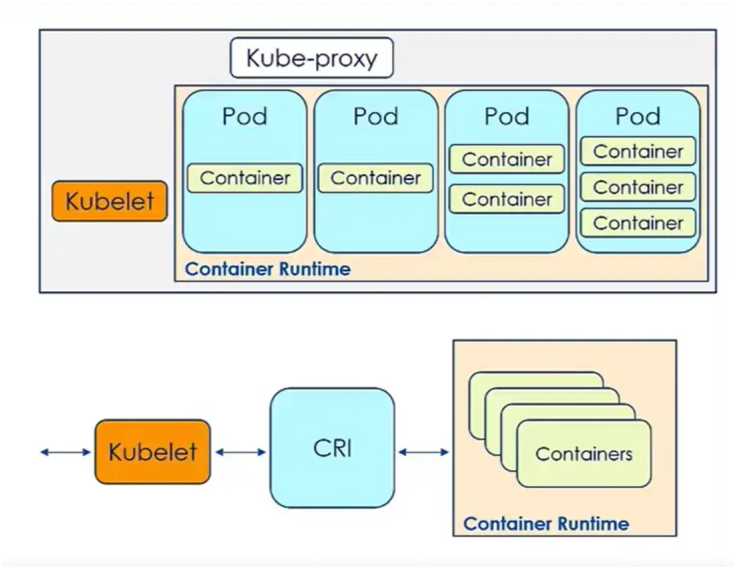

Figure 2.6: *Overview of the worker node*

The Kubelet is a service that communicates with the master node and the containers from the worker node. The kubelet connects to the container using a container runtime interface(CRI).
Kubernetes is a container orchestration engine, however, interestingly enough Kubernetes does not directly execute containers. Kubernetes needs a container runtime, on a node where it's is executing a pod and containers.
In our system CRI-O (**??**) will be used as the container runtime.
The kube-proxy[16] is the networking agent for each node.

2.4.2 Istio

Istio[17] is a Service Mesh. To explain Istio, first it's needed to explain what a Service

Mesh is.
Without Istio, one service makes direct requests to another and in cases of failure, the service needs to handle it by retrying, time outing, opening the circuit breaker, etc.

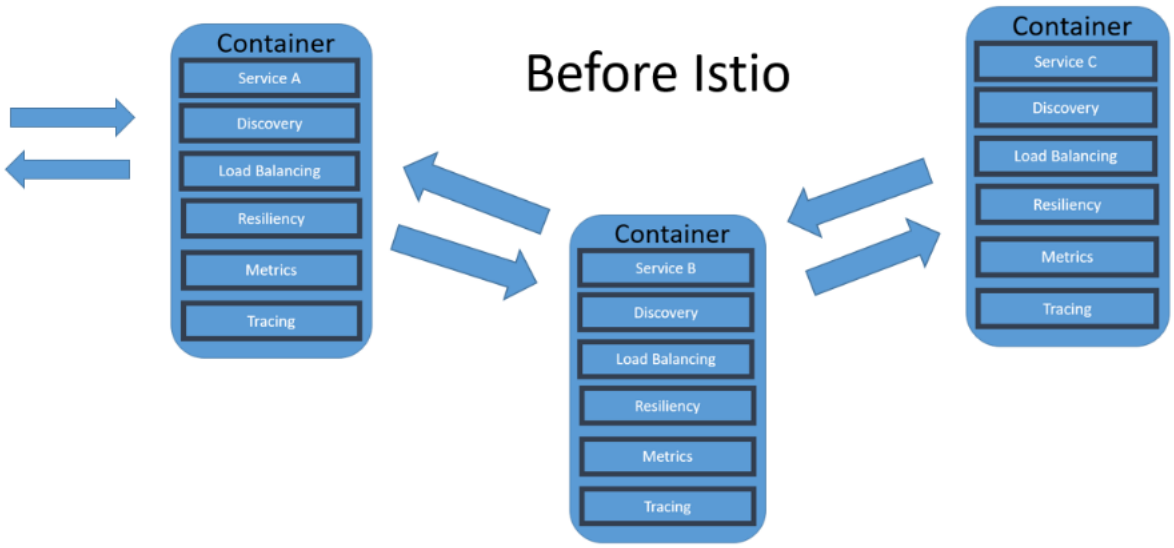

Figure 2.7: *3 containers in a cluster without Istio, where each has to handle load balancing, resiliency, etc... manually, increasing complexity.*

To resolve this, Istio provides an ingenious solution by being completely separated from the services, as mentioned above, and act only by intercepting all network communication. And doing so it can implement:

- **Fault Tolerance:** Using response status codes it understands when a request failed and retries.
- **Canary Rollouts -** Forward only the specified percentage of requests to a new version of the service.
- **Monitoring and Metrics -** The time it took for a service to respond.
- **Tracing and Observability -** It adds special headers in every request and traces them in the cluster.
- **Security -** Extracts the JWT Token and Authenticates and Authorizes users.

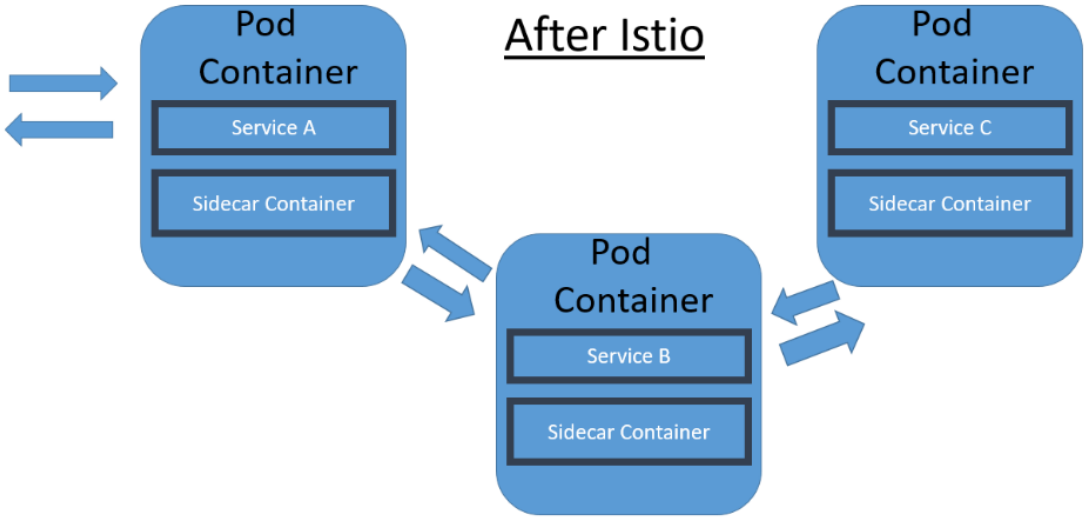

Figure 2.8: *The same 3 containers as above in a cluster with Istio, where now the load balancing, resiliency, etc... is controlled outside of the services, decreasing complexity.*

With this, in our project our system deployed Istio on top of each Cluster to manage communication between microservices.

2.4.3 Docker

Docker is an open-source platform that automates the deployment, scaling, and management of applications through containerization. At its core, Docker provides a way to run applications securely isolated in a container, packaged with all its dependencies and libraries. Docker enables developers to create these containers, and helps in deploying and running these containers on different environments, thereby ensuring consistency and operational efficiency.

In our project, Docker serves as a cornerstone for creating portable and consistent environments that house our microservices. Each Docker container simulates a factory environment encapsulating its own set of applications, libraries, and dependencies. This isolates each simulated factory from one another, thereby ensuring that each can operate and scale independently.

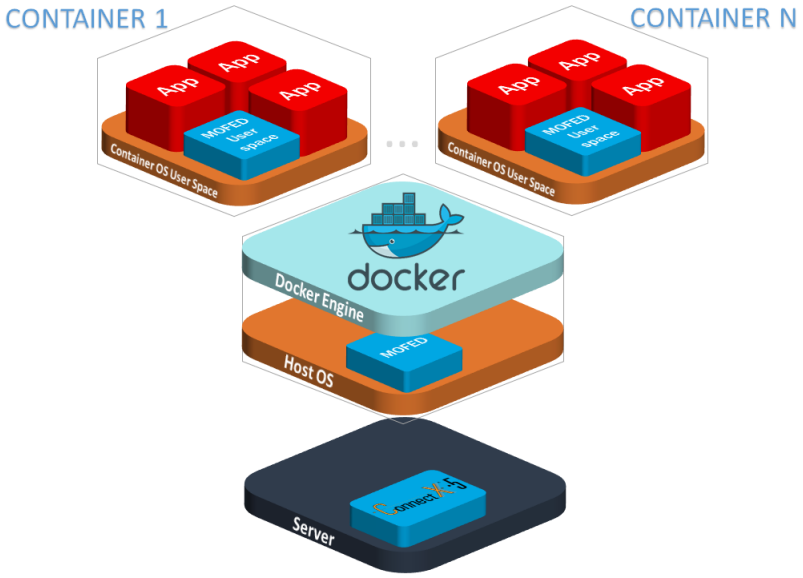

Figure 2.9: *Specifications inside of docker container*

2.4.4 Kubernetes IN Docker (KIND)

Kubernetes IN Docker, or KIND, is a tool for running local Kubernetes clusters leveraging Docker container nodes. It is primarily used for testing Kubernetes itself, but may be used for local development or CI.

In the context of our project, KIND enables us to instantiate our Kubernetes clusters. Each cluster, representing a factory located in a specific country, is deployed and managed using KIND. These KIND-managed Kubernetes clusters form the backbone of our distributed architecture, each running multiple Docker containers that simulate individual factories.

Moreover, KIND allows us to mimic a production-like Kubernetes environment on a local machine. This has been instrumental in our development and testing phases, as it allowed us to iteratively build and improve our system while keeping resource usage minimal. In addition, with KIND, we are able to mirror the state of each factory's operations in real-time by simulating different situations and observing the outcomes.

The usage of Docker and KIND together in our project underlines the power of containerization and cluster management technologies. This combination allows us to create a dynamic, scalable, and robust multi-cluster environment that can efficiently simulate a network of factories and their real-time operations.

2.5 Base Technologies

In this section, we describe the technologies used during the construction of the project. The section is divided in two subsections. The first section, Technologies in use2.5.1, describes briefly the technologies that were used on the final architecture. The next section, Technologies no longer used2.5.2,describes the technologies that were tested or considered use-full but did not meet the expectations in the development case.

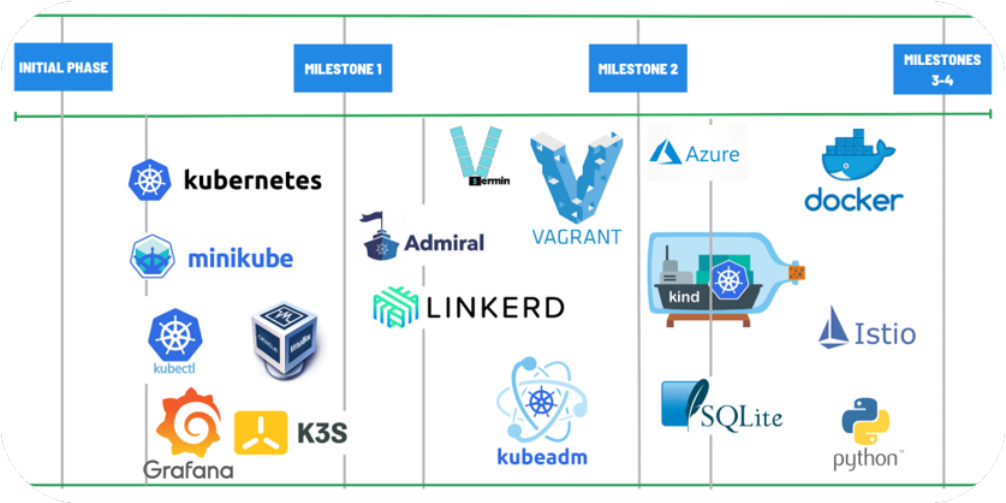

Figure 2.10: *The timeline that references the tecnologies used between the milestone deliveries.*

The figure above shows the technologies, and the more important ones will be explained in the next sections.

2.5.1 Technologies in use (Final Architecture)

Python

Python[18] is a versatile interpreted high-level programming language. It emphasizes code readability using significant indentation. Its constructs and object-oriented approach help programmers write clear, logical code for small and large-scale projects. By supporting multiple programming paradigms, including structured (particularly, procedural), object-oriented and functional programming it became an obvious choice. The group chose Python because of its versatility, easy learning curve, ample collection of modules and the group's familiarity with the language. It will be mainly used for creating the API server that connects to the microservices deployment. It will make use of the Flask[19] module and other modules available for Python.
Aside from this, Python is also used to build the back-end of the portal, making use mainly of the Flask Module.

Docker

Docker2.5.1 is an open-source platform that allows users to automate the deployment and

management of applications in containers. Containers are a form of virtual operating systems that runs applications in an isolated environment. In the project context, the most crucial functionality of Docker is Docker Images, which is a lightweight and executable package that contains the software and the application code to run. It serves as a blueprint to create Docker Containers. This images are used by running a Dockerfile, that contains the code to create the container. The next section, Kind2.5.1, takes advantage of this feature and creates Kubernetes[14] clusters inside the Docker Containers, creating a safe environment and allowing the creating of a multi-cluster platform using this tool.

Kind - Kubernetes in Docker

Kind2.5.1 is a tool that allow to create local Kubernetes[14] clusters using Docker container nodes. It consists of Go packages to implement cluster creating and image build, a command line interface and Docker images written to run Kubertes. To allow node management, Kind bootsraps each node with KubeADM. The main reason Kind was selected is that allows to create clusters with less resources from the host machine than any other option, and since we are planning to run a multicluster environment, storage and resources were limited.

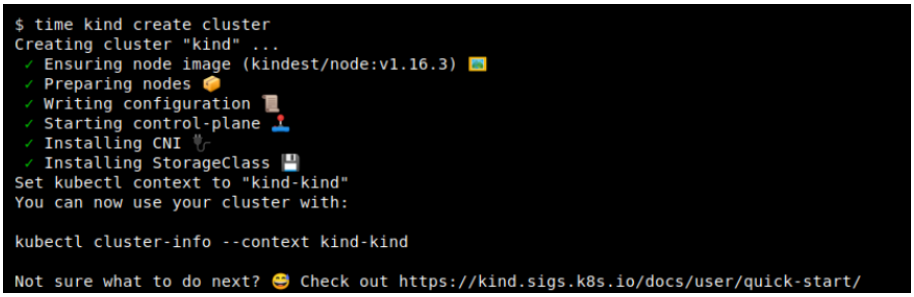

Figure 2.11: *The creation of a Kind Cluster and all the setups to ensure that the cluster is running as expected.*

Kind was the tool that allowed us to move one step further to our goal, using Docker Images 2.5.1 to create images and Kubectl2.4.1 command to deploy them in the selected cluster.

HTML/CSS/JavaScript

HTML/CSS/JavaScript will be used for our Website interface. The portal's front-end will be developed using these languages. We are mainly going to use templates to do this faster as the only goal is to make the platform look more professional. The HTML will be implemented using Flask so it can be dynamic and easily integrated with our API.

Bash

Bash[20], also known as Bourne Again Shell, is a shell program that is executed in a command line interface having as its main function to control operating systems. Bash scripting will be used to deploy kubectl inside the VMs which then will be used to install Kubernetes nodes, pods, configure and deploy using YAML files, install Istio and Admiral and basically control the clusters.

```
//Na pasta cluster1
//before everithing check the host ip and put it on the Cluster 1/Portal/host_ip.txt

sudo kind create cluster --config kind-config.yaml --name portal-cluster
sudo kubectl config use-context kind-portal-cluster
cd Portal
sudo docker build -t portal-deployment .
sudo docker tag portal-deployment istsmerafarlho/projpeci:portal-deployment
sudo docker push istsmerafarlho/projpeci:portal-deployment
sudo kubectl apply -f portal-deployment.yaml

sudo kubectl get services portal-service -o=jsonpath='{.spec.clusterIP}'



//Na pasta cluster2:
sudo kind create cluster --config kind-config-factory.yaml --name factory-cluster
sudo kubectl config use-context kind-factory-cluster
cd Factory
sudo kubectl config use-context kind-factory-cluster
sudo docker build -t factory1-deployment-cluster2 .
sudo docker tag factory1-deployment-cluster2 istsmerafarlho/projpeci:factory1-deployment-cluster2
sudo docker push istsmerafarlho/projpeci:factory1-deployment-cluster2
sudo kubectl apply -f factory1-deployment-cluster2.yaml

sudo docker build -t factory2-deployment-cluster2 .
sudo docker tag factory2-deployment-cluster2 istsmerafarlho/projpeci:factory2-deployment-cluster2
sudo docker push istsmerafarlho/projpeci:factory2-deployment-cluster2
sudo kubectl apply -f factory2-deployment-cluster2.yaml


sudo kubectl get services --all-namespaces
//check the one named portal service

DEPOIS INSTALAR O ISTIO:
cd ..
Install Istio on the primary cluster (portal-cluster):
  curl -L https://istio.io/downloadIstio | sh -
  export PATH="$PATH:/<path_to_istio_directory>/bin"
  istioctl install --set profile=demo -y
  kubectl label namespace default istio-injection=enabled
  Replace <path_to_istio_directory> with the path to your Istio directory.
Create and apply the Istio configuration for the portal service:
  kubectl apply -f portal-istio-config.yaml
  kubectl --context=<factory-cluster-context> create namespace istio-system
  kubectl --context=<factory-cluster-context> label namespace default istio-injection=enabled
  (Replace <factory-cluster-context> with the context of your remote cluster.)
  (obter os context names: kubectl config get-contexts)
NO CLUSTER DA FABRICA:
  istioctl --context=<factory-cluster-context> install --set profile=demo -y
  istioctl x create-remote-secret \
  --context=<factory-cluster-context> \
  --name=<factory-cluster-name> | \
  kubectl apply -f - --context=<portal-cluster-context>
```

Figure 2.12: *A Bash file containing Bash commands that need to be ran on specific node created.These are simple commands for making directories and copying files and running other Bash files.*

SQLite

SQLite is a software library that provides a relational database management system. The distinguishing factor of SQLite is that it is serverless, transactional and zero-configuration, meaning it doesn't require a separate server process or setup procedures to operate. This makes SQLite an incredibly light, yet powerful database solution.

In our project, SQLite serves as the backbone for managing our user accounts and roles for the portal. The Flask-based web portal utilizes SQLite to securely store user credentials, user roles, and other essential information. These roles are instrumental in managing access control within our system, with various roles associated with different levels of access and permissions.

SQLite's lightweight architecture and simplicity make it a perfect fit for our project, as it allows for efficient querying of user data without the overhead of more extensive database management systems.

YAML

YAML[21] is a popular programming language because it's human-readable and easy to understand. It can also be used in conjunction with other programming languages. Because of its flexibility and accessibility, YAML is used by all our Deployment Technologies.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: portal-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: portal
  template:
    metadata:
      labels:
        app: portal
    spec:
      containers:
      - name: portal-container
        image: istsmerafarlho/projpeci:portal-deployment
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: portal-service
spec:
  selector:
    app: portal
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

Figure 2.13: *In this YAML file, the Kubernetes deployment will create the pods and make the service visible outside the Kubernetes cluster by the node's IP address and the port number declared in this property.*

2.5.2 Technologies no longer used

Vagrant

Vagrant[22] is an open-source technology that creates and manages Virtual Machines, more specifically Virtual Box[23] Virtual Machines. Specifying the configuration of a virtual machine in a configuration file, Vagrant creates a virtual machine using one simple command. Almost all the interaction will be done through the command-line interface, where there is access to the virtual machines previously created. We will be using Vagrant to create Virtual Machines that will operate as nodes in a Kubernetes Cluster and to run bash commands in order to automate deployment.

Admiral

Admiral[24] provides automatic configuration and service discovery for multicluster Istio Service Mesh.
Istio has a very robust set of multi-cluster capabilities. Managing this configuration across multiple clusters at scale is challenging.
Admiral takes an opinionated view on this configuration and provides automatic provisioning and syncing across clusters. Let's look at the example below:

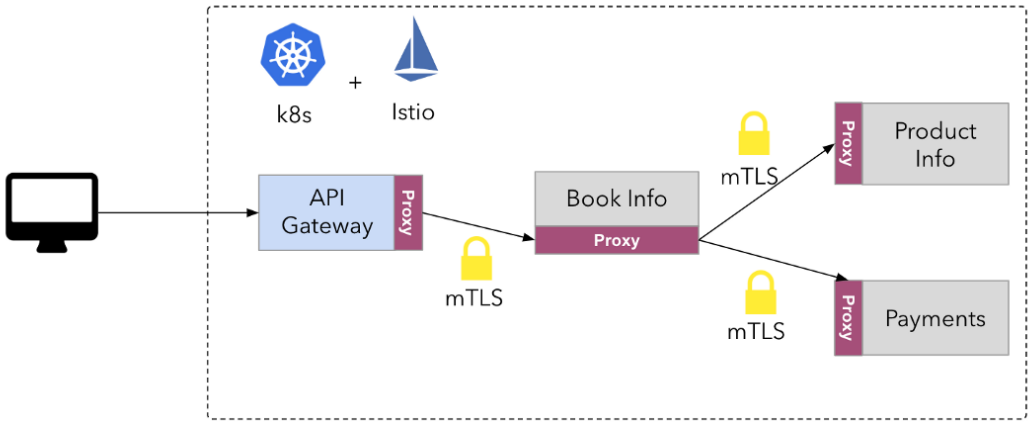

Figure 2.14: *A simple cluster example, in this case a 'Book Info' application is inserted which is a Book Store shop, with Istio installed. As seen it has a single cluster implementation with 3 services, each handling their own information and all connected to the API gateway.*

This simple setup worked well as a solution (both for our project goals and for the example given in the figure); However, as production deployments came and were analised, the requirements were much more complicated than a simple single cluster installation.

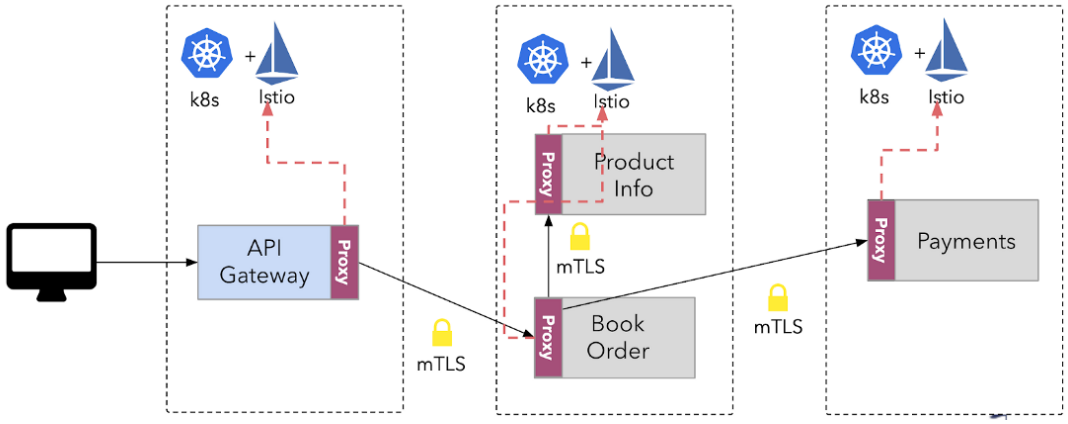

Figure 2.15: *Continuing the 'Book Info' application example, this would be a more realistic approach to the problem. A multi-cluster implementation of the services, where each rectangle with K8s is a independent cluster with Istio installed.*

With this deployment shown above, the proxies connect to the local Istio control plane. The Istio configuration is in each control plane to enable services to communicate between clusters. Another benefit of this topology is that the Istio config is collocated with the service's k8s deployment config, thus making troubleshooting more straightforward.
This is a good start but it was quickly realized the configuration for multi-cluster was heavily complicated and would be challenging to maintain over time.
Also, services would need several DNS names for the same service with different resolution and global routing properties. For example, "default.payments.global" would resolve locally first, and then route to a remote location (using topology routing). Also, payments service would also need names like "payments-west.global" and "payments-east.global" to resolve the respective region but enable failover to the other region. Such names would then be used for testing in one region during deployments and for troubleshooting.
All this sounds already complex... but there's more complexity.
What if, regarding the book info example in the figures, the payment services want to move traffic to the us-east region for a planned maintenance in us-west?
This would require the payments service to change configurations in all of their clients clusters updating DestinationRule locality settings. This would be nearly impossible to do without some automation.
Admiral is that needed automation. Admiral is the controller of Istio control planes. Admiral automates the configuration necessary to enable service to service communication by abstracting the cluster and deployment topologies from service discovery and consumption.

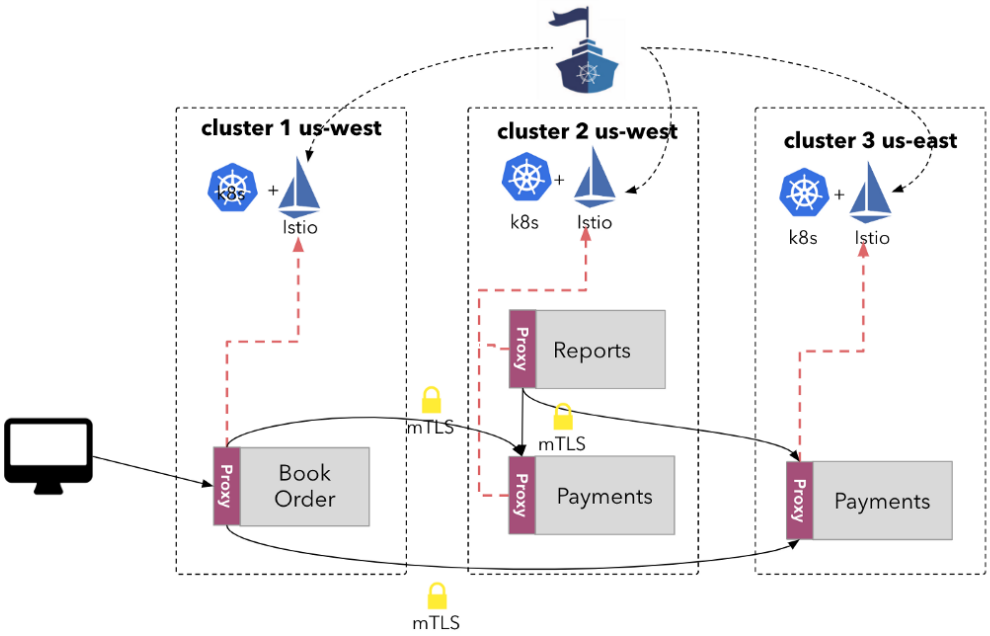

Figure 2.16: *Continued example of the Book Info application but now with Admiral implemented.*

Admiral provides automatic configuration for deployments spanning multiple clusters to work as a single mesh. Admiral also provides automatic provisioning and syncing of Istio configuration across clusters. This removes the burden on developers and mesh operators and helps scale beyond a handful of clusters.
Admiral also introduces a new type to control global traffic routing, with this new global traffic policy CRD (Custom Resource Definitions), now (regarding the book info example given in the figures) the payments service can update regional traffic weights and Admiral takes care of updating configuration in all clusters with the clients consuming the payments service. This can be done via YAML files, as seen below.

```
apiVersion: admiral.io/v1alpha1
kind: GlobalTrafficPolicy
metadata:
  name: payments
  namespace: admiral
spec:
  policy:
  - dns: default.payments.global
    lbType: failover
    target:
    - region: us-west-2/*
      weight: 10
    - region: us-east-2/*
      weight: 90
  selector:
    identity: payments
```

gistfile1.txt hosted with ❤ by GitHub view raw

Figure 2.17: *In this example above, ninety percent of the payments service traffic is routed to the us-east region. This Global Traffic Configuration is automatically converted into Istio configuration and contextually mapped into clusters to enable multi-cluster global routing for the service and the clients within the Mesh.*

We use YAML files to deploy, initiate and change nodes, pods, services and deployments for Istio, Admiral and Kubernetes.
Admiral provides a powerful new Global Routing functionality that was missing from the multi-cluster service mesh implementations, removes the need for manual configuration synchronization between clusters, and generates contextual configuration for each cluster. This makes operating a service mesh composed of as many Kubernetes clusters possible. Which is the reason it was implemented in our project. As our goal is to operate a service mesh composed of multiple Kubernetes clusters.

Azure

Azure is a cloud platform that aggregates a collection of servers an networking solutions, which can run a complex set of distributed applications. This platform allows to build, run, and manage applications across multiple clouds, on-premises, and at the edge, with the tools and frameworks that the user desires.
The main reason Azure was considered valuable was that it allowed to create Virtual Machines on the cloud. This VMs provide an economical and secure way to use the platform's powerful computing capabilities, scalability, high availability, and elasticity, to allow developers to

quickly provision new VMs for development or testing purposes without investing in physical hardware or software licenses.
With this feature, was possible to create Kubernetes clusters on cloud, without being dependent of local resources, and that anyone could access the virtual machine at any time, that solved the problem of being all together to work on the project in the computers that had the power to "feed" a multi-cluster environment.
But everything good ends fast, and since we only had one month of free trial in Azure, we could not afford to pay an account for six months, and this solution was discarded.

ProxMox

In the final stages of our work, it was suggested that we used ProxMox to deploy our project on IT servers. ProxMox is a open-source solution that offers support to virtualization in many operating systems. Instead of having our system running locally with many VMs it would be available an IT VM that would be running our project. This would be a good improvement for our project because if we wanted to scale our project, our own laptops wouldn't have the necessary resources like RAM and processors to run many different clusters, however despite this advantages, it was not possible to do virtualization inside virtual machines (creating new clusters, that are new virtual machines inside the host VM) ,and would increase exponentially the difficulty of the communication between clusters. For this reasons we decided to keep the project on local machines and reduce the number of clusters simultaneously running.

2.6 Concluding Remarks

Throughout this project, our technology choices have been driven by the complexity of our requirements, the need for scalability, stability, security, and the desire to utilize cutting-edge, open-source software. We adopted a microservices architecture using Kubernetes with multi-cluster support facilitated by KIND, all connected via the Istio service mesh.

Istio was chosen for several reasons. It runs natively on Kubernetes, supports high availability, implements a sidecar proxy pattern, and provides its own Gateway and Ingress. Furthermore, Istio's support for multi-cluster environments, comprehensive observability and traffic management features, in combination with its extensive documentation and popularity, made it an ideal choice for our project.

Initially, we considered using Admiral for managing the complexity of Istio's multi-cluster environments. Admiral, being a control plane for Istio, could automate much of the complexities associated with a multi-cluster Istio implementation. However, we eventually transitioned to using KIND (Kubernetes in Docker) which further streamlined our multi-cluster setup and management. By enabling us to run multiple, isolated Kubernetes clusters using Docker container 'nodes,' KIND effectively reduced the operational complexity, removing the need for additional tools like Admiral.

Kubernetes, the basis of our infrastructure, has been chosen for its stability, scalability, and security. As a globally recognized container orchestration tool, Kubernetes is an integral part of managing and deploying containers in our complex environment.

The entire system serves to demonstrate a practical application of a multi-cluster microservices architecture, replicating an Industry 4.0 example of various factories (clusters) communicating real-time updates to a central portal (main cluster). Although this deployment is complex, it showcases the robustness and potential of these technologies when used together.

Reflecting on the iterative and flexible nature of software development, it's crucial to note the role of Admiral in the initial stages of our project. While its functionalities were eventually superseded by KIND, its consideration helped us navigate the landscape of multi-cluster management. The adaptability and power of these open-source technologies have resulted in a final architecture that stands as a testament to our project's evolutionary journey.

As seen with Spotify (2.2) and Auto Trader UK (2.3), implementing Kubernetes with Istio can not only increase the application's performance but also give the development team time to focus on other actions, as seen on Spotify because of the time the Kubernetes implementation saved the Spotify team they were able to further scale-up their operational efficiency.
And with AutoTrader UK, the number of advantages that came along with implementing a Service Mesh with Kubernetes are immense, from confidence to observability they saw a huge increase in their performance and security.

Chapter 3

System Requirements & Architecture

3.1 System Requirements

This section is dedicated to present the system requirements of the early phase of the project.

3.1.1 Actors

- **Office Factory Manager:** This is the central authority figure who will have overarching access to the functionalities of our portal. The Office Factory Manager can manage the clusters, pods, and services (Create, Delete, and Update). They also have the ability to check the resources usage of the clusters and pods, representing metrics like memory and CPU usage.
- **Factory Manager:** The Factory Manager for each factory (represented as a cluster) will have localized control over their specific cluster. They can monitor and manage their respective factory's clusters, pods, and services. They will have access to the same metrics as the Office Factory Manager, but only for their specific cluster.
- **Kubernetes:** Kubernetes is the foundational layer providing an API to create, destroy, and update pods/nodes/clusters. It plays a crucial role in facilitating the communication and control between the managers and the actual services.

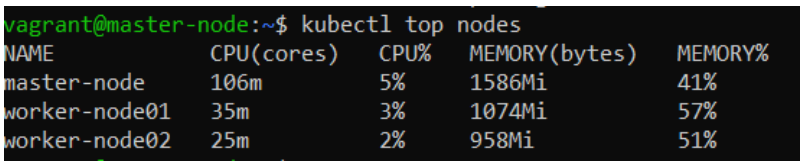

Figure 3.1: *Metrics related to the nodes available on the Kubernetes Cluster example. The command 'kubectl top nodes' is only available when the metrics YAML file is applied, thus installing the Metrics Server[1].*

- **Istio:** Istio, acting as the service mesh, enables the inspection of deployments, resource usage, metrics, and traffic policy definitions. It adds an extra layer of control and observability to the managers over their respective clusters.

- **Administrator:** The administrator maintains the highest level of control and access. They have full access to the back-end of the portal, all clusters, the Istio service mesh, and the SQLite database. They can configure all factories and have direct access to all back-end APIs.

3.1.2 Use Cases - Factory Management System 4.0

In our project, we simulate a scenario where a central office oversees a collection of factories distributed globally. The office president seeks real-time updates concerning each factory's activities, specifically the incoming and outgoing flow of materials.

Our Factory Management System 4.0 provides an accessible user interface (UI) through our portal. This UI enables the Office Factory Manager to conveniently manage clusters corresponding to the various factories. Actions such as creating, scaling up, or scaling down clusters can be easily accomplished through this UI, eliminating the need for complex command-line tools.

The Office Factory Manager, upon logging into the portal, gains access to comprehensive data about all factories. Alternatively, they can log into a specific factory profile to view details exclusive to that unit.

The Factory Manager is capable of monitoring resource usage, including CPU availability and memory consumption, assessing deployments, evaluating the health of pods, and generally overseeing the cluster state.

In the case of the Administrator, they have full access to each cluster, typically via tools such as the Kubernetes Dashboard (see Section **??**).

3.1.3 Functional Requirements

Autoscaling

Our system leverages the autoscaling feature of Kubernetes. This enables the Office Factory Manager, via the portal, to dynamically adjust the number of nodes, clusters and resources according to demand.

Life-cycle Management

Our system uses Kubernetes' own controller to manage the lifecycle of Pods, ensuring their resilience in the face of potential failures and facilitating updates, rollbacks and pauses.

Declarative Model

The system architecture is defined using YAML files. This declarative model ensures system consistency and robustness against unexpected behavior.

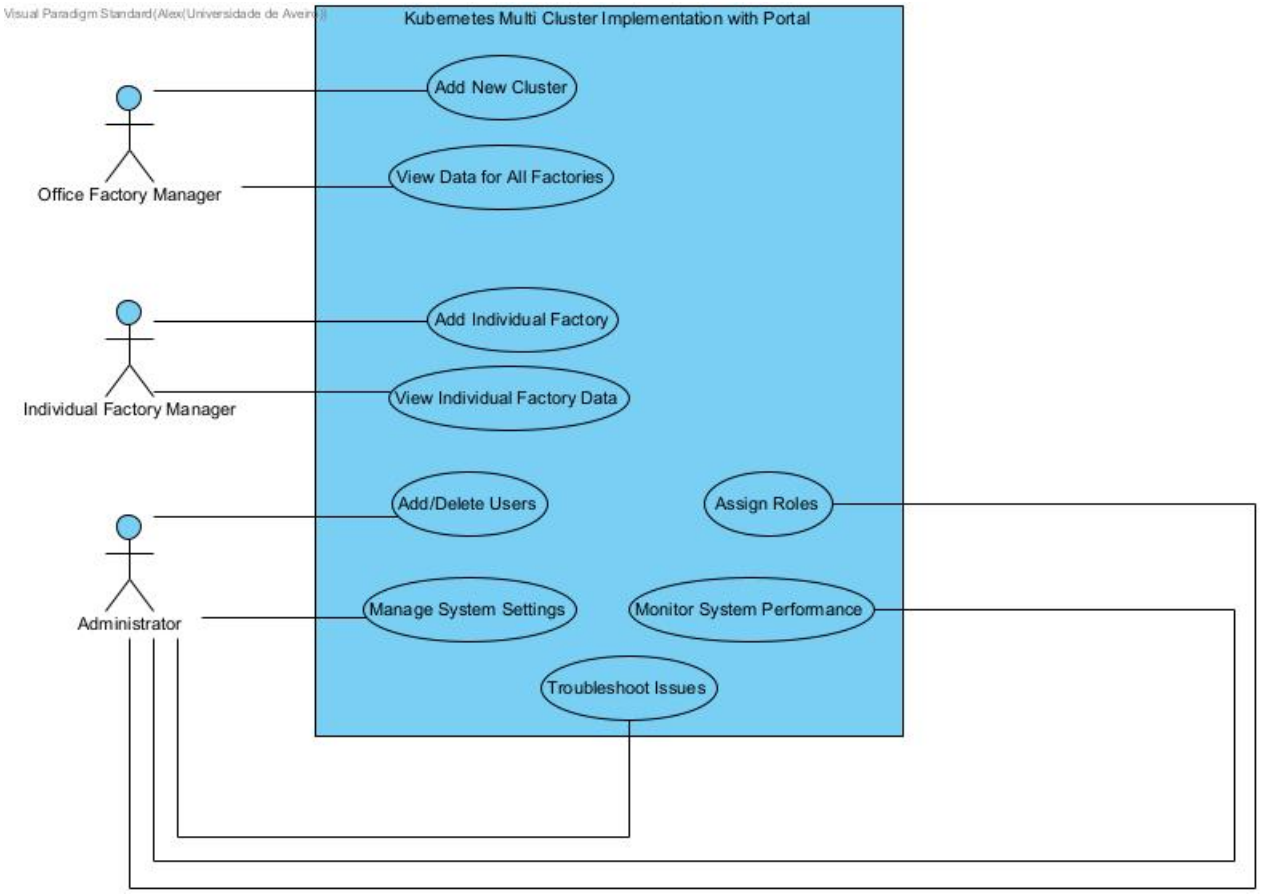

Figure 3.2: *Use Case scenario for the Portal*

Resilience and Self-Healing

Our system harnesses Kubernetes for self-healing functionalities. When a failure occurs, the system automatically redeploys the affected application or component to restore its intended state.

Persistent Storage

Storage and resources in our system are designed to be flexible, allowing dynamic expansion or contraction according to demand.

Load Balancing

The system utilizes Kubernetes' built-in load balancing feature, facilitating workload distribution across nodes and clusters.

Security

The system employs a service mesh to secure service-to-service communication, enforcing

traffic encryption through mutual TLS, authentication through certificate validation, and authorization through access policies.

Traffic management

Our system provides extensive traffic management capabilities, including dynamic service discovery, routing, traffic shadowing, and traffic splitting. It also supports deployment patterns like canary releases and A/B testing, as well as reliability features such as retries, timeouts, rate-limiting, and circuit breakers.

Observability

Our system's service mesh provides detailed observability features, offering insights into distributed tracing and generating comprehensive access logs.

Horizontal Scaling

The system supports horizontal scaling, enabling the dynamic addition or removal of containers based on processing utilization rules and criteria.

Secret and Configuration Management

Our system leverages Kubernetes for secret and configuration management, separating application secrets and configuration data from the image to streamline updates.

3.1.4 Non-Functional Requirements

- The system should provide an intuitive portal for user interaction, mitigating the need for interaction outside of the portal.
- The portal should display a comprehensive list of existing services and the clusters within the service mesh.
- The system must facilitate the selection and deployment of a service across specific clusters directly from the portal.
- The system should translate user-defined actions in the portal into appropriate API calls.
- The system should provide detailed information about containers, services, metrics, and resource usage to the user via the portal.

3.2 System Architecture

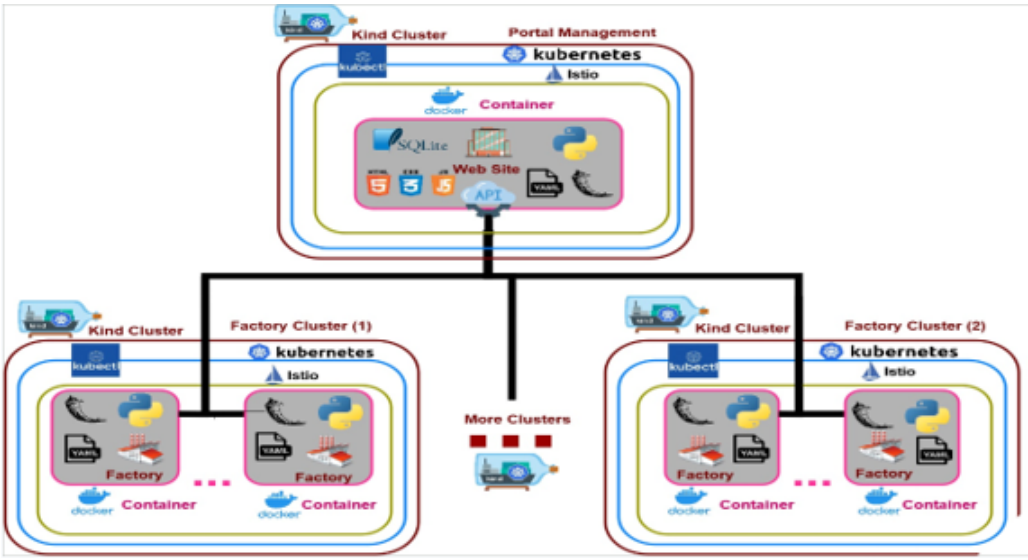

Figure 3.3: *Our Projects Architectural Design*

The architecture design above is our current multi-cluster architecture. It is composed of a main cluster and two remote clusters. All the cluster are running in an Host Virtual Machine using Ubuntu 22.04, the latest version, as the operating system. For each cluster,as mention above, is created a Docker Container 2.5.1 using Kind 2.5.1 and Kubectl to deploy and manage the clusters.

The Portal Clusters contains the services and deployments of the Portal Flask App and the Istio installation and definition. This cluster is the entry-point to visualize the data in the factories cluster.

The Factory clusters contain the services and deployments of the Factories Flask, and each Factory cluster will contain two distinct factories running. The Istio Control Plane on the Portal Cluster injects Proxies in all of the Factory Clusters, that allows and manages the communicating between Portal and Factories cluster, making it only possible from Factory to Cluster, exclusively. Given the scalability of Kubernetes, it is possible to create infinite clusters using a factory app as model, but ensure first that you have the resources available to create them.

More detailed information for the Portal the Factories application is available in the section Final Implementation 3.3 section.

3.2.1 Hardware and Software Required

The hardware and software requirements for our current portal project are outlined below.

Hardware Requirements

The Office Factory Manager's computer, which serves as the host for our application, should have a 64-bit operating system with at least 8GB of RAM. For testing and development environments, our portal application can be deployed on a local machine running Kind - Kubernetes in Docker, which mimics a multi-node Kubernetes cluster.

Software Requirements

The host machine should have Docker installed, as Kind - Kubernetes in Docker uses Docker containers to simulate Kubernetes nodes. Furthermore, to interact with the Kubernetes clusters, kubectl command-line tool is required.

Installations required within the nodes

All the necessary installations within the nodes are handled by the Kubernetes system, including the deployment and management of application containers. Here are the essential components used:

- **A Runtime Engine - Docker:** Docker is the default container runtime for Kubernetes and it's used by Kind - Kubernetes in Docker for creating Kubernetes nodes.
- **A Node Agent - Kubelet:** Kubelet is an agent that runs on each node in the cluster. It makes sure that containers are running in a pod.
- **Kubernetes Command-line tool - Kubectl:** Kubectl is a command-line tool that allows you to run commands against Kubernetes clusters. It is used to deploy applications, inspect and manage cluster resources, and view logs.
- **Kubernetes Cluster - kind:** Kind stands for Kubernetes in Docker, and it's a tool for running local Kubernetes clusters using Docker container "nodes". It's primarily used for testing Kubernetes itself, but may also be used for local development or CI.
- **Service Mesh - Istio:** As mentioned in Istio, this service mesh must be installed and functional for monitoring and controlling microservices in the Kubernetes cluster.
- **Network Security - Calico Network:** Calico is used as a network plugin for Kubernetes, providing a scalable networking solution and enforcing network policies to control and restrict the flow of traffic between pods.
- **Node Resource Usage Checker - Metrics Server:** Metrics server collects resource metrics from Kubelets and exposes them to the Kubernetes API. It is crucial for autoscaling applications based on resource usage and for providing resource usage data to the Kubernetes Dashboard.

- **Admin Interface for each Cluster - Kubernetes Dashboard:** Kubernetes Dashboard is a web-based user interface for managing and monitoring Kubernetes clusters. It provides a real-time overview of the cluster and detailed information about individual pods and nodes.

Note: Although this set up is ideal for testing and development, for production environments, a more robust hardware and software setup would be necessary. This could involve multiple physical or virtual servers, each with a 64-bit operating system, a minimum of 8GB of RAM, and a modern, multi-core CPU. Network connectivity between servers would also be necessary, along with additional software requirements such as a server version of Kubernetes.

3.2.2 Preliminary Experiments

Throughout the development of our project, we encountered several challenges and learned a lot from overcoming them. Early on, we faced issues such as obtaining the namespace from the master node, which was initially giving a connection refused error. This was resolved by adding 'hostNetwork: true' to the Metrics Server YAML file, enabling hostNetwork.

When we attempted to join worker nodes to the master node of a single cluster Kubernetes setup without a Service Mesh, we again faced connection refused errors. After investigating this issue, we discovered that the pod range IP was not wide enough.

Given the complexity of Kubernetes and its related technologies, we spent a significant amount of time learning about the platform and performing tests. We initially used Minikube to create a cluster and successfully deployed a sample application. However, we soon realized that Minikube was not sufficient for our needs due to compatibility issues with Istio.

After switching to Kubeadm, we successfully created a cluster with master and worker nodes and managed to connect a worker node from another host to the master. We also used Play with Kubernetes for testing purposes.

We initially attempted to optimize the creation of Virtual-Machines and software installation using Vermin, but it was not updated or maintained in a long time, leading to issues in implementation. As a result, we switched to using Vagrant, which proved to be a valuable asset for the project.

Our experiments were not limited to these tools and platforms. We also experimented with the Admiral multi-node setup for managing Docker containers. While this setup worked initially, we experienced some challenges:

- **Intermittent connectivity issues:** We encountered several connectivity issues between nodes that required frequent troubleshooting and adjustments to the network settings.
- **Challenges with scaling:** As the number of nodes increased, we faced issues with scaling the setup. The resources required by Admiral and the VMs began to exceed the capacity of our host machine.
- **Lack of community support and updates:** Admiral, similar to Vermin, is not actively maintained, which limited the availability of up-to-date documentation and community support.

Given these challenges, we decided to transition from Admiral and Vagrant to kind and Docker. Kind allowed us to simulate a multi-node Kubernetes cluster on a single machine, making it a better fit for our development and testing needs.

Overall, our preliminary experiments, though filled with challenges, proved crucial in understanding the complex landscape of Kubernetes and related technologies. The lessons learned during this phase shaped our final solution, resulting in a more resilient and scalable architecture.

3.3 Final Implementation

Here's our portal.

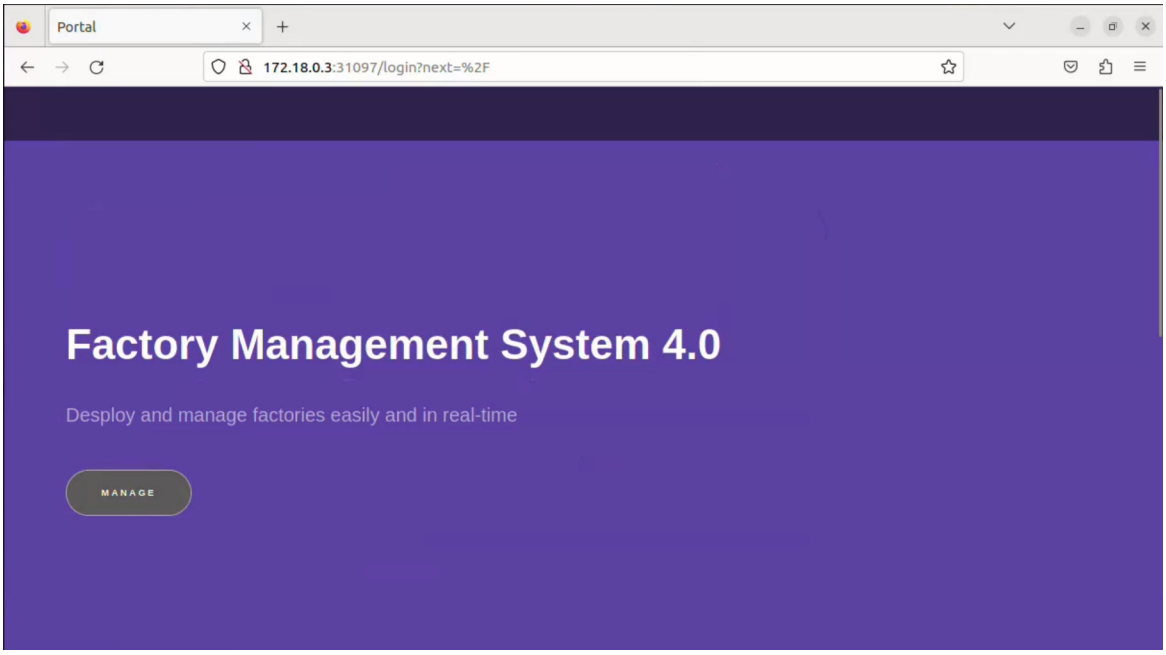

Figure 3.4: *Portal entry point*

3.3.1 Office Factory Manager

The Factory Manager is the one who has access to all the company's information (all the factories).

It can also create a new cluster of factories.

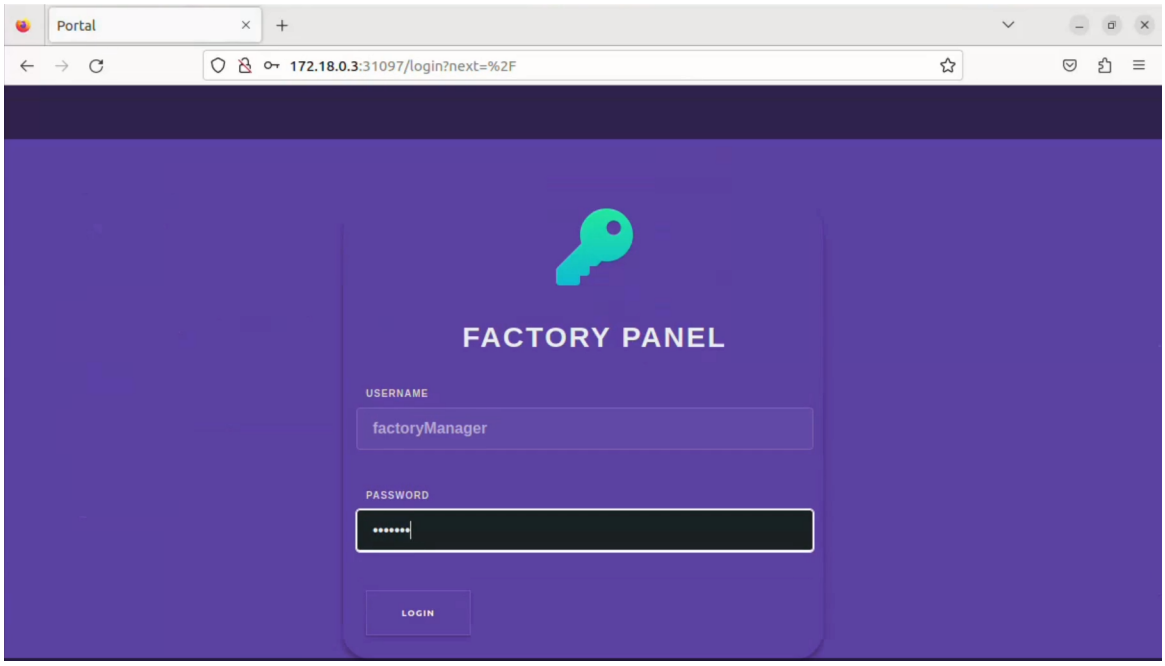

Figure 3.5: *Portal login as Office Factory Manager*

Logging in:

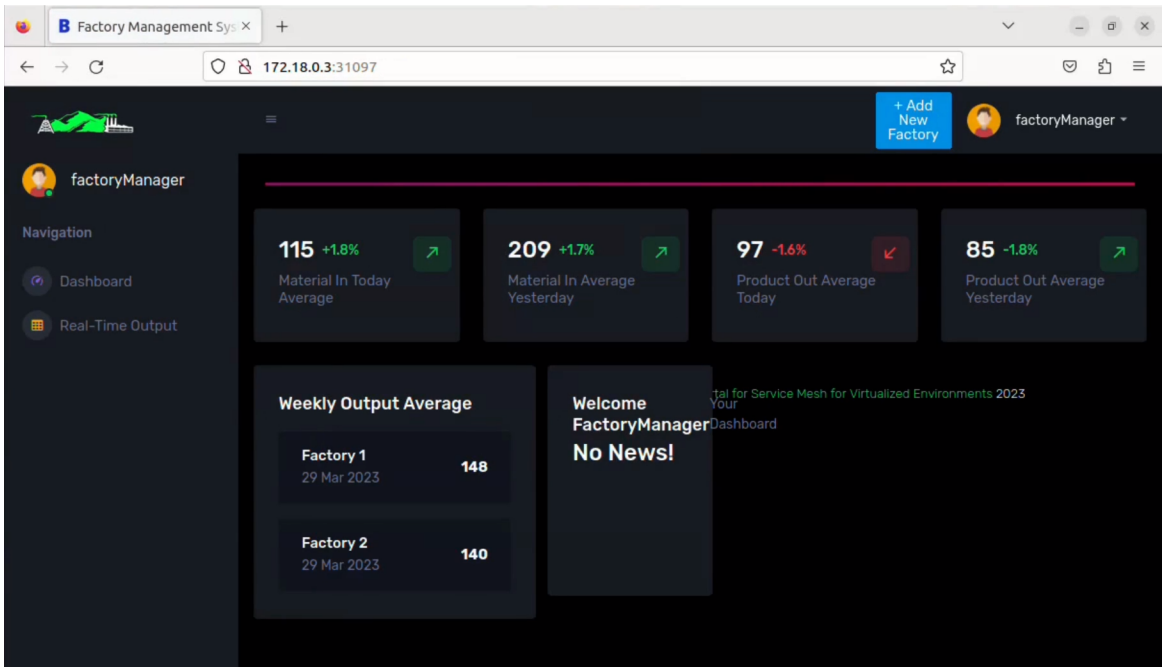

Figure 3.6: *Portal Main Page*

Here we have a summary dashboard that provides real time information about all the factories (average materials, either in or out). It's also possible to create a new factory.

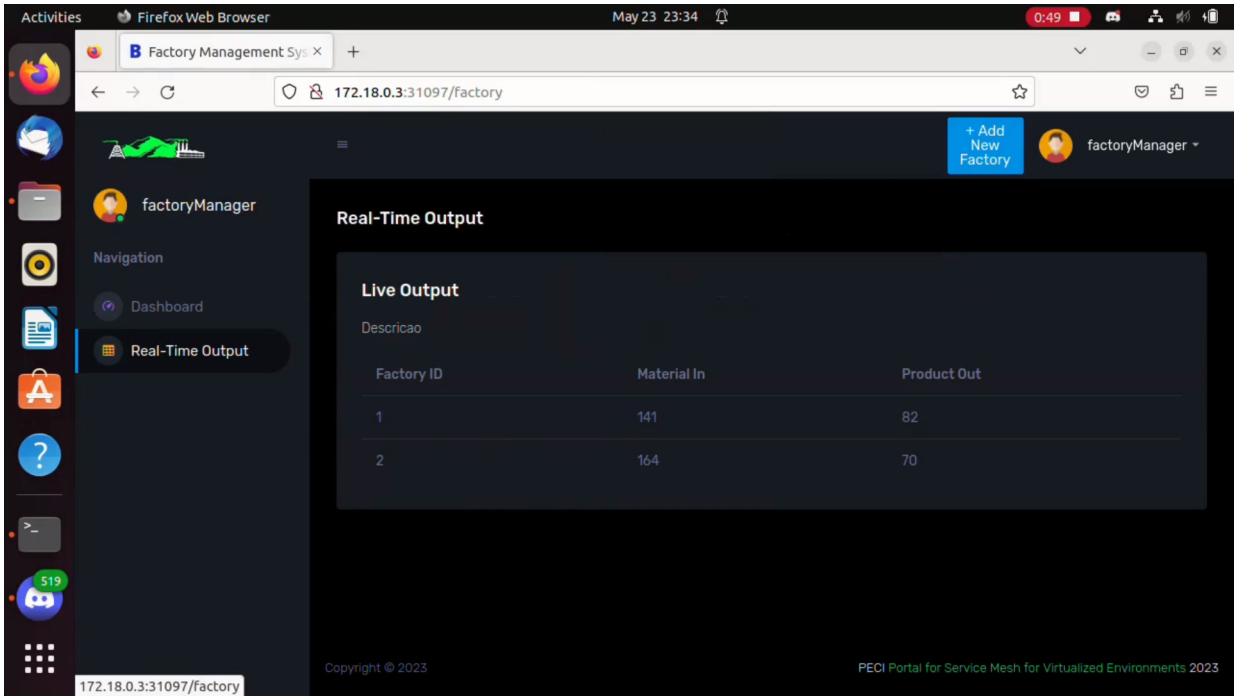

Figure 3.7: *Portal with detailed information for each factory*

To have a more precise view of each existing factory, there is a Real-Time Output section that provides real time information of each of all the factories.

3.3.2 Factory Manager

The factory manager is the manager of one factory. Logically, it only has access to the respectively factory.

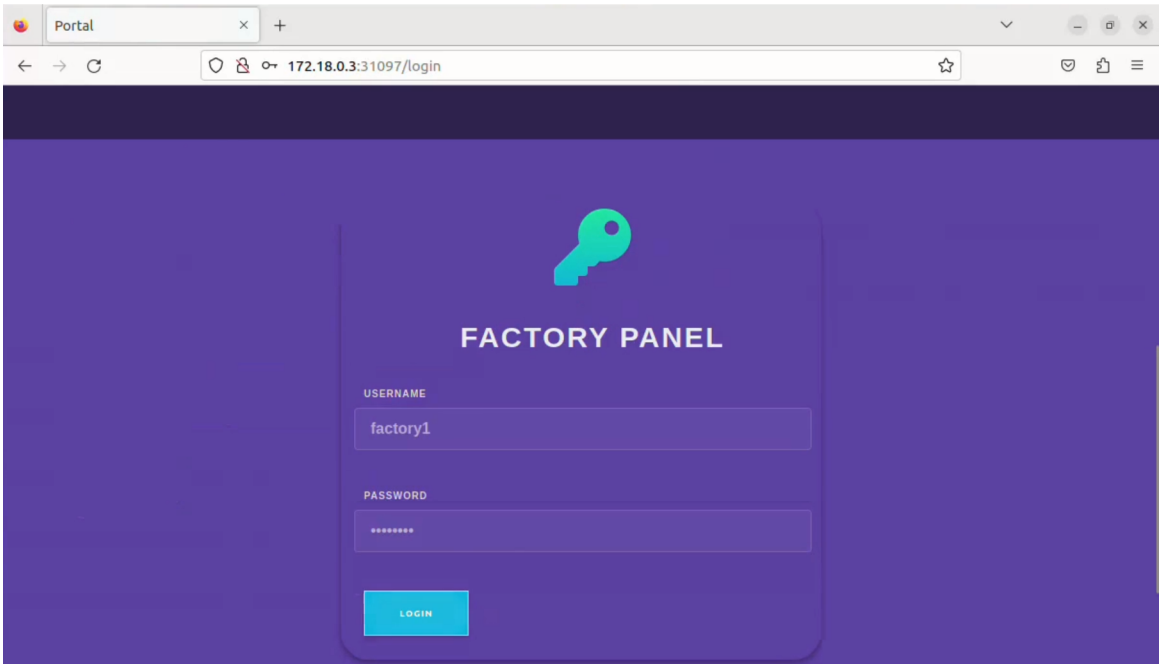

Figure 3.8: *Portal single Factory Manager Login*

Logging in:

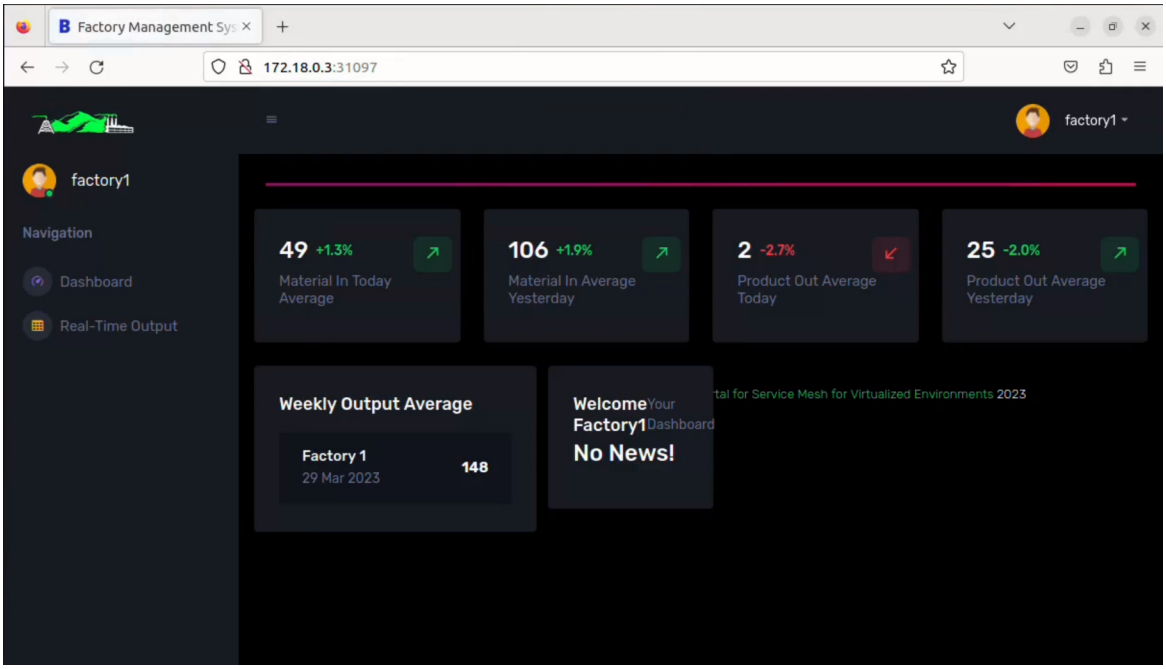

Figure 3.9: *Portal single Factory Main Page*

Similarly to the Office Factory Manager portal's dashboard, there's a dashboard with all the real time information of the respective factory.

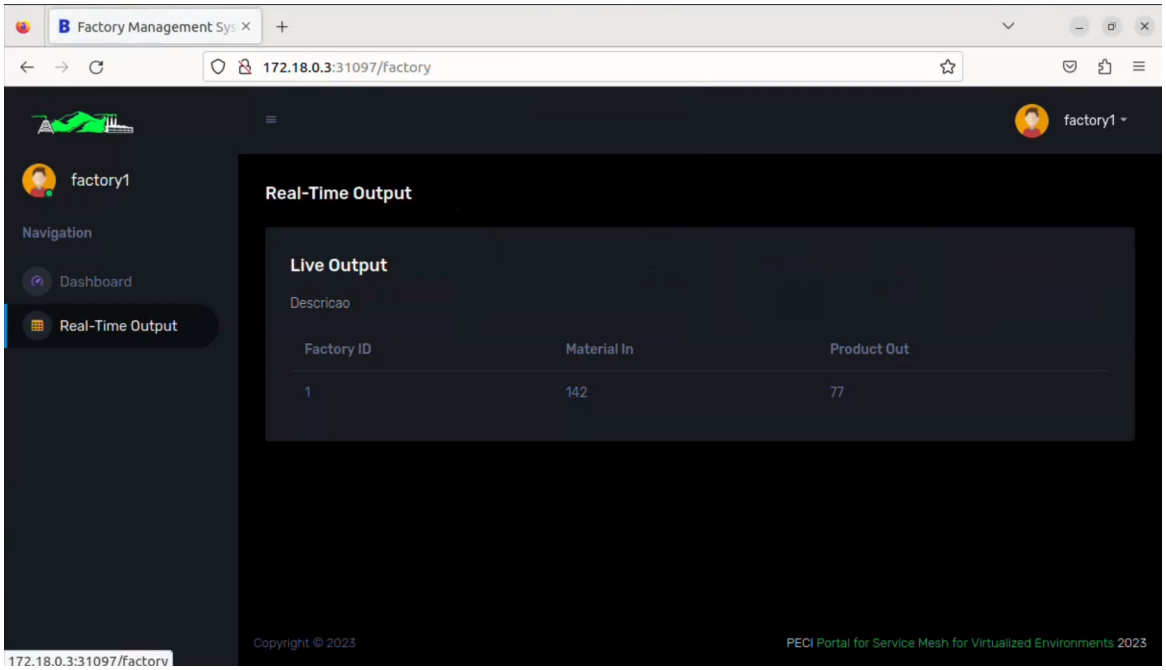

Figure 3.10: *Portal single Factory Manager detailed information*

There is also a Real-Time Output section that show the real time information of the factory.

3.3.3 Context

There is also a Real-Time Output section that show the real time information of the factory.

3.4 Future Work

While our portal solution has made significant strides, there is still room for refinement and expansion to make it a more comprehensive and professional platform. Future improvements aim to enhance both functionality and user experience, taking the portal from a foundational implementation to a more robust, real-world solution.

One of the main areas of improvement is enhancing the dynamic nature of our portal. Currently, the process to add a new factory, which in this context translates to a new cluster, is somewhat manual. It involves running a series of commands to create a local server on the host. We envisage a more streamlined solution, where this process could be initiated with a simple click of a button, thereby reducing the complexity and increasing the efficiency of adding new factories. Similarly, the ability to remove a factory with a simple action is another enhancement that we wish to implement.

At present, the user base is limited, consisting of the Factory Manager and two Factory accounts. Expanding the user management feature is an important area of future work. A registration page allowing the creation of additional users would enhance the accessibility and inclusivity of our platform. This would also involve expanding the SQLite database to store and manage the details of these additional users.

Another significant area for enhancement is the dashboard functionality. Currently, it serves as a basic informational display. We plan to integrate advanced features such as data filtering options, real-time alerts or notifications triggered by predefined conditions, and graphical representation of data. These enhancements will not only make the dashboard more informative but also intuitive and visually appealing.

Last, but by no means least, the front-end design of the portal needs an upgrade. Our objective is to create an interface that is not only aesthetically pleasing but also easy to navigate, especially for users who are not familiar with Kubernetes technology.

In summary, our future work aims to transform our portal into a more dynamic, user-friendly, and sophisticated platform, making it a more viable and practical solution for real-world applications.

Bibliography

[1] Metrics server. https://github.com/kubernetes-sigs/metrics-server. [accessed:14.12.2022].

[2] Service mesh concept explained in plain english. https://medium.com/swlh/service-mesh-explained-in-plain-english-8e5505f74ead, May 2019. [accessed:12.12.2022].

[3] Konghq. https://konghq.com/. [accessed: 01.12.2022].

[4] Readhat openshift. http://www.redhat.com/en/technologies/cloud-computing/openshift/. [accessed: 01.12.2022].

[5] Amazon web services app mesh. http://aws.amazon.com/pt/app-mesh/. [accessed: 02.12.2022].

[6] http://www.ibm.com/pt-en. [accessed: 29.11.2022].

[7] Spotify. https://spotify.com. [accessed:14.12.2022].

[8] Helios website. https://github.com/spotify/helios. [accessed:14.12.2022].

[9] grpc - a high performance, open source universal rpc framework. https://grpc.io/. [accessed:14.12.2022].

[10] Envoy proxy. https://www.envoyproxy.io/. [accessed:14.12.2022].

[11] Autotrader uk. https://www.autotrader.co.uk/. [accessed:14.12.2022].

[12] What is mtls? | mutual tls. https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/. [accessed:14.12.2022].

[13] Googles kubernetes engine. https://cloud.google.com/kubernetes-engine. [accessed:14.12.2022].

[14] Kubernetes. https://kubernetes.io/. [accessed:14.12.2022].

[15] Kubernetes overview. https://medium.com/analytics-vidhya/kubernetes-overview-5f86e06750c4. [accessed:15.12.2022].

[16] kube-proxy. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. [accessed:14.12.2022].

[17] Gedalyah Reback. Istio vs. linkerd vs. consul: A comparison of service meshes. https://logz.io/blog/istio-linkerd-consul-comparison-service-meshes/. [accessed: 25.11.2022].

[18] Python. https://www.python.org/. [accessed:14.12.2022].

[19] Flask. https://flask.palletsprojects.com/. [accessed:14.12.2022].

[20] Bash. https://www.gnu.org/software/bash/. [accessed:14.12.2022].

[21] What is yaml? https://www.redhat.com/en/topics/automation/what-is-yaml. [accessed:14.12.2022].

[22] Vagrant website. https://www.vagrantup.com//. [accessed:01.12.2022].

[23] Virtual box. https://www.virtualbox.org/. [accessed:14.12.2022].

[24] Vijay Iyer Jason Webb, Anil Attuluri. Genius of admiral. https://medium.com/intuit-engineering/genius-of-admiral-3307e63e3ab6, December 2019. [accessed: 29.11.2022].