POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Liquid Computing on Multiclustered Hybrid Environment for Data Protection and Compliance

Supervisor
Prof. Fulvio Risso
Company tutor
Ing. Antonino Sirchia

Candidate
Alessandro Lucani

Academic Year 2022-2023

Summary

With business logic running at the Edge, much larger volumes of data can be secured and processed locally where the data is produced. This reduces the network bandwidth requirements and consumption between Edge and Cloud, increases responsiveness, decreases costs, and protects customers' data privacy.

In particular the data privacy is nowadays becoming a topic more and more relevant, in fact more and more nations are developing and adopting legislation regarding data privacy and protection and so the companies that leverages on cloud services to manage data need to change their processes in order to be compliant to the new regulations.

The objective of this thesis is to propose a solution about the data storage and personal data processing within the scope of the Digital Ecosistem Platforms. This technology is becoming everyday more popular but it has to face the limitation imposed by the regulations in order to exploit its potentiality. In fact it is based on a cloud-native design that provides extensibility through its ability to blend edge devices with more powerful central clusters, creating hybrid cloud scenarios.

This thesis aims to analyze some solutions that don't include any further technological step but only architectural and organizational ones and then, after making some evaluations on these, propose a possible new approach and real-world application to solve the problem. The proposed idea leverages on the liquid computing concept which points to the creation of a resource continuum, by connecting together several clusters and hiding everything behind standard cloud computing concepts. This approach has been deeply studied and the solution that is the result of this thesis make use of it to create the architecture that allows to store and process data according to data protection regulations.

This thesis, developed with Engineering Ingegneria Informatica[5], has been tested in-house and will now be incorporated into the company's solutions offered to costumers. Moreover this solution has been presented at one of the meetings of the European initiative Gaia-X[6] at Paris as one of the assets of the lighthouse project Structura-X[23].

Contents

List of Tables

List of Figures

Part I

Introduction

General introduction

The increasing diffusion of cloud computing lead companies to move to containerized applications and to adopt the cloud native approach in creating solutions.

On the other hand one of the assets that is gaining every day more value are data, in particular personal data, which can be collected in many different ways: by filling a form, by using a device or just by being in a specific place.

These two flows lead to the creation and diffusion of the Digital platforms, huge and widespread systems which are usually based upon container orchestrator in order to be more flexible and easy to be deployed and managed. They are composed of various parts which cooperate in order to give a service to the client that has requested it. These ecosystems can be exploited for various reasons but among their objectives the main ones are to collect and process data from users around the world to create a data workspace.

The adoption of this solution needs to face several issues such as network ones, response speed, availability and others but the main one is the compliance to the worldwide data protection regulations. In fact the data privacy legislation around the world set limits about where data should be stored and how they should be transferred.

Goal of the thesis

The objective of this thesis is to find and analyze the state of the art of the current technologies in order to propose a solution for the issue described above.

In particular the focus will be on containers, orchestrators, data regulations around the world to understand which are the limits to which the solution must be compliant, and then on the Digital platforms in order to find a real application of the proposed solution. In the following paragraphs will be paid attention to several possible theoretical approaches and actual implementations and then, after a careful analysis, a solution will be proposed and the choices made will be motivated.

Outline

- **Chapter 2:** introduction to containers and cloud native approach (going through some implementations and distributions) and presentation of Digital platforms with some examples.
- **Chapter 3:** presentation of data protection regulations going from the theoretical aspects to the description of some of the most famous legislation.

- **Chapter 4:** presentation of the state of art of the possible solutions to the highlighted problem with the description of the proposed new approach.
- **Chapter 5:** deep analysis of the issue and the proposed solution in all its aspects including the actual implementation and the results.
- **Chapter 6:** conclusion and presentation of possible future developments.

Part II

Container orchestration and digital platforms

Chapter 1

Introduction to containers

During the years the process to deploy an application has changed a lot and this section will briefly summarize the main steps that are depicted in the following figure.

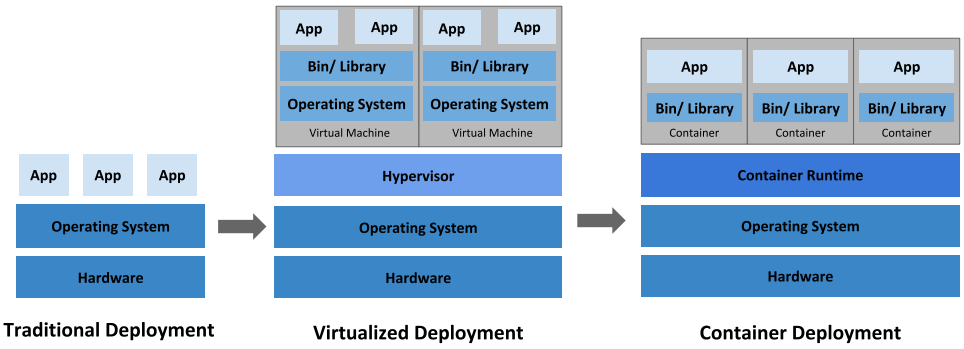

Figure 1.1: Evolution of deployment technology

Long back organizations ran applications on physical servers and each particular server was used in the deployment of only one particular application because there was no way to define resource boundaries for applications, and this caused resource allocation issues. Running only **one application per server** did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers. In this scenario if the organizations wants to run several applications on the same machine the problem of unequal resource division arises. So some applications would take up most of the resources, and as a result, the other applications would underperform.

The problem described above was affecting the entire IT industry so virtualization was introduces as a solution. Virtualization is the process of running a virtual instance of a computer system in a layer abstracted from the actual hardware. This innovation allows to run multiple **Virtual Machines** (VMs) on a single physical server's CPU exploiting the concept of isolation between VMs achieved through an hypervisor that is a software responsible for creating and running virtual machines, it allows one host computer to support multiple guest VMs by virtually sharing its resources, such as memory and processing. In this new scenario the utilization of resources and the unequal resource division was solved so a better scalability can be achieved because an application can be added or updated easily. This innovation solves the problems highlighted before but it brings new issues in fact it is needed to install OS in each of these systems, they use a lot of RAM and the same applies to other physical resources like CPU and hard disk and so this is not suitable for scenarios where the startup time is an important feature.

So, to overcome the problems of the VMs, a new technology was developed: **containers**. Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications and this new step in the virtualization process is called lightweight virtualization. They also allows a better resource utilization and isolation that brings to a scenario with predictable application performances. Containers offers a lot of features that made them so popular. Among them must be highlighted that they allows agile application creation and continuous integration and deployment due to the layered filesystem, improved the DevOps by the Dev and Ops separation of concerns (create application container images at build/release time rather than deployment time) and environmental consistency across development, testing, and production (runs the same on a laptop as it does in the cloud) and they enables a greater portability.

The endless growing of popularity of containers and their increasingly widespread use led to the problem of maintaining and controlling them.

Containers effectively guarantee that applications run the same way anywhere, allowing to scale applications up and down but it is needed some way to help automate the maintenance of those applications, enable the replacement of failed containers automatically, and manage the rollout of updates and reconfigurations of those containers during their lifecycle.

Tools to manage, scale, and maintain containerized applications are called **orchestrators**, and the most common examples of these are Kubernetes, with all its distributions, and Docker Swarm that will be further and deeper evaluated in the next chapter.

Chapter 2

Container Orchestrators

This chapter will focus on presenting the most common orchestrators and in particular, Kubernetes, with some of its main features.

2.1 DockerSwarm

Docker Swarm is a part of the native Docker engine that allows to scale containerized applications inside clusters, arbitrary number of machines logically grouped. It enables central cluster management as well as the orchestration of containers. The Docker Swarm architecture is based on a master Node and a variable number of worker nodes. The master handles the management of the cluster and scheduling tasks, while the swarm workers perform the execution. The main concept that describe the way containers are orchestrated is the **Service**. The Service is the abstraction of how tasks are carried on by the cluster. Each service is composed by a set of individual tasks that are processed in a single container deployed in one or more nodes of the cluster. When creating a service, must be specified which container image it's based on and which commands are run in the container.

Another tool present in the Docker engine is **Docker Compose** that allows to define multi-container applications - or "stacks" - and run them in a Docker node or cluster. Stacks in Docker are defined as groups of interconnected services that share software dependencies and are orchestrated and scaled together.

In the end the correct usage of these two tools lead to a valid orchestration architecture that can be suitable in various scenarios and is ideal for small workloads. Among its advantages there are the simple installation and familiarity and emphasis on ease-of-use.

2.2 Kubernetes

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community. The following section is a brief introduction to Kubernetes that takes inspiration from the official documentation.[15]

2.2.1 Architecture

The first element that must be introduced is the **cluster** that is the first concept that a user get in touch with after deploying Kubernetes. A Kubernetes cluster consists of a set of worker machines, called **nodes**, that run containerized applications. Every cluster has at least one node that acts as the **master** and hosts the control plane. The control plane manages the worker nodes and the pods in the cluster. In production environments, the control plane typically runs on multiple machines, and a cluster typically comprises multiple nodes, ensuring fault-tolerance and high availability. The worker node(s) host the pods that are the components of the application workload.

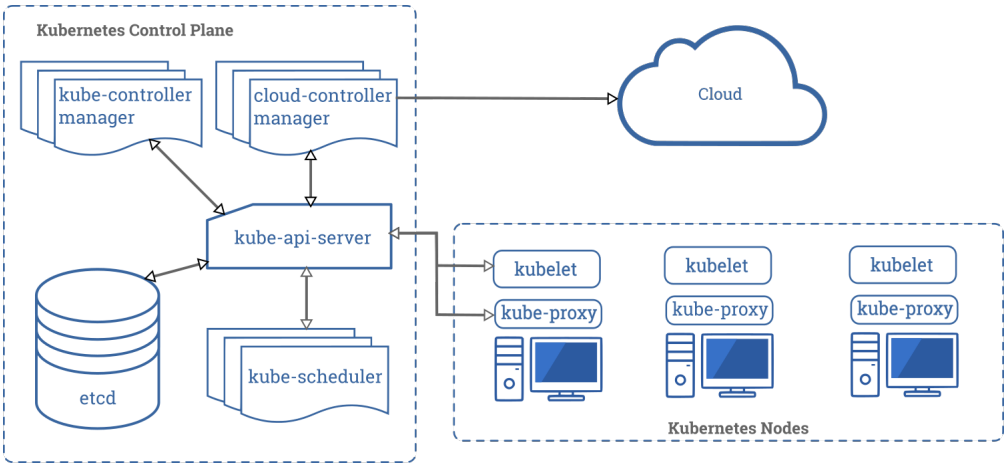

Figure 2.1: Kubernetes cluster components

Control Plane Components

The control plane's components make decisions that affect the entire cluster, such as scheduling, as well as detecting and responding to events within the cluster, such as launching a new pod when there are not enough replicas for a deployment. Control plane components can be run on any machine in the cluster.

API server

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the frontend for the Kubernetes control plane.

The primary implementation of the Kubernetes API server is kube-apiserver. It is built to scale horizontally by adding more instances and in some cases, it's recommended to run multiple instances and distribute the traffic among them.

etcd

The etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

Scheduler

The scheduler is the control plane component that monitors for newly created pods without a designated node, and selects a node for them to run on.

The scheduler takes into account various factors such as resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, workload interference, and deadlines, when making scheduling decisions.

The default scheduler for Kubernetes is kube-scheduler, which runs as part of the control plane. It is also designed to be replaced by a custom scheduling component if desired.

kube-controller-manager

kube-controller-manager is the component of the control plane that manages the controller processes and compares the current state of a resource with its desired state as defined in the resource's definition. It consists of multiple separate controllers and processes, but for simplicity, they are all combined into a single binary and run as a single process.
Some types of these controllers are:

- **Node controller**: responsible for noticing and responding when nodes go down.
- **Replication controller**: ensures that a specified number of pod replicas are running at any one time.
- **Service Account & Token controllers**: create default accounts and API access tokens for new namespaces.

cloud-controller-manager

cloud-controller-manager is a component of the Kubernetes control plane that includes cloud-specific control logic. It only runs controllers that are designed for for each specific cloud provider.

Like the kube-controller-manager, the cloud-controller-manager combines multiple independent control loops into a single binary that is run as a single process and can be scaled horizontally.
Some controllers that can have cloud provider dependencies:

- **Node controller**: responsible for updating Node objects when new servers are created in your cloud infrastructure. The node controller obtains information about the hosts running inside your tenancy with the cloud provider.
- **Route controller**: responsible for setting up routes in the underlying cloud infrastructure.
- **Service controller**: creates, updates and deletes cloud provider load balancers.

Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet

The kubelet is a agent that runs on each node in the cluster, and it responsible for ensuring that the containers in a pod are running properly.

It follows a set of specifications that are provided through different means, and ensures that the containers described in those are running and healthy.

kube-proxy

kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

It manages network rules on the nodes, which allow for network communication to the pods from sessions inside or outside the cluster.

Container runtime

The container runtime is the software that is responsible for running containers.

Kubernetes supports container runtimes such as containerd, CRI-O, Docker Engine, Mirantis Container Runtime (formerly known as Docker Enterprise Edition) and any other implementation of the Kubernetes CRI (Container Runtime Interface).

2.2.2 Objects vocabulary

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

- which containerized applications are running (and on which nodes)
- the resources available to those applications
- the policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

When an object is created, the user is effectively communicating to the Kubernetes system what the desired state of the cluster's workload should look like. The Kubernetes controllers will continuously work to ensure that the object exists.

To interact with Kubernetes objects, the Kubernetes APIs must be used, which can be accessed through the API server that is present in the control plan

When creating an object in Kubernetes, the object spec that describes its desired state must be provided, as well as some basic information about the object (such as a name). Most often, the information is provided to kubectl, the Kubernetes command-line tool, in a *.yaml* file and then kubectl converts the information to JSON when making the API request.

Here there is an example of the *.yaml* file used to create a pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
```

Namespaces

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. The names of resources must be unique within a namespace, but not across different namespaces. Namespaces are used to divide cluster resources among multiple users by means of resource quota, a tool that sets limits on resource consumption.
Kubernetes starts with four initial namespaces:

- **default**: the default namespace for objects with no other namespace.
- **kube-system**: the namespace for objects created by the Kubernetes system.
- **kube-public**: this namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster.
- **kube-node-lease**: this namespace is used for Lease objects associated with each node that allow the kubelet to send heartbeats so that the control plane can detect node failure.

Labels and Selectors

Labels are key/value pairs that are attached to objects, such as pods. They are intended to be used to identify objects in a way that is meaningful to users, but does not carry any semantics for the core system. Labels can be added to objects at creation time and can be modified at any point later on.

Unlike names and UIDs, labels do not provide uniqueness. It is common for multiple objects to carry the same label(s). A label selector can be used to identify a group of objects that share a common characteristic.

Finalizers

Finalizers are namespaced keys that instruct Kubernetes to wait for specific conditions to be met before fully deleting resources that are marked for deletion.

When an object with finalizers is requested to be deleted, the Kubernetes API marks the object by populating the *.metadata.deletionTimestamp* field and returns a 202 status code (HTTP "Accepted"). The object remains in a terminating state while the control plane performs the actions defined by the finalizers. Once these actions are completed, the controller removes the relevant finalizers from the object. When the *.metadata.finalizers* field is empty, Kubernetes considers the deletion complete and deletes the object.

Finalizers can be used to control the garbage collection of resources. For example, a finalizer can be defined to clean up related resources or infrastructure before the controller deletes the target resource.

Pod

Pods are the building blocks of a Kubernetes deployment. They are the smallest deployable units that can be created and managed in a Kubernetes cluster. Pods consist of one or more containers that are tightly coupled and share storage and network resources. They are always co-located and co-scheduled, and run in a shared context (Figure 2.2).

Pods are typically not created directly, but rather through other resources such as **Deployments** or **StatefulSets**. These resources provide additional functionality such as replication, rollout, and automatic healing in case of pod failure. For example, if a node fails, a controller will notice that the pods on that node have stopped working and create a replacement pod, which the scheduler will place on a healthy node.

Pods are designed to share resources, communicate with each other, and coordinate their termination.

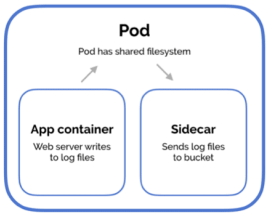

Figure 2.2: Multi-container pod example

ReplicaSet

ReplicaSets are responsible for maintaining a stable number of running pods at all times. They are commonly used to ensure a specified number of identical pods are running for high availability. However, they are often not used directly, but rather through a Deployment. A Deployment is a higher-level concept that manages ReplicaSets, providing declarative updates to pods and additional features.

StatefulSet

The StatefulSet resource is designed to manage stateful applications in Kubernetes.

It is responsible for deploying and scaling a set of pods, and ensuring the ordering and uniqueness of these pods.

When persistence is required for your workload, using a StatefulSet can be an effective solution. The persistent pod identifiers make it possible to match existing storage volumes to new pods, even in the event of pod failure.

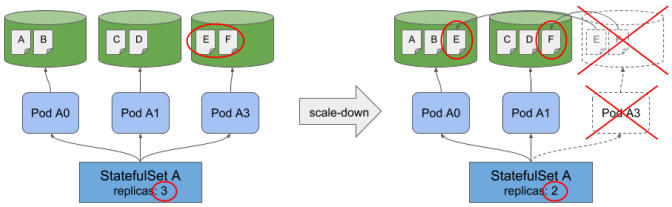

Figure 2.3: Example of scaling down with StatefulSet

Deployment

A Deployment is a resource object in Kubernetes that provides declarative updates to applications. A deployment allows to describe an application's life cycle, such as which images to use for the app, the number of pods there should be, and the way in which they should be updated.

A Deployment is a high-level concept in Kubernetes that manages ReplicaSets and Pods, providing declarative updates and rollouts for your application. It enables you to define the desired state of your application, including the number of replicas, the images to use, and the update strategy. This allows for easy scaling, rolling updates, and automatic recovery in case of failures.

Here there is an example of a deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

In this example a Deployment named **nginx-deployment** is created, indicated by the *.metadata.name* field. The Deployment creates three replicated Pods, indicated by the *.spec.replicas* field. The *.spec.selector* field defines how the Deployment finds which pods to manage. In this case, the selected label is *app: nginx* and it is defined in the *.spec.template* section.
The *.spec.selector* field contains the following sub-fields:

- The pods are labeled *app: nginx* using the *.metadata.labels* field.
- The pod template's specification, *.template.spec* field, indicates that the pods run one container which runs the nginx Docker Hub image at version 1.14.2.
- Create one container and name it nginx using the *.spec.template.spec.containers[0] .name* field.

- The container exposes the port 80 to be contacted using the *.spec.template.spec .containers[0].ports[0].containerPort* field.

Service

A Service is an abstract way to expose an application running on a set of pods as a network service. The application does not require modification for using an unfamiliar service discovery mechanism. Pods are given unique IP addresses and a shared DNS name, allowing for load balancing.
There are 4 types of service that are used in different scenarios:

- **ClusterIP**: exposes a service which is only accessible from within the cluster. This is the deafult value for the Service type.
- **NodePort**: exposes a service via a static port on each node's IP.
  The NodePort Service can be accessed from outside the cluster at **<NodeIP>:<NodePort>**.
- **LoadBalancer**: exposes the service via the cloud provider's load balancer.
- **ExternalName**: maps a service to a predefined externalName field by returning a value for the CNAME record.

The following is an example of a pod and a Service related to it:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app.kubernetes.io/name: proxy
spec:
  containers:
  - name: nginx
    image: nginx:stable
    ports:
      - containerPort: 80
        name: http-web-svc
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
```

```
19 spec:
20   selector:
21     app.kubernetes.io/name: proxy
22   ports:
23   - name: name-of-service-port
24     protocol: TCP
25     port: 80
26     targetPort: http-web-svc
```

This specification creates a pod, as explained above, and a new Service object named "nginx-service", which targets TCP port named *http-web-svc* on any pod with the *app.kubernetes.io/name=proxy* label and exposes it through port 80. In this case the type is not specified so it is a ClusterIP. It can be selected using the *.spec.type* field.

There is also another solution that is the use of an Ingress to expose your Service. Ingress is not a Service type, but it acts as the entry point for your cluster. It allows to expose multiple services under the same IP address.

Ingress Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource. An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An Ingress specifies rules that map incoming requests to specific services based on the request's host or path. This allows for easy access to multiple services within a cluster using a single IP address or domain name. It needs an Ingress controller to work. The Ingress controller, on the other hand, is a piece of software that implements the ingress rules and actually routes the traffic. It watches the Kubernetes API for new ingress resources and configures a load balancer or reverse proxy to enforce the rules. There are different types of ingress controllers available, such as Nginx, HAProxy, Istio, etc. Must be highlighted that an Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.

EndpointSlices EndpointSlices provide a simple way to track network endpoints within a Kubernetes cluster. An EndpointSlice contains references to a set of network endpoints. The control plane automatically creates EndpointSlices for any Kubernetes Service that has a selector specified. These EndpointSlices include references to all the Pods that match the Service selector. EndpointSlices group network endpoints together by unique combinations of protocol, port number, and Service name.

2.2.3 Extension possibilities

Kubernetes is highly configurable and extensible and that's precisely why there is rarely the need to fork or submit patches to the Kubernetes project code.

Kubernetes is designed to be automated by writing client programs. Any program that reads and/or writes to the Kubernetes API can provide useful automation. There is a specific pattern for writing client programs that work well with Kubernetes called Controller pattern. Controllers typically read an object's *.spec*, possibly do things, and then update the object's *.status*.

In particular in this section the focus will be on the **Operator Pattern** and the **Custom Resources** because they will be used in the next chapters.

2.2.3.1 Operator Pattern

Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop.

Kubernetes is designed for automation and the operator pattern lets you extend the cluster's behaviour without modifying the code of Kubernetes itself by linking controllers to one or more custom resources. Operators are clients of the Kubernetes API that act as controllers for a Custom Resource.

2.2.3.2 Custom Resources

Custom resources are extensions of the Kubernetes API. A resource is an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind; for example, the built-in pods resource contains a collection of pod objects.

A custom resource is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes installation. However, many core Kubernetes functions are now built using custom resources, making Kubernetes more modular.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster itself. Once a custom resource is installed, users can create and access its objects just as they do for built-in resources like pods.

The Kubernetes declarative API enforces a separation of responsibilities.The desired state of the resource can be declared. The Kubernetes controller keeps the current state of Kubernetes objects in sync with your declared desired state. The Operator pattern combines custom resources and custom controllers.

2.2.4 Distributions

When setting up a Kubernetes environment, there are two possibilities: vanilla Kubernetes and managed Kubernetes. With vanilla Kubernetes, a software development team has to pull the Kubernetes source code binaries, follow the code path, and build the environment on the machine. The other option, managed Kubernetes, presents some pre-installed and pre-configured tools that improve some features such as storage, security, deployment, monitoring, etc. Managed Kubernetes versions are also known as Kubernetes distributions.

2.2.4.1 Openshift

Red Hat OpenShift[19] is a leading enterprise Kubernetes platform that enables a cloud-like experience everywhere it's deployed. Whether it's in the cloud, on-premise or at the edge, Red Hat OpenShift gives the ability to choose where to build, deploy, and run applications through a consistent experience.

The foundation of OpenShift Container Platform is based on Kubernetes and therefore shares the same technology.

Both **Kubernetes and OpenShift** can deploy and run on public cloud and local environments to enable a better end user experience. But there are some differences that must be taken into consideration for better explointing the resources. This section summarize the main differences.[20]

- **Deployment**: Kubernetes offers more flexibility as an open-source framework and can be installed on almost any platform — like Microsoft Azure and AWS — as well as any Linux distribution, including Ubuntu and Debian. OpenShift, on the other hand, requires Red Hat's proprietary Red Hat Enterprise Linux Atomic Host (RHELAH), Fedora, or CentOS.
- **Security**: OpenShift has stricter security policies. For instance, it is forbidden to run a container as root. It also offers a secure-by-default option to enhance security. Kubernetes doesn't come with built-in authentication or authorization capabilities, so developers must create bearer tokens and other authentication procedures manually.
- **Support**: Kubernetes has a large active community of developers who continuously collaborate on refining the platform. It also offers support for multiple frameworks and languages. OpenShift has a much smaller support community that is limited primarily to Red Hat developers.
- **Networking**: Kubernetes lacks a networking solution but lets users employ third-party network plug-ins. OpenShift, on the other hand, has its out-of-the-box networking solution called Open vSwitch, which comes with three

native plug-ins.

- **Templates**: Kubernetes offers Helm templates that are easy to use and provide a generous amount of flexibility. OpenShift templates are nowhere near as flexible or user-friendly.

- **Image Registry Management**: Kubernetes doesn't have an integrated image registry, although it allows you to pull images from a private registry so you can create your own pods. Additionally, you can make your own Docker registry.

  OpenShift, on the other hand, has an in-built image registry and pairs seamlessly with DockerHub or Red Hat. Therefore, developers can use image streams to effortlessly search for and manage container images.

- **User Experience and Interface**: the Kubernetes interface, which can be complex, may confuse beginners. Users who want to access the Kubernetes web graphics user interface (GUI) must install the Kubernetes dashboard and use kube-proxy to send their machine's port to the cluster server. Users also must create bearer tokens to make authentication and authorization easier, since the dashboard doesn't have a login page.

  OpenShift, conversely, features an intuitive web console which includes a one-touch login page. The console offers a simple, form-based interface, allowing users to add, delete, and modify resources. OpenShift has the distinct user advantage

2.2.4.2 Rancher

Rancher[22] is a complete software stack for teams adopting containers. It addresses the operational and security challenges of managing multiple Kubernetes clusters, while providing integrated tools for running containerized workloads, and also unites them with centralized authentication, access control and observability.

The main scenario of applicability of Rancher is the management of several Kubernetes cluster, maybe with different distributions from different vendors.

2.2.4.3 Lightweight distributions

Kubernetes is the most widely orchestrator but there are some use cases in which it is not needed in all its features or the hardware characteristics of the testbed are not enough to execute correctly the full Kubernetes. For this situations some lightweight versions of Kubernetes were developed and in this section the focus will be on three of them: K3S, MicroK8s and Kubernetes Kind

K3s[12] is a highly available, certified Kubernetes distribution designed for production workloads in unattended, resource-constrained, remote locations or inside IoT appliances. The main features of this distribution are the *lightweight* and the *small* size of the binary, the fact that, unlike traditional Kubernetes, which runs its components in different processes, K3s runs the control plane, kubelet, and kube-proxy in a *single* Server, or Agent, *process*, and also k3s can use seamlessly *containerd*, which it ships inside itself, or an existing *Docker* installation instead. All of the embedded K3s components can be switched off, giving the user the *flexibility* to install their own ingress controller, DNS server, and CNI.

MicroK8s[18] is a single-package fully conformant lightweight Kubernetes that works on 42 flavours of Linux and was built to perfectly suit to developer workstations, IoT, Edge and CI/CD scenarios. The principal concepts under this distribution are: to be *small*, developers want the smallest K8s for laptop and workstation development, *simplicity*, minimize administration and operations with a single-package install that has no moving parts for simplicity and certainty, *security*, updates are available for all security issues and can be applied immediately or scheduled to suit your maintenance cycle, it's comprehensive, MicroK8s includes a curated collection of manifests for *common K8s capabilities and services* (Service Mesh, Serverless, Monitoring, Ingress, DNS, Dashboard, Clustering).

Kind[13] is a tool for running local Kubernetes clusters using Docker container as nodes. Kind was primarily designed for testing Kubernetes itself, but may be used for local development or CI. The main principles at the base of Kind are: *gacefully degradation*, as much as possible kind should not fail, because it is to be used for testing and partially degraded states can still be useful and still be debugged, *follow Kubernetes API conventions*, *minimize assumptions*, which currently are having Docker installed, that the "node" follows the standard format and the standard Kubernetes assumptions, *no external state*, it is offloaded into the "node" containers and this simplifies a lot of problems and eases portability.

2.2.4.4 Cloud distributions

The last category of distributions described are the cloud managed distributions offered by a large number of provider. The main competitors in this field are Microsoft, Amazon and Google with their offers of Kubernetes distributions that hides all the management of nodes and the complexity of the installation.

EKS Amazon Elastic Kubernetes Service[4] is a managed Kubernetes service to run Kubernetes in the AWS cloud and on-premises data centers. The term

"managed" means that certain aspects of Kubernetes, such as provisioning each node and connecting them to form the cluster, are managed by the cloud provider.

AKS Azure Kubernetes Service[2] simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance. Since Kubernetes masters are managed by Azure the user only manages and maintains the agent nodes.

GKE Google Kubernetes Engine[8] provides a managed environment for deploying, managing, and scaling your containerized applications using Google infrastructure. The GKE environment consists of multiple machines (specifically, Compute Engine instances) grouped together to form a cluster.

Chapter 3

Digital platforms

The term Digital Ecosystem Platform is composed by the word "ecosystem" that comes from biology and is a contraction of "ecological system" and it describes a system in which entities have a mutual dependence. In this systems, each member is complementary to the others and the overall value is made by the single components but also by their interaction.

A platform can be made by a fixed set of members dedicated wholly to that platform or by a dynamical set of entities that can enter, exit or change freely and participating in multiple platforms simultaneously.

3.1 Main characteristics

From the technical, legal and business-related points of view the difficulties found in digital ecosystems are significant, for example service orchestration, delivery and monetization, as well as customer communication and data management across the entire ecosystem constitute some of the most significant challenges.

Ecosystem Platforms focus on flawlessly connecting users and smart devices on a platform, guaranteeing a wide range of additional services to process the data. The platform ecosystem creates revenue streams from platform usage.

There is also a super category of ecosystem platform, often called super ecosystem platforms, which are the most complex type of digital ecosystem. They focus on integrating several platforms into one integrated service, while also capturing user data from the integrated platform.

The use of this resource lead to a faster adoption of technology. The service orchestration and management that is in charge of the platform allows companies to implement new technology in an easier a more faster way, allowing them to take full advantage of cloud services and SaaS.

Another important point that needs to be stressed out is the relation of these

platforms with orchestrators that allows them to be easily deployed, scaled and managed. In general the adoption of this technology is useful for having control on the number of replicas deployed for each services and scale in order to deny the overload of the resources and the deployment of a new instance of the platform is almost architecture independent because it leverages on Docker images that runs in pod managed by the orchestrator.

In the following sections some main examples of data ecosystem platform will be analyzed.

3.2 Thingsboard

ThingsBoard[25] is an open-source IoT platform that enables rapid development, management, and scaling of IoT projects which goal is to provide the out-of-the-box IoT cloud or on-premises solution that will enable server-side infrastructure for IoT applications.

The main features of this platform are:

- Provisioning of devices, assets and customers, and defining relations between them.
- Collecting and visualizing data from devices and assets.
- Analyzing incoming telemetry and triggering alarms with complex event processing.
- Controlling devices using remote procedure calls.
- Building work-flows based on a device life-cycle event, REST API event, RPC request, etc.
- Designing dynamic and responsive dashboards and presenting devices or assets telemetry and insights to customers.
- Enabling use-case specific features using customizable rule chains.
- Pushing device data to other systems.

The main concepts on which it leverages are **devices**, **attributes** and **alarms**.

The platform strongly based on the idea of **devices** that are basic IoT entities that can produce telemetry data and handle RPC commands, for example, sensors, actuators, switches.

ThingsBoard provides the ability to assign custom **attributes** to entities and manage them. Those attributes are stored in a database and may be used for data visualization and data processing.

Attributes are key-value pairs and this gives them flexibility and simplicity and allow easy and seamless integration with almost any IoT device on the market. The key is always a string and is basically an attribute name, while the attribute value can be either string, boolean, double, integer or JSON.

ThingsBoard provides the ability to create and manage **alarms** related to entities: devices, assets, customers, etc. For example, there is the possibility of creating an alarm when the temperature sensor reading is above a certain threshold. Of course, this is a very simplified case, and real scenarios can be much more complex.

3.3 Digital Enabler

Digital Enabler[3] is a data driven, cloud native, ecosystem platform. Designed and engineered by Engineering's R&D Labs over the past years, today it is one of the few available, fully functioning, ready to use, Cloud Ecosystems Platforms available on the market.

It is a multi-purpose platform for organizations, communities and marketplace end-users because it offers advanced analytics, data presentation, subscriptions and data monetization process.

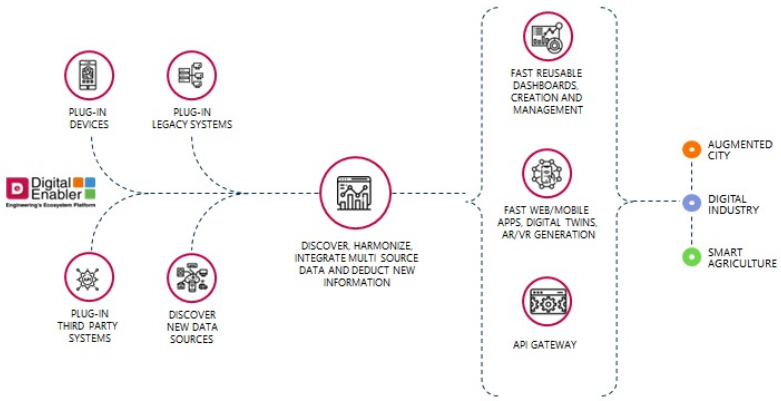

Figure 3.1: Digital Enabler workflow

It enables fast connection of various sources, including real-time streams, data integration through low-code approach, and rule-based action triggering. One of the tools included is the Discovery Engine which automatically discovers public data sources. It features a robust built-in API Manager and Gateway, greatly

improving interoperability and offers the possibility of publishing and accessing APIs through a marketplace. Another important aspect of this platform are all teh features that regards the edge computing and sensor management. It optimizes IoT potential by facilitating easy device connection, with control over deployed assets supported by integrated Big Data and AI tools. The key aspect of the Digital Enabler lies in the concept of ecosystem by fostering collaboration among businesses and utilizing open-data as a valuable asset.

3.3.1 Architecture

The platform is built to have a core component on the cloud and an edge one. Both of them are based on the same layered infrastructure but with some differences that are needed for the different scope of the two implementations. Starting from the most technical aspects, there is an orchestrator that in this case is Openshift. For what concerns the Data spaces, the platform offers several possibilities among which SQl and NoSql DBs, Time Series DB and Graph DBs.

Going up in the architecture there are different levels that describes a pahse of data management:

- **DataSource connection**: includes all the connectors that allows to create a connection to edge devices in roder to collect data .
- **Device Management & Data Management**: layer composed of tools that manages (add, configure and delete) devices and data sources in general.
- **Data Integration**: formed of software for mashups, serverless functions and workflows that allows to create custom flows to reorganize, reformat and enrich data..
- **Analytics & AI**: includes AI processes, early Warning and anomaly detection.
- **Application Enablement**: the upper layer which is composed by data visualization tools (dashboards and synoptic development) and the API gateway.

For what concerns the edge component it is linked to a gateway that act as a connection point between the single core component and the possible multiple edge instances. The gateway supports several protocols in other to be more flexible and avoid the vendor lock-in. The edge component has a structure that is very similar to the one described above but it has a different scope and among its main objectives there are:

- **Distributed Endpoints**: easy connectivity to High-Value Assets and external platforms, e.g. Eurotech, AWS and Azure.

- **IoT HUB capabilities**: device provisioning (eg. Status monitoring, Configuration management, Data plane), data availability (all data provided by IoT Devices are available and can be used for multiple purposes), IoT Devices data can be mashed up with data from other data sources, easy connection with with FIWARE/Eclipse Open Frameworks.
- **Integration capabilities**: Hybrid IoT ecosystems (single modules can be easily integrated to existing frameworks), Partner ecosystems (Eurotech, MindSphere, Telit, Sensors and IoT devices providers), Azure and AWS IoT frameworks (full compatibility with market platforms, for integrated solutions in greatly reduced time to market).

3.3.2 Services

The main services offered by the DigitalEnabler are:

- **Data discovery**: identify potential data sources within an ecosystem (e.g. city).
- **Data collection**: collect insightful data and make it easily exploitable via the platform.
- **Low/No code Data integration adn harmonization, workflow**: Mashup multi-source data to create new knowledge, harmonize data to standard data models. Define and monitor data pipelines.
- **IOT & Edge Management**: Orchestrate the IoT backbone making available a wide set of standard protocols to interact with field devices.
- **Rule Engine, Advanced analytics, Serverless**: Rules and AI to trigger actions based on conditions and identify data trends, apply algorithms and Machine Learning models to deduct new valuable information. Developers are allowed to upload and run any code in a serverless manner.
- **Data visualization and provisioning**: Visualize data through intuitive and replicable dashboards built in a self-service manner, Digital Twins, AR/VR apps and APIs.

3.3.3 Challenges

The Digital platforms have the main purpose of managing data and this arises several problems regarding the privacy of the personal data and the storage of them. Companies which use personal data have to follow a strict regulations, variable depending on geographical region or field of application (e.g. the medical

sector has different requirements from the financial one). This causes conflicts with the usage of ecosystem platforms that leverages on the cloud to have better performances and scale rapidly. In fact, because of this strictness, is not allowed to transport personal data over the Internet to a central powerful cluster to perform some operations on them.

In particular the concepts to which digital platforms must comply are the **data management at rest** and **data sovereignty** that will be further explained in next chapter together with some of the principal worldwide regulations regarding data protection.

Part III

Data protection regulations

This is becoming increasingly important as more data needs to be secured. While the use of the internet is widespread, not all countries have laws in place. In fact, data from the United Nations Conference on Trade and Development (UNCTAD)[26] reveals that only 71% of countries have data protection legislation, while 5% of countries do not produce data. The map below depicts the global distribution of these regulations.

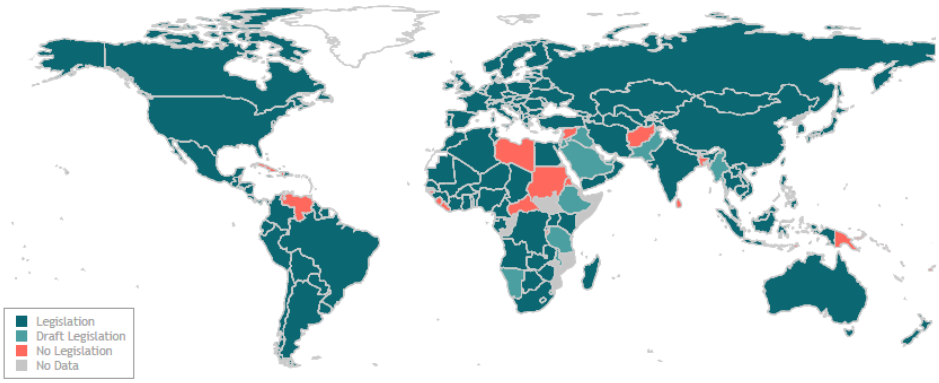

Figure 3.2: Data Protection and Privacy Legislation Worldwide[26]

In particular regions such as Asia and Africa has respectively only the 57% and 61% of countries covered by a legislation. In the following part a deep analysis will be carried out in particular on the European regulation (GDPR) and then on the USA regulations, the Chinese and the Brazilian ones.

Chapter 4

GDPR

The General Data Protection Regulation (GDPR) is the strongest law regarding data privacy and security in the world. It was approved by the European Union but it has value on countries and organizations around the world that collect data related to people in the EU.

4.1 History

The concept of privacy rights has its origins European Convention on Human Rights of the 1950 which states: "*Everyone has the right to respect for his private and family life, his home and his correspondence*".As technology advanced rapidly over the years, regulations had to keep pace. Some examples of this rapid ascent are the project ARPANET (forerunner of internet) was built in 1969, the 26 March 1976 the queen Elizabeth II sent an email to the Royal Signals and Radar Establishment, in 1979 the emoticon where invented and in 1991 the HTTP protocol was defined by Tim Berners-Lee. So in 1995 The European Data Protection Directive was releases. The directive established minimum standards for data privacy, and each country used it as a basis to create its own laws. Then many companies were created like Google (1998), Facebook(2006) and the first privacy problems arises. In 2011 Goggle was sued by a user for scanning her emails. Google confirmed that Gmail has used an automated scanning technology to show users relevant advertisements that help to keep the service free. So it was evident that the regulation on data protection had to be updated. In particular the Europe's data protection authority declared the EU needed "**a comprehensive approach on personal data protection**".

The GDPR come into effect in 2016 after passing the European Parliament and all organization were required to be compliant for the 25 May 2018.

4.2 Key concepts

The GDPR[7] is a regulation aims to address most scenarios concerning the manipulation and storage of personal data and outlines the corresponding penalties for breaking the rules. It is important to note that personal data refers to any information that relates to an individual that can be directly or indirectly identified. This includes, but is not limited to, names, email addresses, location information, ethnicity, gender, biometric data, religious beliefs, web cookies, and political opinions. Even pseudonymous data can be considered personal data if it is relatively easy to identify someone from it. The GDPR requires **data controllers**, the individuals who decide how and why personal data will be processed, to demonstrate their compliance with the regulation. If they cannot do so, they are considered non-compliant and subject to fines, which can reach up to €20 million or 4% of their global revenue, whichever is higher. Additionally, the **data subject**, the person whose data is being processed, has the right to seek compensation for damages. In the next section, the main principles of the GDPR will be discussed.

The Article 5 regulates, according to seven principles, the way in which data must be processed. For data processing is intended any action performed on data, whether automated or manual.

- **Lawfulness, fairness and transparency** : processing must be lawful, fair, and transparent to the data subject.
- **Purpose limitation**: data processing propose must be legitimate and specified explicitly to the data subject when collecting information.
- **Data minimization**: collected and processed data must be the only absolutely necessary for the purposes specified.
- **Accuracy**: personal data must be kept accurate and up to date.
- **Storage limitation**: personally identifying data may only be stored for as long as necessary for the specified purpose.
- **Integrity and confidentiality**: processing must be done in such a way as to ensure appropriate security, integrity, and confidentiality (e.g. by using encryption).
- **Accountability**: the data controller is responsible for being able to demonstrate GDPR compliance with all of these principles.

Lawfulness, fairness and transparency

Whenever processing data there must be a good reason to do so and this is expressed in the GDPR by the concept of the lawfulness which lists a series of valid reasons for the collection and processing of data:

1) the user has given you consent to do so.
2) there is a contract to honor.
3) it's necessary to fulfill a legal obligation.
4) for protection of vital interests of a natural person.
5) it's a public task done in public interest.
6) proof can be provided that the data controller has legitimate interest, and it's not overridden by data subject's rights and interests.

The principle of fairness comes directly from the lawfulness. Essentially, it means that the data controller must not deliberately conceal information about the reasons for collecting personal data and how it will be processed. Adhering to this principle ensures that personal data is not misused or handled improperly.

Transparency, on the other hand, involves being clear, straightforward, and honest with data subjects about who is collecting their personal data, the purposes for doing so, and how the data will be processed.

Purpose limitation

This emphasizes that data collection should only occur for specific, explicit, and lawful purposes and that these purposes should be clearly defined and communicated to individuals via a privacy notice. The collected data must be used in accordance with the defined purpose and if it is necessary to use it for an incompatible purpose, the data subject must give explicit consent.

Data minimization

Data minimization requires that only the minimum amount of personal data necessary for the specified purpose be collected, avoiding the collection of excessive information that could result in privacy issues.

Accuracy

The accuracy principle requires data collectors to ensure the accuracy of the information they collect. This involves regularly updating and verifying the data to avoid incorrect or misleading information.

Storage limitation

The GDPR requires that the length of time data is kept stored must be justified. This is often done through the use of data retention policies. These policies outline what data should be stored, where it should be stored, and for how long. Typically, data that is not actively used is deleted or anonymized for historical reports after a specified period of time.

Integrity and confidentiality

This principle outlines the guidelines about the security of the collected data. It mandates the protection of personal data from unauthorized or unlawful access, as well as accidental loss, destruction, or damage. The data controller must take appropriate measures to ensure the security of the collected data, both against internal and external threats.

Accountability

According to the GDPR, supervisory authorities have the right to request evidence of an organization's compliance with the regulation's rules at any given time. As a result, organizations must have the necessary measures and documentation in place to demonstrate their adherence to the principles of data processin

4.3 Data management at rest

When speaking about data a differentiation based on the way data are used must be considered because there are different rules basing on the state in which data are:

- **Data at rest**: data that is not in use or being transferred is referred to as "data at rest." This refers to data that is stored on a physical local device or in the cloud. Data at rest is easier to secure, as it encompasses the majority of data throughout most of its "lifecycle".
- **Data in transit**: refers to data that is being transferred from one place to another, such as between two storage devices. It is considered the most vulnerable of the three data states, making it easier for malicious actors to execute Man in the Middle (MitM) attacks.
- **Data in use**: data that is being actively utilized or accessed at the present moment is referred to as data "in use." It remains in this state until it is either stored or transferred to another location.

In this section the focus will be on the **data at rest** because the use case developed manages data in this state and must be compliant to the related regulations. This category of data must undergo several best practices to ensure the privacy of the information stored:

- **Identify and locate data**: organizations handling customer data must be aware of which data is considered sensitive and where it is stored. To ensure the privacy of sensitive information, companies must implement processes to restrict its storage locations. However, this can only be accomplished they aren't able to properly identify the critical nature of their data.
- **Classify data**: data classification is an ongoing process that enables organizations to continuously evaluate the sensitivity of their data and adjust their data protection measures accordingly. This includes not only implementing encryption, but also establishing policies for managing encryption keys and implementing access controls.
- **Enforce encryption at rest**: data encryption is the process of converting data into an unreadable form that cannot be understood without the proper decryption keys. By encrypting data while it is stored, the data remains protected even if there is unauthorized access. However, encryption can impact performance, so it is important to properly classify data and choose the most appropriate encryption method to minimize performance issues, such as selectively encrypting specific database fields, rows, or columns.
- **Enforce access control**: various protective measures, such as access control using the principle of least privilege, backups, segregation, and version tracking, can all help secure data while it is at rest.

In the actual technological scenario with the spread of the cloud storage and computing many companies uses these type of services to store their data. This process, that functionally speaking, is completely transparent and smooth brings some privacy problems according to GDPR regulations because that data go on the internet and until they arrive in the remote storage they are data in motion, that are intrinsically less secure than data at rest. It is important to remember that the GDPR requires data collected on European citizens to be kept within Europe, which can restrict the use of cloud storage. To ensure GDPR compliance, it is recommended to adopt a hybrid cloud approach, keeping sensitive data on-premises for better control over data usage and location. This is a crucial aspect to keep in mind when considering the proposed solution later. The highlight of this point is of central importance to understand the solution proposed later.

4.4 Data sovereignty

Data sovereignty means that personal information is governed by the laws of the country in which it is collected and physically stored. This determines the legal rights of the individuals whose data is being gathered, retained, or processed, as these rights vary based on the geographical location of the data. Thus, organizations must comply with different requirements based on where their data is stored.

Data sovereignty is distinct from data localization and data residency:

- **Data localization**: refers to a government policy that requires organizations to keep data within a specific geographical location and not transfer it outside that area. It is a specific application of the principle of data sovereignty.
- **Data residency**: refers to the act of a company selecting a specific geographical location to store its data. This choice may be made to avoid legal requirements, benefit from tax laws, or for performance reasons. When data is stored in a particular location, the company becomes subject to the data sovereignty laws of that region.

Companies that use **cloud infrastructure** must take a comprehensive approach to addressing data sovereignty. Some nations have restrictions on transferring data outside their borders, and some states have privacy laws that limit the sharing of personal data with third parties. This means that businesses operating in these countries may be legally prohibited from transferring their data to third-party cloud providers for storage or processing. Additionally, cloud-stored data can be subject to the laws of multiple countries, creating a particularly challenging situation in hybrid-cloud scenarios, where the deployment must comply with different local legal requirements.

Data sovereignty, which concerns the laws and regulations regarding the storage and transfer of data, affects not just legal matters, but also has a significant impact on businesses. Data subjects who provide personal information expect guarantees about where their data will be stored, while providers aim to have the flexibility to choose the best geographical location for their data to optimize their cloud infrastructure.

Chapter 5

Other worldwide data management regulations

In this section, the focus will shift from European regulations to other data management laws that are globally relevant. The section will highlight the data management aspect of each regulation and compare it to the GDPR. Three regulations have been selected for comparison: the US situation, where each state has its own regulation and there is no federal regulation; the Chinese regulation, which is heavily influenced by state control; and the Brazilian regulation, which is inspired by the GDPR and showcases the growing need for data privacy regulation in developing countries.

5.1 US regulation

In the United States, there is not a singular law dedicated to data protection, but rather a multitude of regulations at both the federal and state levels that dictate how U.S. residents' data should be handled. The Federal Trade Commission Act grants the U.S. Federal Trade Commission (FTC) broad authority to take legal action against unfair or deceptive practices, as well as to enforce federal regulations governing privacy and data protection. Additionally, individual states have their own statutes that safeguard the privacy rights of their residents, with protections varying widely between states.

Regarding **data management**, the FTC has established that companies must implement "reasonable" security measures to protect personal data, taking into account various factors such as the amount and sensitivity of the information held, the company's size and complexity, and the cost of implementing appropriate processes. Some federal and state-level regulations also require companies to ensure the security of personal information, such as the GLBA and HIPAA, which

mandate security requirements for financial and healthcare entities, respectively. Certain states also impose data security obligations on specific entities that handle limited types of personal information. For instance, the New York Department of Financial Services (NYDFS) implemented regulations in 2017 that require "regulated entities" to implement a cybersecurity program and governance processes, and to report cybersecurity incidents like data breaches and attempted infiltrations to regulators.

5.2 Chinese regulation

In China, there are three distinct acts that make up the regulatory framework for data governance. These include the Cybersecurity Law (CSL), which was enacted in 2017, and the Data Security Law (DSL) and Personal Information Protection Law (PIPL), which were promulgated and implemented in 2021. The DSL and CSL focus on safeguarding national security and public interests, while the PIPL is primarily concerned with safeguarding personal information rights and interests during the processing of personal data.

The CSL consolidates a variety of laws and regulations related to cybersecurity under one comprehensive framework, with the goal of protecting China's national security, fighting online crime, and enhancing information and network security.

The PIPL, which is the most recent of the three laws, draws heavily on the European Union's General Data Protection Regulation (GDPR). Its primary objectives are to protect personal data, regulate its processing, and promote reasonable use of personal information.

The regulation of **data management** in China is primarily governed by the Data Security Law (DSL), which builds upon the Cybersecurity Law (CSL) by expanding the focus on national security and classifying data based on its importance to Chinese national security. The DSL creates two main categories of data: core data, which pertains to Chinese national or economic security, citizens' welfare, or significant public interests, and important data, which is a level below core data. However, the DSL does not provide precise definitions for these categories.

The DSL also imposes requirements on data localization and cross-border transfers for core and important data. For instance, operators of critical information infrastructure (CII) are required to store data generated in China within the country, and if data needs to be transferred abroad, a security assessment must be conducted beforehand. Moreover, the DSL prohibits CII operators and other network operators from disclosing any data stored in China to any foreign judicial or law enforcement body without the consent of Chinese authorities.

Similar to the PIPL, the DSL has extraterritorial scope, meaning that companies outside of China that handle data subject to the DSL's provisions are required

to comply with the DSL's requirements.

5.3 Brazilian regulation

The Brazilian General Data Protection Law (LGPD) is a regulatory framework that governs the protection of personal data in Brazil. It outlines regulations on processing activities, the rights of data subjects, obligations of data processing agents and data protection officers, information security parameters, and requirements for the international transfer of personal data. Modeled after the European GDPR, the LGPD places Brazil among the group of Latin American countries with a general data protection law. The law also has extraterritorial scope, which means that not only companies established in Brazil, but also entities that process or have collected data within Brazilian territory, and those that offer or provide goods and services to individuals located in Brazil are subject to its rules.

The LGPD is based on fundamental principles that are similar to those of the GDPR, including purpose, necessity, adequacy, and transparency. These principles require that personal data processing is conducted for specific and legitimate reasons, is necessary for the purpose, is adequate to fulfill the purpose, and that individuals have access to clear and easily accessible information.

In terms of **data management**, the LGPD and GDPR share similar principles such as explicit purpose of data collection and processing, transparency in all stages of data processing, and appropriate use of the data.

Part IV

State of art

As mentioned previously, digital platforms face challenges in managing data when deploying on cloud infrastructures, especially with regards to personal data location. This chapter will present potential approaches that could be considered to comply with data privacy regulations.

One option is to use multiple clusters, each located in a different region and compliant with local regulations.

Another solution is to engage a reputable and widely-used cloud infrastructure provider to manage data protection compliance on the platform's behalf.

Lastly, a liquid computing approach could be utilized to create a resource continuum across several clusters.

Chapter 6

Use multiple scattered clusters

This first solution is based upon the usage of several cluster, one for each region that has a different regulation. This solution is suitable for scenarios where the company that uses the digital platform has several users for each region and so each instance of the deployed platform can be exploited. The main advantage of this approach is the fact that each cluster will be independent from the other and should be compliant only to its local regulation. For example a typical scenario can be to have a cluster in Europe that stores and process data gathered from citizens of European countries while another cluster in China to work with data that refers to Chinese people. In this way the principle of data sovereignty and the location of data at rest is respected and at the same time the two cluster can have customization depending on different needs.

On the other hand this scenario requires a considerable amount of economic resources to provide the widespread cloud infrastructure and the other drawback is the fact that each cluster should be managed independently and needs a specific configuration which is an effort that cannot be ignored. About the first disadvantage, recalling the previous example, the provider company have to buy and maintain two different cluster with their associated costs, both hardware (e.g. electricity, facility space) and software (e.g. engineering and installation manpower, system monitoring) ones, which may be unsustainable for all enterprises. Regarding the second drawback highlighted the scenario presented above requires to have direct access to both clusters because they are physically and logically separated and this duplicate the effort and the costs. This would include the configuration of all the tools and processes present in the digital platform that may be the same in both installation. Another point that must be stressed out is the fact that both deployment are a single point of failure because the second one cannot act as a

backup one for the other one as it would break the data privacy principles.

6.1 Main characteristics

This approach is based on some principles that make this solution a valid opportunity in some scenarios:

- **Instances independence**: each instance in the super-cluster (the all infrastructure managed by the operator that includes all the clusters) is managed independently from the others in therms of hardware involved, and consequently costs, software and tools deployed and how data are processed.
- **Geographic location-based customization**: the possibility of creating different clusters in different locations is a feature that gives more flexibility to the operators. In particular this enforces the ability to adapt to customer needs or laws in place depending on the geographical location of the cluster in question.
- **Autonomous instance management**: the management of each instance is entirely handled by the operator int his approach. This allows all possible customization. The other important point that must be stressed is the avoidance of the vendor lock-in. This concept describe the situation where the cost of switching to a different vendor is so high that the customer is essentially stuck with the original vendor. With the autonomous management this problem is completely avoided, but it requires a greater economic and technical effort to perform this management.

6.2 Deployment scenarios

Each of the presented solutions is more suitable for a different customer and a different scenario. Regarding this first approach, some of the scenarios that best exploit this method will be presented.

(a) **Customized cluster replicas**: the considered solution perfectly fits in a situation where the operator needs to offer different services to customers. In this scenario having multiple clusters spread around the world allows to offer customized services depending on the customer's needs and location and it reduces latency by positioning clusters closer to the end user.

(b) **Huge worldwide spread company**: the other scenario is the one that address the main problem highlighted in this thesis: the data management

of users that are under different legislation. In this case having customized cluster with are compliant to local regulations is one of the possible solutions. Having an entire working cluster per area allows to store and process personal data collected from users in a certain area locally and this is achieved without sacrificing any performance.

Chapter 7

Use private provider technologies

The second approach to the highlighted problem leverages on some known and widespread cloud services provider. Buying the infrastructure from the world biggest companies gives the possibility to exploit their hundreds points of presence but with respect to the previous solution the managing of the infrastructure is in charge of the cloud provider. The main advantage of this solution is the outsourcing of all the low level cluster management to external service without giving up the flexibility in term of covered regions. The major drawback is the fact that this solution locks the platform, or some tools of it, to a particular private technology owned by the provider creating a dependence that will cause problems in case of necessity of changing infrastructure, this phenomenon is called **vendor lock-in**, as explained in the previous solution.

This solution descends from the previous one, adding a level of abstraction, and so it has the same advantage of the previous one in terms of cluster independence but also the high cost. Even if the infrastructure management is no more directly needed the cost of the cloud services is not negligible and, as said before it adds the vendor lock-in problem.

Coming back to the example presented in this chapter the company that has the digital platform and wants to deploy it should buy the IaaS in the two regions where it has the users, so the provider to which the enterprise buys the service should be big enough to cover that areas.

7.1 Main characteristics

This second solution is aimed at customers with different needs than the previous one, but it derives from it so the first two characteristics are in common:

- **Instances independence**
- **Geographic location-based customization**

The big difference is in the third principle:

- **Outsourcing low level management**: this approach leverages on the services offered big providers that have resource spread all over the world. This is a great possibility because it avoids low-level resource management, but risks vendor lock-in. This feature is particularly useful in certain scenarios that will be described later. Additionally, the ability to outsource all hardware resource provisioning, and beyond, enables the use of this solution in situations where specialized personnel is not present.

7.2 Deployment scenarios

This solution, unlike the previous one, targets a specific audience, going into detail on certain aspects as previously highlighted. The main application scenarios are:

(a) **Small worldwide spread company**: the target of this solution are companies that does not have the budget or the possibility of having huge and complete self-managed clusters around the world so leveraging on services that provide this resource can be a perfect solution for implementing ecosystem platform compliant to the different data privacy regulations.

(b) **Huge worldwide spread non technology specialized company**: the outsourcing of the low-level management duties increases the possibility of implementing ecosystem platforms even by non-specialized operators, leveraging ready-made services and provider support for all technical cluster management.

Chapter 8

Liquid computing

The third solution refers to concept called **liquid computing** and make the most of edge devices without the need of having multiple replicas of the platform. This has the main advantage, in terms of costs and complexity, of not having to manage multiple clusters and it allows to have a central cluster that acts as a central point of control of the whole platform. It allows to exploit the edge computing resources, that are becoming more and more widespread, connecting them to the central powerful cluster that can control, manage and deploy tools on them. The second aspect that must be stressed is the fact that this solution connects standard clusters that can be already present or that can be shared helping the companies for what concerns financial issues.

This concept is deeper explained in this section with a drill down on some implementations that offers different additional features to the bare theoretical concept.

The liquid computing usually refers to a style of workflow interaction of applications and computing services across multiple devices, such as computers, smartphones, and tablets. The term was coined in July 2014 by InfoWorld, but the underlying concepts have long existed in computer science. An example of this approach is Apple's Handoff (Continuity) service in iOS 8 and OS X Yosemite. The focus of this section is a broader concept which encompasses the creation of a resource continuum composed of cloud and edge-grade infrastructures, on-premise clusters, as well as, in its widest form, single end-user and IoT devices. The information retrieved in this chapter takes inspiration from the official presenting paper of Liqo[9] and the Liqo official documentation[17].

8.1 Main characteristics

We envision liquid computing as a continuum of resources and services allowing the seamless and efficient deployment of applications, independently of the underlying infrastructure. Here are presented the main characteristics of liquid computing:

- **Intent-driven**: clusters have varying properties such as geographical location and security, which is why cluster spread exists. Adopting an internet-driven approach allows end users to specify high-level policies for their workloads, such as geographical location and cost constraints. Automated schedulers then choose the best place to execute the workload based on available resources and the policies specified by the user, across the entire infrastructure without borders.
- **Decentralized architecture**: The resource continuum operates with a peer-to-peer method, lacking a central authority for control and management, and no privileged members. Each entity can make independent, dynamic decisions on whom to connect with, facilitated by a dynamic discovery and peering protocol. This protocol automatically identifies available peers and negotiates peering agreements based on the demands and offerings of each participant.
- **Multi-ownership**: each entity retains full control over its own infrastructure, while making choices at any moment as to how many resources and services to share and with whom. Once a new peering connection is established, the target infrastructure's control plane is responsible for setting up the proper isolation measures such as resource quotas, network, and security policies.
- **Fluid topology**: entities have the flexibility to enter and exit the system at any moment, regardless of the size of their infrastructure, which can range from large data centers to Internet of Things (IoT) devices and personal devices. The approach aims to handle highly dynamic situations, characterized by frequent and unexpected connections and disconnections.

8.2 Deployment scenarios

Overall, the following section will present the three deployment scenarios that are mostly fostered by liquid computing.

(a) **Elastic cluster**: enables the transparent utilization of resources located in different locations, allowing for load balancing and the ability to handle spikes in demand (cloud bursting). This is particularly useful in edge computing scenarios, where computational capacity is limited, as well as in traditional cloud

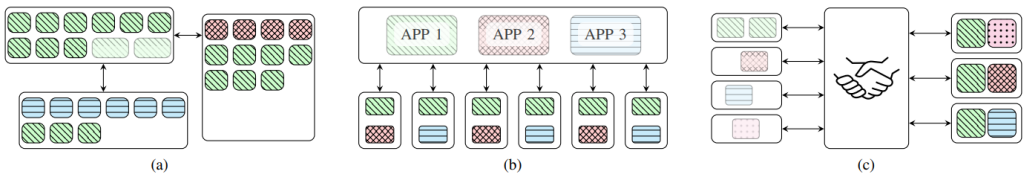

Figure 8.1: A graphical representation of the three deployment scenarios fostered by liquid computing

environments where multiple clusters are active. Entities engaged in resource sharing can be located in on-premise and public data centers, allowing for the deployment of applications with specific latency or data privacy requirements close to the end-users, while also taking advantage of the virtually unlimited computational power offered by larger infrastructures for resource-intensive tasks.

(b) **Super cluster**: liquid computing model allows for the creation of a higher-level cluster abstraction, acting as a single point of access for deploying and controlling applications across a vast infrastructure made up of hundreds or thousands of smaller clusters, often located at the edge of the network. Despite this centralization, the level-2 clusters maintain their own local orchestration logic, making them resilient to network disruptions and preventing potential synchronization problems with high-latency WAN connections. This approach streamlines the replication of jobs across clusters, enabling operators to easily replicate services on a subset of edge clusters to serve end-users in their immediate vicinity.

(c) **Brokering cluster**: liquid computing paradigm can promote the establishment of new Resource and Service Exchange Points (RXPs), which act as intermediaries between consumers and providers. For consumers, accessing services offered by providers becomes easier as they only need to connect with a single RXP to immediately benefit from the full range of resources and services available. On the other hand, resource providers are incentivized to participate as they can reach a larger number of potential customers with ease.

8.3 Admiralty

Admiralty[1] is a system of Kubernetes controllers that enables intelligent workload scheduling across clusters, which was launched at the end of 2018. It supports

multiple multi-cluster topologies, including the **Central Control Plane**, **Cloud Bursting**, and **Decentralized Federation**.

In the Central Control Plane topology, a central cluster, also known as the "management cluster," manages several workload clusters. This topology is useful when there is a powerful central cluster and some smaller edge clusters.

The Cloud Bursting topology is suitable for scenarios where there is a primary on-premises cluster with fixed resources that manages both itself and a secondary cloud-based cluster with elastic resources.

The Decentralized Federation topology is designed for organizations in different administrative and trust domains that manage different clusters and want to connect their control planes, share resources, or delegate operations without relying on a central cluster owned by a single organization.

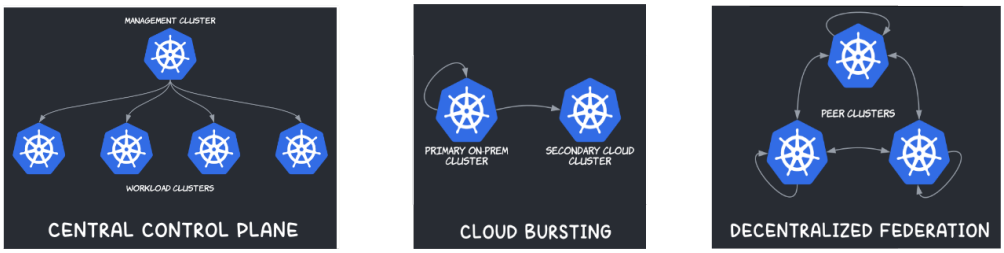

Figure 8.2: Admiralty multi-cluster tolopologies

The various topologies that Admiralty offers demonstrate how the solution can be applied to a range of use cases.

The solution proposed by Admiralty is based on a multi-cluster scheduling algorithm that leverages on four main components:

- a mutating pod admission webhook
- the proxy scheduler
- the pod chaperon controller
- the candidate scheduler

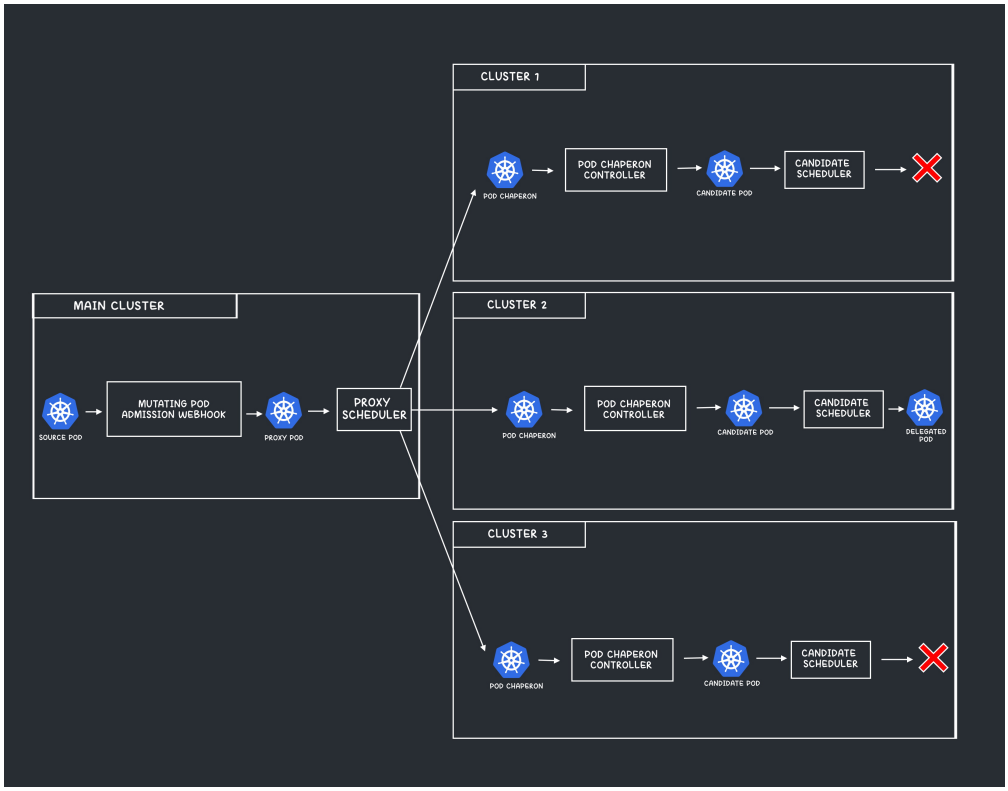

Figure 8.3: Admiralty scheduling process

The cluster where the source pod is created runs the **mutating pod admission webhook** and the **proxy scheduler**, while the target cluster(s) where the pod will be deployed is managed by the **pod chaperon controller** and the **candidate scheduler**.

The first step is to transform the user-requested source pod into a proxy pod through the **mutating pod admission webhook**, saving the original manifest and modifying scheduling constraints to select virtual nodes and the proxy scheduler

The **proxy scheduler** oversees the scheduling and binding of proxy pods to virtual nodes representing target clusters. Candidate pod chaperons are created in each target cluster, and later, the candidate scheduler in that cluster reserves a node for the corresponding candidate pod, if available.de for the corresponding candidate pod if available.

Pod chaperons are custom resources that are intermediate objects between proxy pods and candidate pods. The proxy scheduler creates the candidate pod chaperon using the original spec of the pod saved in the first step and sets the scheduler to the candidate scheduler.

The **pod chaperon controller** creates a candidate pod using the same spec

field as before and updates the pod chaperon status from the **candidate pod** status. If the candidate pod is deleted directly, the controller replaces the pod even in case of split-brain scenarios.

The **candidate scheduler** of each target cluster schedules candidate pods to regular nodes. If the candidate pod is unschedulable, the proxy scheduler sees it from the pod chaperon's status, stops waiting, and rejects the virtual node. If it succeeds, the candidate scheduler waits for the proxy scheduler to select a candidate pod as the **delegated pod**. At this point, the virtual nodes are filtered and scored using Kubernetes functions/plugins, and only one virtual node is finally reserved for the proxy pod.

The proxy scheduler sees the success or failure of the candidate scheduler from the pod chaperon's status. If it succeeds, all other candidate pods are deleted. If it fails, it is put back in the scheduling queue. This process is done at this point because the candidate schedulers have all the knowledge required to determine if those pods can be scheduled, unlike the proxy scheduler which knows little about the target clusters.

8.4 KubeEdge

KubeEdge[14], which was initiated in 2018, is an open-source platform aimed at expanding the containerized application orchestration capabilities to the edge. It was primarily designed to address the requirements of the IoT industry, where there is a need to better utilize the resources of edge devices and leverage a familiar technology for central control. KubeEdge is based on Kubernetes and provides fundamental infrastructure support for networking, application deployment, and metadata synchronization between the cloud and the edge. With Kubernetes-native support, KubeEdge enables users to utilize the standard Kubernetes APIs for interacting with the infrastructure and managing edge devices through Custom Resource Definitions (CRDs).

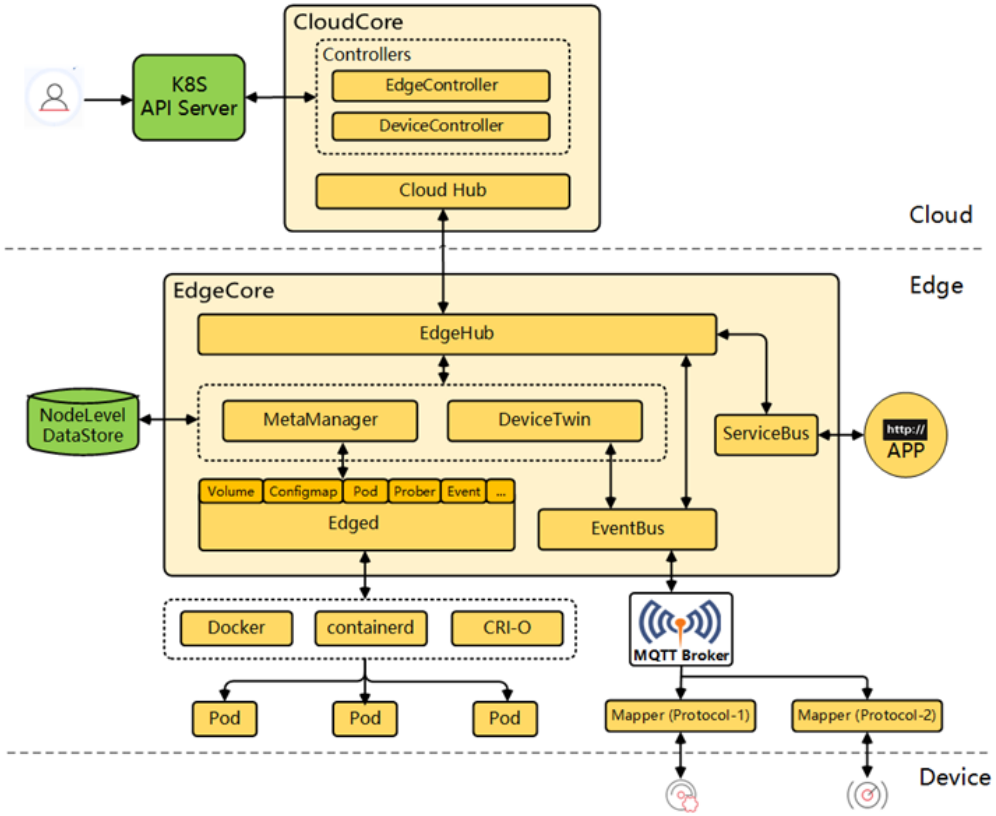

Figure 8.4: KubeEdge architectural schema

The KubeEdge architecture is asymmetrical and so the components are divided in **Cloud Components**, the one that runs inside the remote core, and **Edge Components**, the ones that are deployed in the edge device.

In the first set are included: the **CloudHub**, the **EdgeController** and the **DeviceController**.

The middleware component of the CloudCore, known as the **CloudHub**, facilitates communication between Controllers and the Edge side by reading and writing messages from the Edge and publishing them to the Controllers. To establish a connection to the Edge, the CloudHub uses HTTP over WebSocket, specifically through the EdgeHub module. For internal communication, it directly communicates with the Controllers.

The **EdgeController** acts as a bridge between the Kubernetes API-Server and EdgeCore, and its primary functions include relaying add, update, and delete events from the K8s API-Server to EdgeCore, monitoring and updating the status of resources and events (such as nodes, pods, and configmaps) to the K8s API-Server, and subscribing to messages from EdgeCore.

The **DeviceController** is a key component of KubeEdge responsible for managing devices. KubeEdge leverages Kubernetes CRDs to define device metadata and status, with the device controller facilitating synchronization between edge and cloud. The device controller initiates two independent goroutines to manage this synchronization: upstream and downstream controllers. The upstream controller synchronizes device updates from edge to cloud via the device twin component, while the downstream controller synchronizes device updates from cloud to edge by monitoring the API server.

Device management in KubeEdge is achieved through the use of device models and device instances. The former describes device properties and methods to access them, serving as a reusable template for creating and managing multiple devices. On the other hand, a device instance is an actual object representing a specific device, instantiated from the device model and referencing properties defined in the model. The device specification is static, defining the desired state of device properties, while the device status contains dynamic data reflecting the current device state.

Here there is a representation of the flow in case of an update from Edge to Cloud.

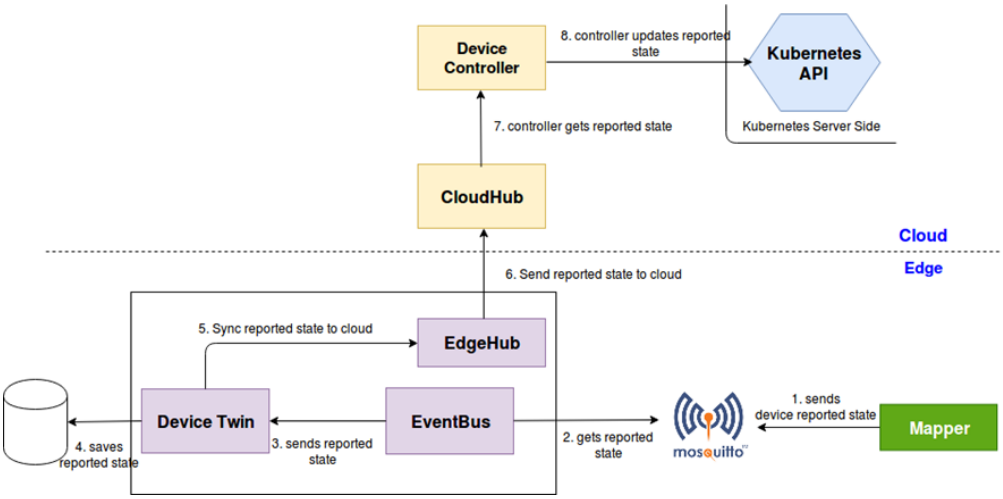

Figure 8.5: KubeEdge flow example from Edge to Cloud

The mapper monitors devices for any updates and transmits them to the event bus using the MQTT broker. The event bus then forwards the device's reported state to the device twin, which saves it locally and synchronizes the updates to the cloud. In parallel, the device controller keeps an eye out for any device updates coming from the edge, which it receives through the cloudhub, and updates the reported state in the cloud accordingly.

The following graph shows the update from Cloud to Edge.

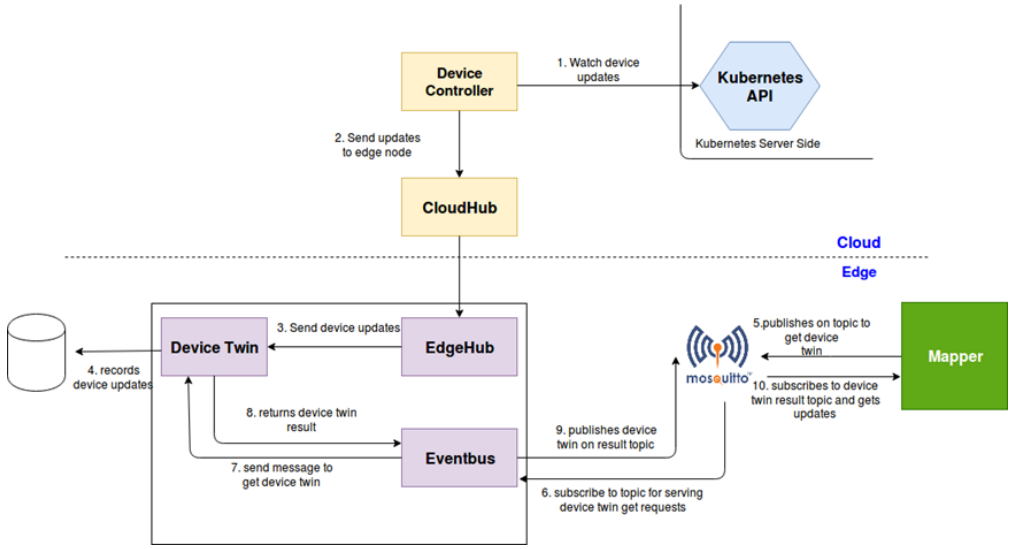

Figure 8.6: KubeEdge flow example from Cloud to Edge

The device updates in the cloud are monitored by the device controller, which then propagates them to the edge node. These updates are also stored locally in the device twin. The mapper receives the updates via the MQTT broker and performs actions on the device accordingly.

In the Edge Components are included: **EdgeD**, the **EventBus**, the **EdgeHub**, the **DeviceTwin**, the **MetaManager**, and the **ServiceBus**.

EdgeD is an edge node module which manages pod lifecycle. It helps users to deploy containerized workloads or applications at the edge node. There are many modules which work in tandem to achieve edged functionalities, among them there are:

- Pod Management: this component manages the addition, deletion, and modification of pods, while also monitoring their health status through the pod status manager and the Pod Lifecycle Event Generator.
- Pod Lifecycle Event Generator: this module is designed to monitor the status of pods in the edge environment. It updates the pod status manager for each pod with information obtained every second using probes for liveness and readiness.
- Secret Management: secrets in edged are managed independently with separate configuration messages and interfaces for operations such as addition,

deletion, and modification. The cache store is updated using these interfaces to manage secrets. Additionally, secrets are kept separate from other resources for security purposes.

- Volume Management: the volume manager runs as an edge routine and is responsible for handling volume operations such as attachment, mounting, unmounting, and detachment based on the pods scheduled on the edge node. Prior to starting a pod, the specified volumes referenced in the pod specifications are attached and mounted. During this time, the flow is blocked to prevent other operations from taking place.

- MetaClient: is an interface of the Metamanager for the edged module. It enables the edged module to retrieve details of ConfigMaps and secrets from the Metamanager or cloud. It also sends synchronization messages, node status, and pod status to the Metamanager in the cloud.

- Container Runtime: through its Container Runtime Interface (CRI), EdgeD provides a plugin interface that allows for the use of various container runtimes such as Docker, containerd, CRI-O, and others, without requiring recompilation.

The **EdgeBus** module is responsible for facilitating communication between the cloud and the edge by interacting with the CloudHub components in the cloud. Its functions include synchronizing updates to cloud-side resources, reporting changes in host and device status on the edge, and forwarding messages between the cloud and the edge to the appropriate modules.

The **DeviceTwin** module plays a vital role in KubeEdge by storing the device status, managing device attributes, and handling device twin operations. It establishes a connection between the edge device and edge node and ensures that the device status is synced to the cloud. The module is also responsible for synchronizing the device twin information between the edge and cloud, and it provides query interfaces that can be used by applications.

The **MetaManager** is the message processor between Edged and Edgehub. It's also responsible for storing, retrieving metadata to and from a lightweight database (SQLite).

The **EventBus** acts as an interface for sending and receiving messages on MQTT topics.

The **ServiceBus** is an HTTP client designed to communicate with applications running on the edge. It functions similarly to the EventBus, but uses a different protocol. When a message passes through the ServiceBus, the flow is as follows: The cloud sends a message to the edge via CloudHub. EdgeHub receives the

message and passes it to the ServiceBus, which makes an HTTP call and sends the response back to the cloud via EdgeHub.

The components inside the KubeEdge architecture are linked through Beehive, which is a messaging framework based on go-channels and it is used to add or remove modules to the edgecore or to a group.

8.5 Liqo

Liqo is an open-source project that supports the vision of liquid computing described above. It extends Kubernetes to enable the creation of dynamic and seamless multi-cluster topologies, regardless of underlying infrastructure limitations. Liqo achieves this without introducing modifications to standard Kubernetes APIs for application deployment, making it compatible with a wide range of common infrastructures and cluster types, with no restrictions on networking configurations. In the following section, we will present the main architectural features of Liqo and focus on its actual implementation.

8.5.1 Architecture

- **Discovering and Peering with Remote Clusters**: the Liqo discovery logic plays a crucial role in identifying potential remote clusters for peering. The output of this process is a remote network endpoint that can be utilized to initiate the authentication procedure, which includes the desired peering state and any additional attributes that may be needed. This information is then represented through a ForeignCluster Custom Resource (CR).

  The peering procedure is divided into several phases, beginning with the authentication process, which involves a private key generated by server A and a Certificate Signing Request. Once cluster B has been selected as the desired target, the negotiation phase begins, and a ResourceRequest CR is generated locally by cluster A to request computational resources and/or services from the remote cluster. The CRDReplicator is then activated to duplicate the CR on the remote cluster. When the ResourceRequest is received, cluster B discovers cluster A and initiates the symmetrical authentication procedure. If the request made by cluster A is accepted, cluster B sends a ResourceOffer CR to indicate its willingness to share available resources/services at a specified price, and duplicates it to cluster A through the CRDReplicator. Upon acceptance of the ResourceOffer, the peering relationship is established, and the inter-cluster network fabric is established. Furthermore, the reverse procedure can be initiated by cluster B, allowing for bidirectional peering.

- **Virtual Node Abstraction**: Liqo uses the virtual node concept to mask the resources that are shared by each remote cluster. This approach enables the local cluster to seamlessly expand by taking into account the virtual node(s) when determining the optimal location for executing workloads, using the vanilla Kubernetes scheduler. To implement this virtual node abstraction, an extended version of the Virtual Kubelet project[27] is used. This project replaces a traditional kubelet when managing non-physical nodes, which allows it to control arbitrary objects via standard Kubernetes APIs. Therefore, the virtual kubelet (VK) enables custom logic to handle the lifecycle of both the node itself and the pods hosted within it. The virtual kubelet is responsible for the following three features:
  - *Node lifecycle handling*: the virtual kubelet (VK) manages the virtual node, which conceals the shared resources of the remote cluster. It ensures that the node status conforms to the negotiated configuration (ResourceOffer), and performs regular health checks to evaluate the accessibility of the remote cluster.
  - *Pod lifecycle handling*: unlike a traditional kubelet, which starts containers on a designated node, the Liqo VK implementation maps each operation to a corresponding twin pod object in the remote cluster for actual execution. Remote status changes are automatically propagated to the respective local pods, allowing for proper monitoring and administrative inspection. To create an offloaded pod, a user requests the execution of a new pod, either directly or through higher level abstractions (such as Deployments), which is then assigned by the Kubernetes scheduler to a virtual node. The corresponding VK instance creates a twin copy of the pod in the remote cluster using a ShadowPod, a CR that wraps the pod definition and triggers the remote enforcement logic. This approach avoids resiliency problems in split-brain scenarios that can cause service disruption if remote pods are deleted following node failure or eviction.
  - *Resource and service reflection*: the Liqo VK handles the remote propagation and synchronization of artifacts necessary for proper execution of offloaded workloads. Currently, it supports shadow ConfigMaps and Secrets, which store application configurations, as well as shadow Services and EndpointSlices to enable intercommunication between microservices distributed across multiple clusters.
- **Workload Scheduling Policies**: a set of labels, key/value pairs describing the main characteristics of each peered remote cluster, is associated with it and automatically propagated to the corresponding virtual node. These labels are configured by the cluster administrators and enable fine-tuned selection

of the cluster(s) on which each workload should be executed, based on the workload's requirements and the current resource continuum capabilities.

- **Liqo Network Fabric**: the liqo network fabric extends the Kubernetes network model across multiple independent clusters, enabling offloaded pods to communicate with each other as if they were locally executed. To achieve this, the liqo networking module enhances the standard Kubernetes feature that allows pods on a node to communicate with all pods on any other node without NAT translation. The liqo module ensures that all pods in a given cluster can communicate with all pods on all remote peered clusters, with or without NAT translation. However, since liqo supports arbitrary clusters with completely uncoordinated parameters and components (such as CNI), it is impossible to guarantee non-overlapping pod IP address ranges (i.e., PodCIDR). Therefore, liqo supports IP translation mechanisms, providing NATless communication whenever the address ranges are disjointed.

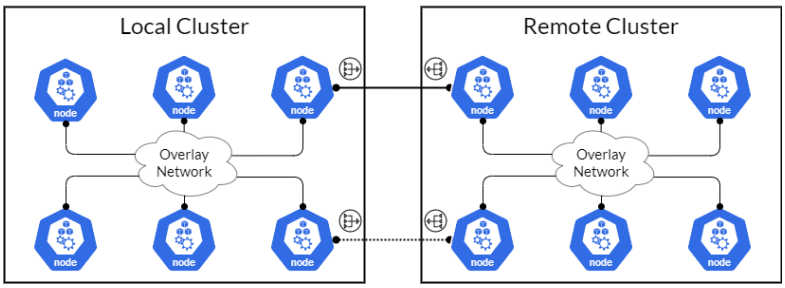

Figure 8.7: Liqo network fabric high level representation

- **Liqo Storage Fabric**: liqo transparently enables also the offloading of stateful tasks through a transparent inter-cluster storage continuum. The solution is based on two main pillars:
  - *Storage binding deferral*: to ensure that storage pools are created in the exact location where their associated pods are scheduled for execution (whether local or remote), the pools are not created until the first consumer is scheduled onto the cluster.
  - *Data gravity*: the automatic policies come into effect during subsequent scheduling processes, directing pods to the appropriate cluster. This ensures that pods requesting existing storage pools (e.g., following a restart) are scheduled onto the cluster where the corresponding data physically resides.

  Kubernetes uses PersistentVolumeClaim (PVC) to request storage, which

eventually leads to the creation of a PersistentVolume (PV) either manually or through a StorageClass. Liqo implements a virtual storage class to create storage pools on different clusters. If the target is a physical node, PVC operations are remapped to a second node associated with the corresponding storage class to provision the volume transparently. However, for virtual nodes, the reflection logic creates a remote shadow PVC, remapped to the negotiated storage class, and synchronizes the PV information to enable pod binding.

8.5.2 Implementation details

Liqo's codebase is written in Go, and its user-facing APIs and internal status are defined using Custom Resource Definitions (CRDs). The operators pattern is used to implement the logic, where each controller is responsible for ensuring that the observed status in the cluster aligns with the desired status expressed through the relevant resources. Besides the network fabric already described, liqo includes the following four components:

- *liqo-controller-manager*: similarly to the Kubernetes controller manager, it groups together all the different operators dealing with liqo resources.
- *liqo-virtual-kubelet*: is designed to run one replica for each remote cluster, providing isolation and separating the different authentication tokens. This control plane is responsible for managing the lifecycle of the virtual node and the pods running on it, as well as for resource and service reflection.
- *liqo-crd-replicator*: component responsible for the interaction between peering clusters, enabling resource negotiation and network setup procedures through the exchange of CRs.
- *liqo-webhook*: a mutating webhook is used to determine which subset of pods can be scheduled on virtual nodes based on system administrator policies. These policies may include strict constraints based on an intent-driven approach.

8.6 Comparison

Technical aspects

The solutions described in the previous sections offer different approaches to the same problem. All of them leverages on the Virtual Kubelet that is an open-source Kubernetes kubelet implementation that masquerades as a kubelet for the

purposes of connecting Kubernetes to other APIs. Admiralty, for example, in terms of performance, because of its scheduling-driven approach, has worse performance than Liqo when deploying a high number of pods.

KubeEdge, like Liqo, enforces the concept of edge autonomy that consists in keeping the edge nodes (and resources) running even when the network between the cloud and the edge is unstable or offline. The main differences between Liqo and this solution are that Liqo connects standard independent working clusters while KubeEdge connects edge devices to a CloudCore. The CloudCore contains an extended version of standard Kubernetes components instead at the edge runs EdgeD, which is a custom module that manages the pods lifecycle and other components useful for service exposure and status reflection. The second difference is that, for the reasons explained above, KubeEdge has a unidirectional connection with a predefined core and edge and not all the nodes have the same importance. Furthermore, KubeEdge does not work with Openshift at the moment this thesis is written and it has a strong limitation in the deployment scenarios in fact it is available only for Ubuntu and CentOS OS.

Interest

Analysing data on GitHub KubeEdge has the highest number of stars, 5.1k, and watch, 173. Behind it are Liqo and Admiralty with respectively 655 and 490 stars, 20 and 14 watches.

	Liqo	Admiralty	KubeEdge
Stars	655	490	5.1k
Watch	20	14	174
Forks	55	70	1.4k
Commits	1368	244	4288
Contributors	28	5	26
Start date	11/2019	12/2018	09/2018
Last update	8/07/2022	17/03/2022	12/07/2022
Percentage solved issues	122 on 142 86%	48 on 70 69%	1072 on 1277 84%

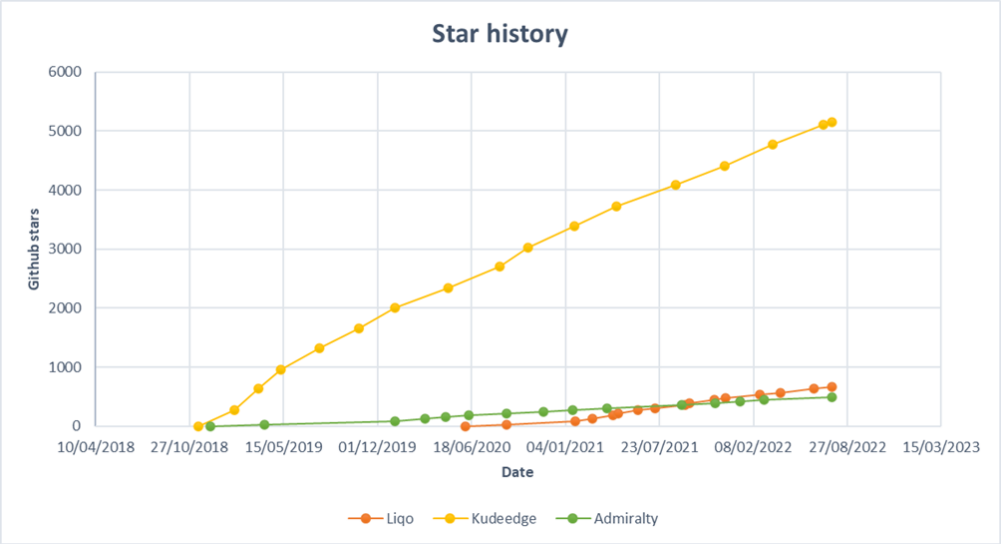

Figure 8.8: Alternatives community comparison

Further looking at the star trend, the picture above shows how, despite being more recent, Liqo is growing faster than Admiralty, while the KudeEdge fame is still rising and the trend does not seem to slow down.

Development

Looking again at the statistics inside the GitHub repositories KubeEgde has a way higher number of forks, 1.4k, and commits, 4274, in the main branch. Also in this category at the second and third place are Liqo with 55 forks and 1368 commits and Admiralty with 70 forks and 244 commits. In this case the number of forks and commits shows different trends and this is maybe also influenced by the number of contributors, way higher in Liqo than in Admiralty, that participate in the main project and don't fork the project on themselves. The number of contributors in Liqo is higher also than the one in KubeEdge representing a bigger developer community.

Updates

In this category Liqo and KubeEdge emerge because they are continuously updated, even at this time, then there is Admiralty that has his last commit in March of this year. Must be also highlighted that Liqo projected started in 2019 and Admiralty and KubeEdge in 2018. When speaking about updates must be taken into consideration even the way in which developers takes care of issues. In this category Liqo and KubeEdge has similar percentages, but KubeEdge has 10

times the number of total issues. At the last place there is Admiralty with a small number of issues and a not so good percentage of solved ones.

8.7 Other minor alternatives

8.7.1 Tensile-kube

tensile-kube[24] is a project that ensures high utilisation in case of resources fragmented across multiple clusters. This means that it leverages on the resource continuum abstraction of different clusters not geographically close exploiting at the maximum the available resources. It supports remote pod offloading, as well as resource reflection, along with custom scheduling. It doesn't support resilience mechanisms in case of split brain scenarios so in case of loss of connectivity between the clusters the remote cluster doesn't manage the lifecycle of the offloaded pods on its own. From the community point of view it has a small number of stars, watches, forks and commit with respect to the other competitors highlighted previously. Furthermore, it doesn't have a website but only the repository on GitHub, so it is harder to find information and documentation about it and in general interacting with it is more difficult. From the updates point of view tensile-kube hasn't been updated in the last 9 months and it is the most recent born project among the exposed ones.

Part V

Design and proof-of-concept

Chapter 9

General concept

This thesis addresses the proposal of expanding control over data localization while still utilizing standard tools such as Kubernetes. The use of cloud platforms for data processing raises the issue of **data sovereignty**, which means that the data is subject to the laws and regulations of the country where it was collected. This presents challenges for data owners who want their data to be stored nearby and for data controllers who face restrictions related to the physical location of data during IT system implementation due to official regulations.

One potential solution to this issue is to store the data at the edge, closer to the source of collection. However, this requires access to remote infrastructure for installation, management, and support, which may not always be possible and could result in slower processes or a need for customized configurations for each specific case.

The issue being addressed raises necessary conditions that have been analyzed in order to make the best decision on the methodology to propose the most suitable solution. The macro-requirements that emerged are:

- having the possibility of processing data from users that come from every part of the world.
- deploying the platform with customization, even in terms of which tools are installed, depending on the needs.
- connecting to IoT devices that are used to collect data.
- allowing the connection third-party clusters or devices that are already in place to deploy tools or collect/store data.

With regards to the first issue, all three proposed solutions provide a way to address it. However, the first solution is slightly inferior as it requires starting from scratch to set up new clusters, whereas the third solution utilizes existing

resources, and the second solution involves purchasing pre-configured ones. All three solutions offer the potential to meet the second requirement through different architectures and processes. Regarding the third requirement, the best option appears to be liquid computing, as it directly offers a way to connect IoT devices. The first solution involves installing a light version of an orchestrator on the IoT device, but it is not a straightforward or manageable method. As for the solution that relies on a provider, it can be difficult to find an operator that offers this possibility. The last criteria involves the ability to utilize third-party resources, which is achievable through the third proposed solution. The other two solutions do not provide this possibility unless the resource manager makes the credentials available, but this is not necessary with liquid computing through peering.

Therefore, it can be stated that the best solution in this case is liquid computing because it solves all the requirements imposed by the problem at hand in the best way.

Specifically, this approach eliminates duplications and cost increases while promoting higher flexibility and easier management of the platform as a whole. The chosen solution allows for data storage in compliance with data protection regulations and central management and control of all edge deployments, as all instances are perceived as part of a single cluster. This is a crucial aspect, as it facilitates the release of new versions of platform tools and troubleshooting without the need for direct access to the edge cluster.

The solution proposed here is based on some choices according to the research on the state of art presented previously.

Chapter 10

Use cases

After the general concept explanation, the following chapter will focus on the two main use cases taken into consideration, starting from the theoretical idea and then going inside he actual implementation.

The two use cases cover the remote data storage and the remote data processing using AI flows.

10.1 Database offloading on Edge

The first use case studied consider the problem of storing data according to the data protection regulations or any other particular requirement that may be present in the specific case. This requirements often include the limitation in terms of geographical location of sensible data and so standard cloud Persistence as a Service cannot be used. An example use case of this concept is in the medical field, where sensitive data is collected by hospitals or other structures and must be stored. There are privacy policies that these data must adhere to, but the goal is to be able to store them locally while still leveraging the full potential of a digital platform with all its services.

The proposed solution involves configuring a single cluster using the principle of liquid computing. This cluster would be located between an edge device, which could be present in the data collection structure or elsewhere, and the digital platform's tools used for data storage.

10.2 Remote Data processing

Another use case that was considered involves running processing algorithms in a remote instance. This can be beneficial in terms of regulations, such as applying

anonymization algorithms at the edge before sending data to the central cloud or performing AI processes faster without the need to move data.

Although this use case has not been implemented yet, it is a highly valuable application that will be included in future implications.

Chapter 11

Implementation

This chapter will cover the implementation of the previously presented use case, detailing the technology choices made and explaining the reasoning behind them.

After conducting a comparison, Kubernetes was chosen as the most suitable technology for orchestration, particularly Openshift and K3s distributions, due to its widespread use and interoperability across various distributions.

The interconnection between the cloud and edge instances is achieved through the liquid computing concept, which was evaluated in the presentation of three possible solutions. The selected solution, presented in this thesis, offers several advantages from an architectural and financial perspective, as described earlier. The use of the Liqo tool ensures the continuity of resources and central management, guaranteeing service continuity even in the case of temporary connectivity loss. Furthermore, the use of Liqo allows for a significant simplification in terms of both cluster management and software development and deployment, thanks to the continuity of resources that allows hiding the distances between instances, such as in the discovery of services present within the entire cluster. The development of this project resulted in the resolution of several issues, leading to new features in the latest release of Liqo (v0.6.0), which was used in the final implementation.

For data processing, a digital ecosystem platform was selected, and specifically the DigitalEnabler solution with the Database Manager tool for creating persistence instances.

The product of this thesis is part of the Structura-X project, which is a part of the Gaia-X initiative that will be further explained in the following chapter.

11.1 Database offloading on Edge with Liqo and Digital Enabler

The project began with an exploration of Liqo, specifically its installation on Openshift. The testbed used was an internal Openshift installation at an Engineering data center, using the OKD version 4.6 that corresponds to Kubernetes 1.19. The nodes were running the FedoraCoreOS operating system, which is one of the supported systems along with CentOS and Red Hat release. The cluster was located behind a load balancer that exposed a public hostname, and services were exposed through NodePorts or through Openshift Router, which is an Ingress distribution with improved security and the ability to handle multiple weighted backends, as the cluster did not have the standard Kubernetes object named Ingress.

Initially, the installation and testing were performed between the Openshift cluster and a publicly hosted machine. Later, the Liqo installation was integrated into the Structura-X project, and peering was established with the federated partner within the project.

At the beginning of this research project, the latest Liqo release was v0.4.0, which presented some issues that were further explored and resolved in the subsequent release also thanks to the results of this thesis, v0.6.0.

This troubleshooting process required a deep dive into Liqo, including all its features and processes, as well as the Openshift platform and the networking configuration of the testbed.

Liqo offers two different methods of peering: the **out-of-band** one and the **in-band** one.

The typical method for peering is known as an "out-of-band control plane," as the Liqo control plane traffic travels outside the VPN tunnel that connects the two clusters. The VPN tunnel is created dynamically in a later stage of the peering process and is only utilized for cross-cluster pods traffic, as depicted in the figure below. This approach is well-suited for linking clusters controlled by different administrative domains, since each party only interacts with its own cluster: the provider acquires an authentication token, which is subsequently shared with the consumer to initiate the peering process. This mechanism requires each cluster to expose three endpoints: the Liqo authentication service, the Liqo VPN endpoint, and the Kubernetes API server, all of which must be accessible from the pods running in the remote cluster.

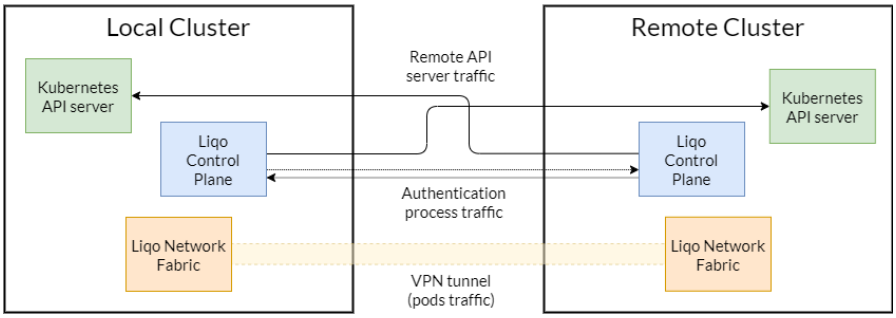

Figure 11.1: Liqo out-of-band peering[16]

The in-band approach creates a VPN tunnel to transfer all Liqo control plane traffic. This tunnel is established at the beginning of the peering process and all three cross-cluster traffic flows are transmitted through it, as shown in the diagram. Compared to the out-of-band approach, this method requires fewer peering requirements. The Liqo CLI tool performs the network parameter negotiation, and only the Liqo VPN endpoint needs to be accessible from the pods in the remote cluster. Unlike the out-of-band approach, the Kubernetes API service does not need to be exposed outside the cluster because the inter-cluster traffic flows through the VPN tunnel. Additionally, the administrator initiating the peering process must have access to both clusters.

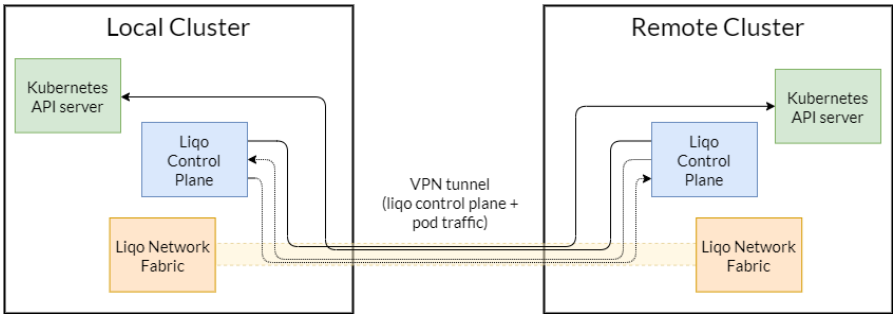

Figure 11.2: Liqo in-band peering[16]

The thesis project was developed in two different steps, each with its own scenario that required the use of different peering methods. In the first scenario, where only a single machine was being connected, the in-band method was used since access to both clusters was possible and the limited number of connectivity requirements made it more suitable, especially to avoid firewall-related problems. In the second scenario, which involved the Structura-X project, the out-of-band peering method was used since the clusters were under different administrative domains and this was the only viable solution.

The development of the Database Manager required an enabling technology to provide high availability persistence instances. Therefore, a thorough assessment of some solutions was conducted, and the final decision was to use an operator containing Custom Resource Definitions, which allows the creation of the necessary databases. The chosen operator was the Percona Operator[21], which automates routine database operations and provides the ability to create a custom internal DBaaS.

YAMLs study and definition

The initial stage of the project involved studying the custom resources in order to comprehend their precise functionality, and creating a custom resource yaml that includes the necessary components for optimal performance. During this phase, three technologies were evaluated: MYSQL, POSTGRESQL, and MONGODB.

MYSQL

In this case, the utilized resource is the PerconaXtraDBCluster, which includes the actual pxc pods with the persistence volume claim, a high availability proxy responsible for distributing incoming requests, and a log collector that consolidates logs from all deployed instances into a single location. Additionally, a custom finalizer was added to delete PVCs when the resource is removed.

Another necessary yaml file is the Secret, which contains the root user's password for the database and other related tools.

POSTGRESQL

In this case, the resource used is the PerconaPGCluster. It consists of several components, including the pgPrimary which is the pod that holds the actual volume, the backup which is scheduled to perform backups at the expected timescales, the pgBouncer which exposes the service to contact the database, and the pgReplicas which specifies the number of high availability replicas.

Similar to the previous case, there is also a Secret that includes the password for the customizable database user and for other components.

MONGODB

In the final case, the resource used is the PerconaServerMongoDB, which includes the ReplicaSet representing the database implementation and a backup pod. Similar to previous cases, a custom finalizer has been added to delete the PVCs when the resource is deleted.

Additionally, a Secret is used, which includes the names of the users and their associated passwords for all the tools used in the process.

The development process related to this thesis project began from a first simple implementation of the tool and started by extending the DTO and DAO structures adding the needed information for the offloading of the database.

YAMLs customization

The development process of this thesis project involved first extending the DTO and DAO structures to include the necessary information for offloading the database. The next step was to create methods to take the information sent by the frontend through the DTO and insert them into the YAML files to customize the resources.

To perform these operations, the Jackson library was used, specifically the databind[10] and the dataformat[11] components for parsing and interacting with the YAML files. The ObjectNode and JsonNode objects were used to navigate the YAML structure and write values to it, while the yamlMapper object was used to map the YAML file to a JsonNode object.

The operations performed for all three cases were similar, with small differences. This phase involved inserting the chosen password (and possibly the username) in the Secret, and adding the name of the database, the number of replicas, the volume size, and binding the previously created Secret in the resource file. Additionally, the correct affinity was added, which will be explained in a later paragraph.

Affinity

In this project, affinity is used to specify rules for scheduling pods. Specifically, nodeAffinity is used to select a node based on a specified label. The process involves searching for the requested label on all available nodes and selecting nodes that have the same or different value of the label from the *value* field, depending on the value of the *operator*.

```
...
spec:
  ...
    affinity:
      advanced:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: liqo.io/type
                    operator: In
                    values:
```

```
13                          - virtual-node
14  ...
```

For this case the affinity needed select the label **liqo.io/type** that is by default present in all the virtual nodes created by the Liqo peering. It has always the value **virtual-node**.

Initially, the Database Manager was tested with only one virtual node representing the publicly hosted machine. However, for the Structura-X project, it was necessary to allow the selection of the virtual node among the peered ones. To achieve this, the presence of the label **kubernetes.io/hostname** was exploited, which has a value starting with **liqo-** and ending with the name of the remote Liqo instance set by the flag *—cluster-name* used in the *liqoctl install* command. This improvement allowed for the selection of the desired virtual node among the peered ones.

So the final section will be:

```
1  ...
2  spec:
3     ...
4       affinity:
5         advanced:
6           nodeAffinity:
7             requiredDuringSchedulingIgnoredDuringExecution:
8               nodeSelectorTerms:
9                 - matchExpressions:
10                    - key: liqo.io/type
11                      operator: In
12                      values:
13                        - virtual-node
14                    - key: kubernetes.io/hostname
15                      operator: In
16                      values:
17                        - liqo-{peered-node}
18 ...
```

To improve the user experience of the latest addition, a list of available virtual nodes has been added to the Structura-X frontend. This is achieved by a method that interacts with the API server to obtain all nodes that have the label **liqo.io/type**. To allow the user to make an informed choice about where to locate the database, labels describing the region, latitude, and longitude of the remote cluster have been added to all virtual nodes.

Chapter 12

Results and evaluation

12.1 A real-world use case: Gaia-X and Structura-X project

Gaia-X[6] is a software framework developed as a European initiative to provide governance and control over cloud/edge technologies. The framework is designed to be deployed on top of any existing cloud platform that follows the Gaia-X standard. This enables the creation of Data Spaces, digital ecosystems that reflect the underlying physical or analog ecosystem by enabling data collection and exchange among multiple organizations.

Currently, proprietary, non-transparent, and non-interoperable technologies limit the exchange of data between federated partners, which affects the necessary level of trust. Gaia-X aims to bridge this trust gap by implementing a new generation of digital platforms that provide transparency, controllability, portability, and interoperability of services.

The architecture of Gaia-X is decentralized, and its implementation is based on a common standard, the Gaia-X standard, followed by multiple individual platforms. The goal is not to compete with existing hyperscalers but to connect data and make it available to a broad audience via open interfaces and standards.

Recently, European cloud providers launched Structura-X, a lighthouse project that meets all Gaia-X requirements and 28 companies and organisations have agreed to make their cloud services Gaia-X compliant. The project aims to create an ecosystem for European data sovereignty by shaping the existing infrastructure services of the partners into a European cloud infrastructure. The infrastructure will enable users to test and deploy their services and data rooms on a Gaia-X-approved infrastructure, allowing for new cross-sector and cross-country collaboration in the cloud and decreasing fragmentation in the European cloud market. Structura-X will enable the necessary scale for new cross-sector and cross-country

collaboration in the cloud, helping to decrease the previous fragmentation of the European cloud market.

12.1.1 The federation

The federation is the entity behind the project and that allows the use case to became a real world implementations that will be consumed by users. The actual federation is composed be CSPs and IXPs. The firsts are the Communications Service Providers, the ones that offers services upon the infrastructure (for example the database offloading on edge), while the others are the Internet Exchange Points the is the one which provides the infrastructure and the connection layer to the CSPs.

The picture below describes the involved entities: Engineering, Polytechnic of Turin, Aruba and Ionos as CSPs and Topix and DE-CIX as IXPs. The graphic shows how they interact among each other and the fact that this project collects partners belonging to different domains, from service providers to academic institutions and that is still expanding.

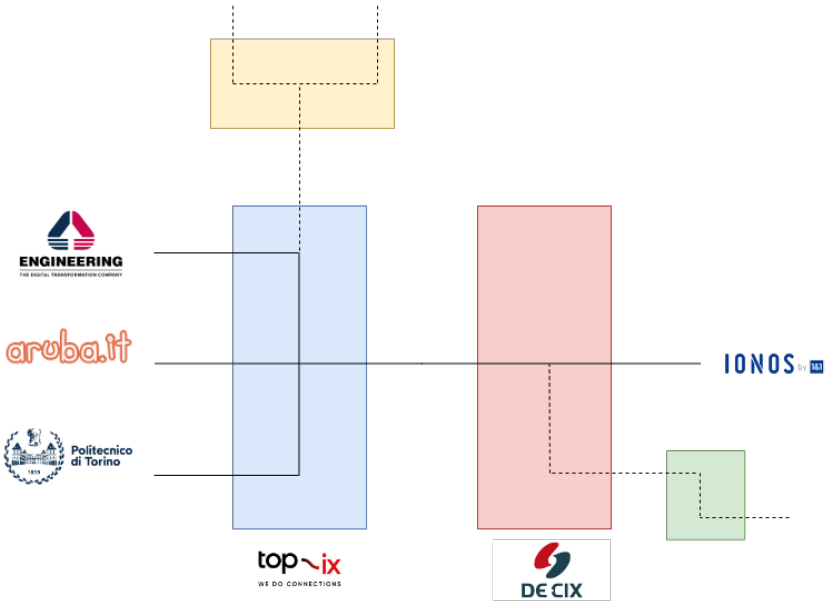

Figure 12.1: Project partners

12.2 Issues emerged and solved

In this section, the main challenges that emerged during the project's development will be discussed, along with the solutions that were adopted to overcome them. The work carried out in this thesis contributed to Liqo's evolution as it uncovered several issues, some of which led to permanent fixes now available in Liqo version 0.6.0.

12.2.1 Accessibility of API Server

The initial challenge encountered during the project was the accessibility of the API server of the Openshift cluster, which is attributed to the network configuration and the load balancer positioned in front of the cluster. The process of peering through Liqo requires the host executing the command to communicate with the API servers of all parties involved, which was not possible due to the presence of the load balancer. The first approach was to manually modify the resources created by Liqo to establish the correct IP addresses. However, this problem is now resolved in Liqo version 0.6.0 with the introduction of the option *--set apiServer.address=* that allows users to customize the exposed addresses.

12.2.2 Fedora CoreOS interfaces management

Another issue that emerged during the project and caused several problems is related to the behavior of Fedora CoreOS with newly created interfaces inside a network namespace that is different from the host namespace. The diagram below illustrates all the interfaces that are present in the liqo-gateway pod.

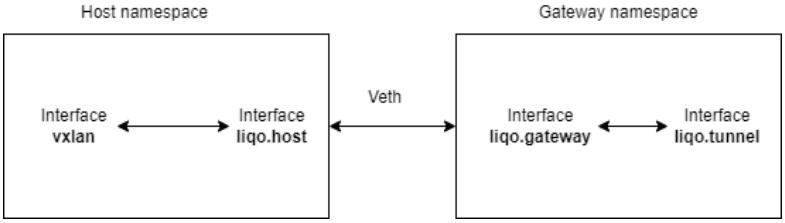

Figure 12.2: Liqo-gateway network namespaces and interfaces

The behavior of Fedora CoreOS with newly created interfaces inside a network namespace different from the host one caused a significant problem during various steps of the project. Specifically, the issue arose when creating the **liqo.tunnel** interface, which exists in the newly created namespace. The operating system created the interface once, deleted it, and then immediately recreated it, causing

issues with the configuration of the two interfaces in the gateway namespace. When the interface is initially created, Liqo takes the MAC address of the interface and creates a permanent entry in the APR table of the namespace. This operation enables communication in standard scenarios, but since the second interface, the one that remains, has a different MAC address from the first one, the ARP table becomes incorrect. To solve this problem, the new version of Liqo includes a handler that watches the newly created interface. If the MAC address changes, the handler modifies the ARP table to keep everything aligned and maintain correct configuration.

12.2.3 Openshift privileges management

The third issue pertains to a unique feature of Openshift that differs from the standard Kubernetes in terms of permission management. Specifically, the Red Hat distribution operates differently with regard to permissions related to pods that lack a higher-level abstraction (e.g., Deployments) and are not associated with a properly privileged service account. To resolve this issue, the virtual kubelet in the Openshift environment was updated in the v0.6.0 of Liqo to include the "anyuid" SCC (security context constraints), which is automatically associated with pods created by cluster administrators by default.

12.2.4 Customize type of service depending on the cluster configuration

The final issue concerns the services that Liqo exposes for reflecting objects and managing the offloading process. These services, including the custom **liqo-gateway** and **liqo-auth**, are contacted and exposed by Liqo during peering, in addition to the standard API server. The last one is in charge of the mutual authentication of the two involved clusters while the first one come into play for all the other connections. By default, these services are created as **LoadBalancer** type, which can lead to errors during the Liqo initialization phase, as the installer waits indefinitely for the service to come up. To solve this, the Liqo installer now allows customization of the service type and address during installation using the following flags: *—set auth.service.type=NodePort —set gateway.service.type=NodePort —set auth.config.addressOverride= —set gateway.config.addressOverride=*

12.3 Split-brain scenario evaluation

The use case described requires resistance to split-brain scenarios, where the connection between the two clusters is lost. During such a scenario, it is essential

to ensure that data is not lost and remains undamaged. Additionally, the data should be accessible inside the remote cluster even during downtime.

In this situation, Liqo maintains a behavior that meets the requirements of the use case. In fact, in case of loss of connection between the peers, each peer remains partially independent, that is, it maintains the state of its resources to the current one (for example, by rescheduling pods if necessary), but it does not allow manual modification of the edge cluster (in terms of scaling horizontally a deployment for example), which is not considered necessary for the use case unless there are particular situations and prolonged connection problems over time.

The evaluation involved setting up a standard scenario with an offloaded database. Then, one of the cluster's interfaces towards the internet was intentionally turned off. Tests were then performed from both clusters to reach the database, and as expected, the cluster where the database was deployed successfully reached it while the other cluster could not establish a connection.

Next, some pods of the database were restarted and everything worked smoothly. Successful connection tests further confirmed that the independence in the pod life cycle of the two clusters is real.

Finally, the interface was restarted and everything returned to the previous situation. The scenario used for these tests is described in the diagram below.

12.4 Inter-cluster traffic evaluation

The objective of the latest evaluation was to quantify the data transferred between the two clusters to estimate the potential costs in terms of internet traffic.

The test setup comprised an Openshift cluster within the Engineering network and a self-hosted K3s distribution, and the in-band peering mode was used.

The test was initiated from the start, beginning with the establishment of peering between the two clusters, with the peering being unidirectional from the Openshift cluster to K3s. A namespace was then offloaded, followed by the deployment of a database from the Openshift cluster to the virtual node representing the K3s instance. The traffic exchanged was then monitored for several hours to evaluate the situation.

The results are summarized in the following table

Phase	Direction	Traffic (KB)	Total traffic per phase (KB)
Peering	Openshift->K3s	109,5	190.6
Peering	K3s->Openshift	81,1	
Offloading	Openshift->K3s	3504,5	9078,1
Offloading	K3s->Openshift	5573,6	
1° hour	Openshift->K3s	19251,2	46899,2
1° hour	K3s->Openshift	27648	
2° hour	Openshift->K3s	19763,2	47616
2° hour	K3s->Openshift	27852,8	
3° hour	Openshift->K3s	18841,6	45875,2
3° hour	K3s->Openshift	27033,6	
4° hour	Openshift->K3s	19660,8	47411,2
4° hour	K3s->Openshift	27750,4	
Total		197070,3 (192,5 MB)	

In conclusion, the test results show that the total amount of traffic generated by Liqo is relatively low, with an average of 0.5 MB per hour. This low amount of traffic is a significant advantage of Liqo, as each cluster has independent resource management, and only the status of offloaded pods needs to be transmitted.

To better represent and understand the collected data two graphs are reported here.

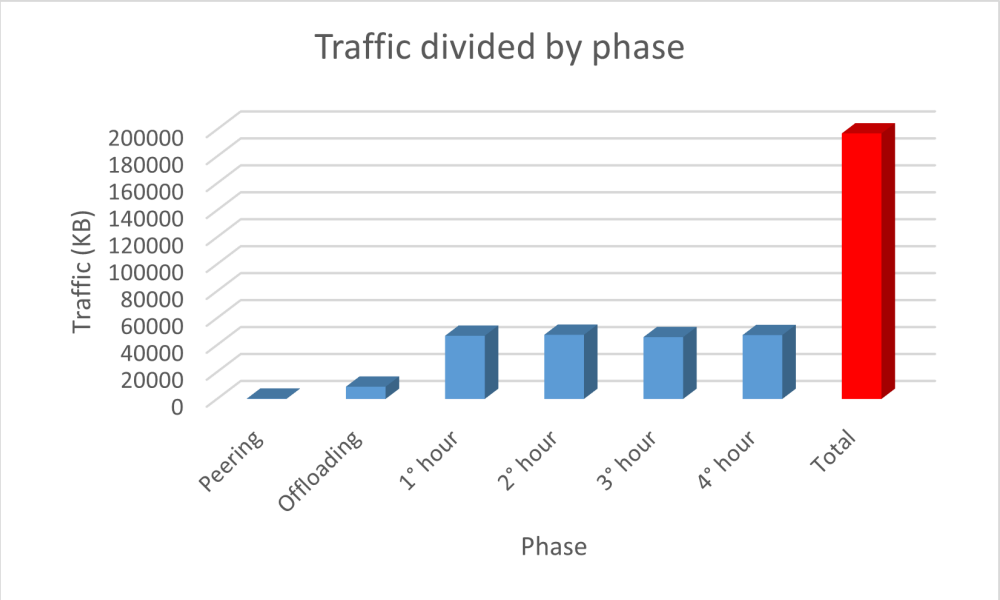

Figure 12.3: Inter-cluster traffic compared by phase

The first one describes the amount of traffic divided by phase. The main consideration is that the peering and offloading phases, which could have been considered the heaviest, have a marginal impact.

In particular the peering phase is very light from the traffic point of view and the only packets that are transmitted are the authentication ones and the ones used to build the VPN tunnel.

The offloading phase is slightly more traffic consuming because it includes the offloading of the namespace and then the communication regarding the resources that must be deployed remotely.

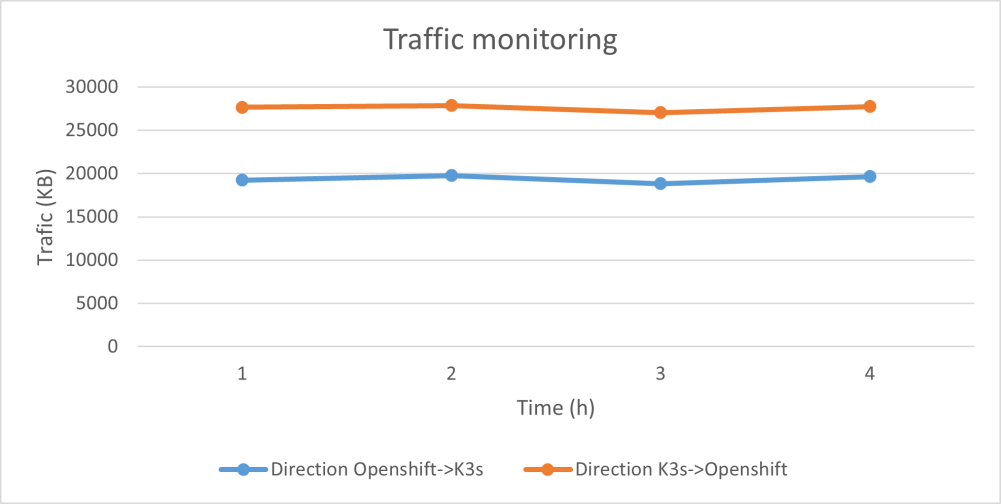

Figure 12.4: Inter-cluster traffic monitoring over time

The first report presents a breakdown of the traffic amount based on different phases. The main observation is that the phases that might have been expected to generate the highest amount of traffic, namely the peering and offloading phases, actually have a relatively minor impact.

Specifically, the peering phase is very lightweight in terms of traffic, with only authentication packets and those needed to set up the VPN tunnel being transmitted.

The offloading phase is somewhat more traffic-intensive due to the offloading of the namespace and the communication required to deploy resources remotely.

The final graph depicts the traffic during normal monitoring and it shows a relatively constant rate of 20-30 MB in both directions. It is worth noting that the line representing the traffic from K3s to Openshift is higher, as the resource is offloaded from Openshift to K3s and the latter cluster needs to send back the information on the status of the pods.

Part VI

Conclusion and future developments

Conclusions

The work conducted in this thesis has resulted in a practical and effective solution for real-world implementations. The evaluations and tests performed have demonstrated the feasibility and effectiveness of the proposed approach.

In particular, the project has been extensively tested through its usage in the Structura-X project, and being part of the Engineering catalog confirms its readiness for production use. The proof-of-concept built for the Gaia-X initiative has demonstrated the possibility of connecting multiple clusters to create a federation of CSPs from different parts of the world at reasonable costs and within reasonable timeframes.

The simulations conducted on the proposed solution revealed two significant aspects that are relevant for companies intending to adopt this idea in production environments.

The first aspect is the behavior in a split-brain scenario, where connectivity issues occur between the clusters. The test results showed that the project's operation is not affected by this potential event and guarantees continuity of service to the users in every situation.

The other crucial outcome is the small amount of traffic exchanged between the clusters, making the solution accessible to everyone.

In the near future, the main area of application for this project will be the medical field, where personal data issues are common and a solution is needed to extend cloud capabilities to multiple tools. The project's capabilities are remarkable, ranging from remote control, deployment, and management of software leveraging the abstraction of a single cluster, to data storage and remote processing, which can accelerate and improve efficiency in medical procedures.

Future developments

Two potential future improvements have been identified for the proposed solution. The first is the implementation of remote processing, which was discussed as a theoretical concept in this thesis. This could be accomplished by building upon the existing technology and making necessary customizations based on the specific requirements of the processing task.

The second possible improvement is the integration of remote processing with database offloading. This would enable automatic storage of personal data that cannot be transferred to the cloud in the local environment, while anonymizing other data through an AI flow before transferring it to the cloud.

Bibliography

[1] Admiralty official documentation. https://admiralty.io/docs/. (Cited on page 65.)

[2] Azure kubernetes service official documentation. https://learn.microsoft.com/en-us/azure/aks/. (Cited on page 32.)

[3] Digitalenabler. https://www.eng.it/en/our-platforms-solutions/digital-enabler. (Cited on page 35.)

[4] Amazon eks official documentation. https://aws.amazon.com/eks/. (Cited on page 31.)

[5] Engineering ingegneria informatica. https://www.eng.it/. (Cited on page 2.)

[6] Gaia-x official website. https://gaia-x.eu/. (Cited on pages 2 and 93.)

[7] What is gdpr, the eu's new data protection law? https://gdpr.eu/what-is-gdpr. (Cited on page 44.)

[8] Amazon eks official documentation. https://cloud.google.com/kubernetes-engine. (Cited on page 32.)

[9] M. Iorio, F. Risso, A. Palesandro, L. Camiciotti, and A. Manzalini. *Computing Without Borders: The Way Towards Liquid Computing*. 2022. (Cited on page 63.)

[10] Jackson databind official repository. https://github.com/FasterXML/jackson-databind. (Cited on page 91.)

[11] Jackson dataformat yaml officail repository. https://github.com/FasterXML/jackson-dataformat-yaml. (Cited on page 91.)

[12] K3s official documentation. https://k3s.io/. (Cited on pages 3 and 31.)

[13] Kind official documentation. https://kind.sigs.k8s.io/. (Cited on pages 3 and 31.)

[14] Kudeege official documentation. https://kubeedge.io/en/docs/. (Cited on page 68.)

[15] Kubernetes official documentation. https://kubernetes.io/docs/home/. (Cited on page 18.)

[16] Liqo peering documentation. https://docs.liqo.io/en/v0.6.0/features/peering.html. (Cited on pages 7 and 89.)

[17] Liqo official documentation. https://docs.liqo.io/en/v0.5.0/. (Cited on page 63.)

[18] microk8s official documentation. https://microk8s.io/docs. (Cited on pages 3 and 31.)

[19] Openshift official documentation. https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10. (Cited on page 29.)

[20] Understanding the difference between kubernetes vs. openshift. https://www.simplilearn.com/kubernetes-vs-openshift-article. (Cited on page 29.)

[21] Percona kubernetes operator documentation. https://www.percona.com/software/percona-kubernetes-operators. (Cited on page 90.)

[22] Rancher official documentation. https://www.rancher.com/. (Cited on page 30.)

[23] Structura-x presentation. https://gaia-x.eu/news/latest-news/structura-x-lighthouse-project-for-european-cloud-infrastructure-is-launched-concrete-implementation-and-alignment-with-the-gaia-x-roadmap-of-compatible-services/. (Cited on page 2.)

[24] tensile-kube official repository. https://github.com/virtual-kubelet/tensile-kube. (Cited on page 79.)

[25] Thingsboard official website. https://thingsboard.io/. (Cited on page 34.)

[26] United nations conference on trade and development. https://unctad.org/page/data-protection-and-privacy-legislation-worldwide. (Cited on pages 7 and 41.)

[27] Virtual kubelet. https://github.com/virtual-kubelet/virtual-kubelet. (Cited on page 74.)