Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

A Policy-Driven Kubernetes-based Architecture for Resource Management in Multi-Cloud Environments

Supervisor

Prof. Sandro Luigi Fiore

Student

Leonardo Vicentini

Co-supervisors

Dott. Diego Braga

Dott. Francesco Lumpp

Academic year 2023/2024

Acknowledgements

Thanks to my Family and Friends for the support and encouragement throughout the years.

Contents

List of Figures

Abstract

contesto
- motivazioni
- riassunto problema affrontato
- tecniche utilizzate analisi requisiti, analisi pprogetti/prodotti disponibili creazione proof of concept
- risultati raggiunti: e2e testing
- contributo personale

1 Introduction

[Intro]

The work described in this thesis is part of a larger project that aims to develop a system ... The project is divided into three main parts: data analysis, machine learning, and cloud infrastructure as shown in Figure **??**.

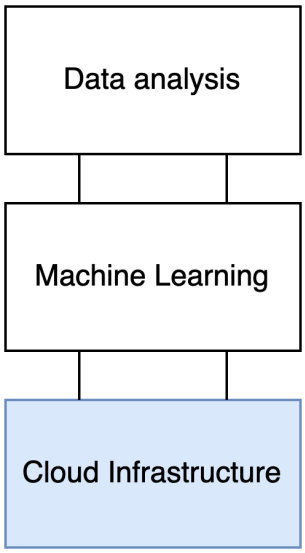

Figure 1.1: Project parts

1.1 Context

Computational sustainability

GreenOps

GreenOps for FinOps (Operating for GreenOps may lead to reduced costs)

1.1.1 GreenOps

1.1.2 Geographical shifting and Time shifting

1.1.3 Carbon-aware workload scheduling

Cloud sustainability

Current Sustainable Cloud Computing Landscape we are in the infrastructure tooling section in particular scheduling (day 1 operatsion)

scaling and resource tuning are usuaully day 2 operation

the system was envisioned with this in mind and is capable of doing that

1.2 Problem statement

test

Use cases (basic ones for the beginning) higher level explanation here first use case (”GreenOps” VM scheduling)

second: scaling down a vm infrastructure already put in place

the system was designed with flexibility in mind therefore a workload could be potentially anything the condition is just to be represented in some way and have something else do certain actions based

on that representation As we will see in section XXX, the most simple of this would be K8s operators this is described in section XYZ

1.3 Method

Developing a real solution, integrating it on top of OSS

production-ready solution

we are on the consumer side, not on the provider side

System architecture to start with: Saima's + Krateo platform

integration into an existing platform (krateo)

leveraging krateo componenets

Krateo Core Provider and cdc instead of developing 1 or more K8s operators from scratch

analysis of possible solutions implemnted poc

Initial analysis of a solution with operators were tried

A PoC comprising 1 operator was created "Synchronization operation" cons: maintainer costs

ideation and creation of architectural diagrams

tackling first use case but create a system that is flexible enough to be used for other use cases as well

1.4 Personal contribution

The project, ideated and supervisioned by Prof. Fiore is mainly divided into 3 parts.

exploratory data analysis data preparation

model training model selection

infrastructure part

GOAL The goal of the project is to employ mainly time-shifting and geographical shifting for the scheduling of workload leveraging a multi-cloud setting. We can choose to schedule workloads in periods and regions with low carbon intensity (when renewables are plentiful). Therefore, targeted workloads are the ones that are not time-sensitive but instead are quite delay-tolerant. For example training a machine learning model could wait until a period of low carbon intensity. Another example is shifting video / image processing, as Google is doing. Kubernetes is leveraged as a platform for scheduling and managing workloads on different cloud providers. Long term goal: "Using electricity when the carbon intensity is low is the best way to ensure investment flows towards low-carbon emitting plants and away from high-carbon emitting plants".

2 Background

2.1 Public cloud providers

what are they, what is this deployment model which are the 3 main players what is an hyperscaler (AWS, Azure, GCP)

2.1.1 Regions and zones

cloud regions regions vs availability zones Cloud providers usually further divide region into ... Each Region supports a subset of the available instance types. We could safely assume that our workload specs are quite standard and therefore can be scheduled on any cloud region.

each provider has a different number of regions and zones and different naming conventions there is no standardization in the industry for this. for instance a region located in london is called eu-west-2 in AWS, uk-west in Azure and europe-west2 in GCP

[table how many regions per cloud provider]

Figure **??** shows the geo-distribution of Azure cloud regions (data centers) and the country Grid Carbon Intensity (year 2024) as reported by Electricity Maps [**?**]. Data center coordinates were retrieved by an Azure CLI command dump [**?**].

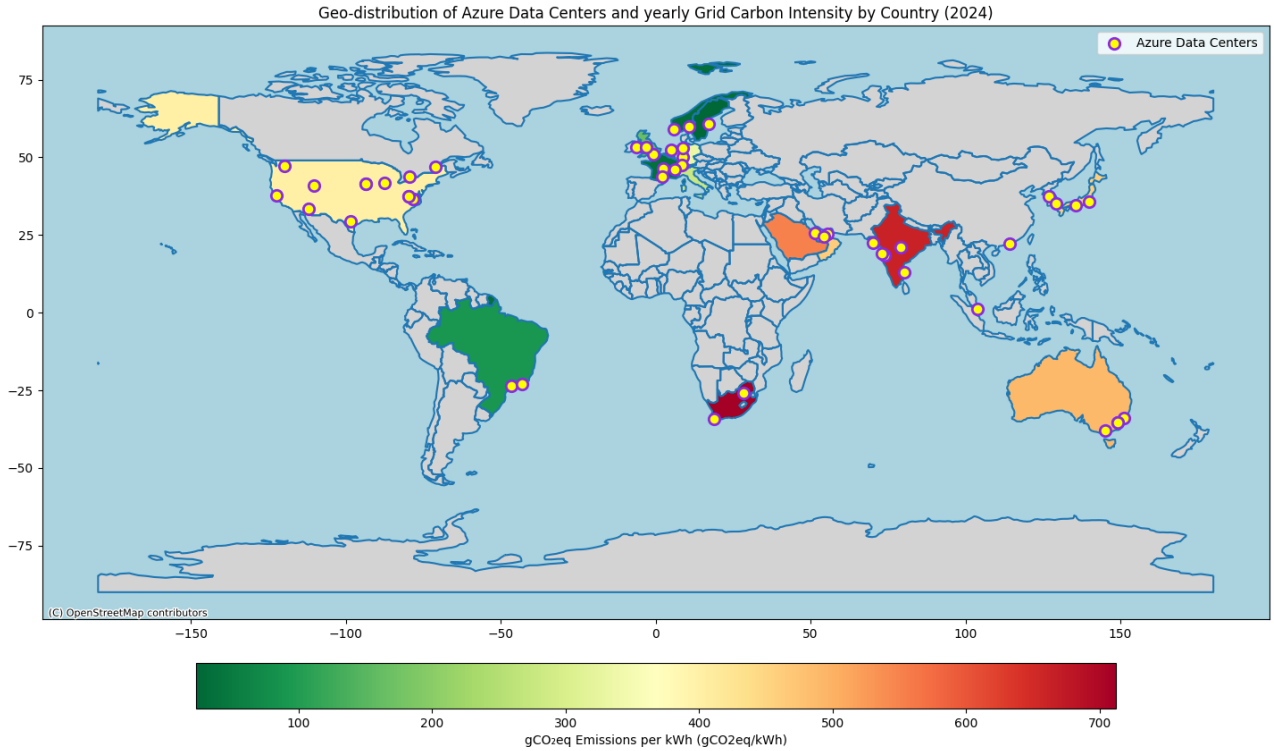

Figure 2.1: Geo-distribution of Azure data centers with country Grid Carbon Intensity

2.1.2 Computational Sustainability by Public cloud providers

what are they already doing

—

We assume that a cloud data center will likely rely on the same energy sources that characterize a specific geographical region (grid). For example, if data from Electricity Maps tell us that Finland is producing energy with low carbon emissions then we assume that the data centers in that area will likely be powered with energy from low carbon sources. However, some cloud providers may have better access to renewable energy sources in certain regions due to their individual initiatives e.g. wind farms that feed directly into their data centers.

what microsoft is already doing with alternative energy sources apart from grid

—

https://blog.google/inside-google/infrastructure/data-centers-work-harder-sun-shines-wind-blows/

Google CFE%: "This is the average percentage of carbon free energy consumed in a particular location on an hourly basis, while taking into account the investments we have made in carbon-free energy in that location. This means that in addition to the carbon free energy that's already supplied by the grid, we have added carbon-free energy generation in that location".

2.1.3 Multi cloud paradigm

what is multi cloud paradigm

why is useful how to achieve

advantages of multi-cloud flexibility risk reduction (reduces vendor lock-in)

why it is important for the organizations

disadvantages increased complexity attack surface increased (security)

Why multi-cloud in the context of our system? Different cloud providers have data centers in various locations around the world. This diversity allows for more options when geographically shifting workloads to regions with lower carbon intensity. However the 3 big players have a big overlap: each one is present almost everywhere. Multi-cloud paradigm could be leveraged for lowering costs. For basic use cases, we can even set a single cloud provider to be used (e.g., Azure) and therefore just a multi-region environment. Being able to work in a multi-cloud environment is also important for accomplishing user / company needs: they can use just a single cloud provider or more than one for different reasons. Therefore if our system supports more cloud providers, it will accomplish more users' needs. If the system is designed to be multi-cloud then flexibility is higher for organizations and users. For the purpose of this work, we will consider only the 3 major Public Cloud Provider as of today: AWS, Azure, GCP

2.2 Kubernetes

Kubernetes is an open-source platform for automating the orchestration of containerized applications. It is widely used in the industry and became the de-facto standard for container orchestration. An extensive description of Kubernetes is out of the scope of this thesis but it is deemed necessary to provide a brief overview of the main concepts of Kubernetes that are relevant for the purpose of this work. Figure **??** shows the Kubernetes architecture as described by the Cloud Native Computing Foundation (CNCF) [**?**].

The main components of Kubernetes are:

- **Kubernetes API server**: the central component that manages the Kubernetes cluster. It exposes the Kubernetes API and is responsible for validating and mutating data for the Kubernetes objects.
- **etcd**: a key-value store used to store the cluster state.
- **Kubernetes controller manager**: a daemon that embeds the core control loops shipped with Kubernetes.
- **Kubernetes scheduler**: a component that assigns Pods to nodes.
- **Kubelet**: an agent that runs on each node in the cluster. It makes sure that containers are correctly running in a Pod.
- **Kubernetes proxy**: a network proxy that runs on each node in the cluster. It maintains network rules and it is responsible for routing network traffic.

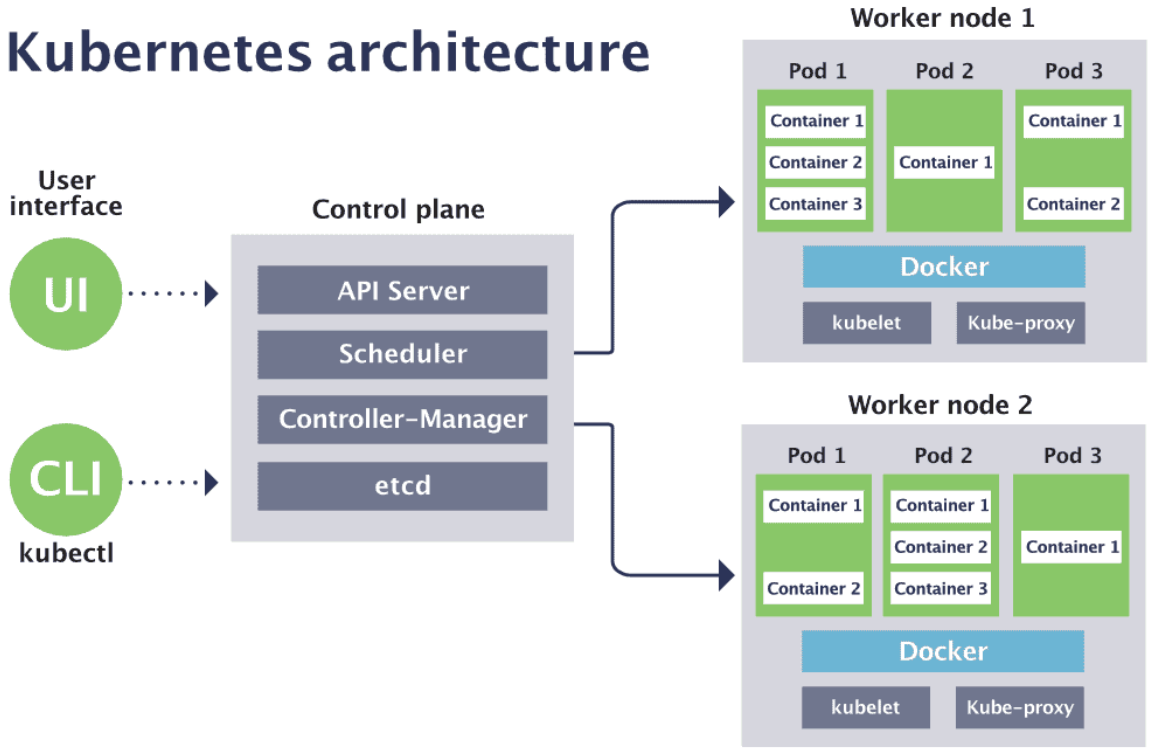

Figure 2.2: Kubernetes architecture by CNCF [**?**]

- **Container runtime**: the software that is responsible for running containers on the Node (in this case represented by Docker).

[TO BE FIXED] It must be noted that in our system we are not extending the Kubernetes scheduler as described for instance in the work of .. since we are not dealing with the scheduling of the in-cluster resources (e.g., Kubernetes Pods). We are instead focusing on the management of external resources on cloud providers. As described in the following section, our focus will be on the Kubernetes API sever and in particular on Admission Control

2.2.1 Kubernetes as a platform

[TO BE FIXED] Kubernetes as a platform to manage external resources representing resources as kubernetes objects to leverage the powerful kubernetes API and tooling

This concept is widely used (cloud provider operators for example)

Many cloud-native development teams work with a mix of configuration systems, APIs, and tools to manage their infrastructure. This mix is often difficult to understand, leading to reduced velocity and expensive mistakes. Config Connector provides a method to configure many Google Cloud services and resources using Kubernetes tooling and APIs.

2.2.2 Kubernetes extendability

[TO BE FIXED]

Kubernetes allows for the extension of its functionalities through the use of **Custom Resource Definitions (CRDs)** and **Kubernetes Operators**, effectively adopting the so-called **Operator paradigm**. Simply put, Custom Resource Definitions are a way to instruct Kubernetes to manage a new resource type. They are a schema that defines the structure of the resource and the Kubernetes API server will validate the resource against the schema. **Custom Resources (CRs)** are actually instances of the resources defined by CRDs and are managed by an Operator. The Operator is a piece of software that is responsible for managing the lifecycle of the resources defined by the CRD.

Effectively the code of an Operator is usually deployed on the Kubernetes cluster in the form of a Deployment. This concept is not different of what actually happens for standard built-in Kubernetes resources which are however manged by built-in controllers. An high-level overview of the Operator paradigm is depicted in Figure **??**.

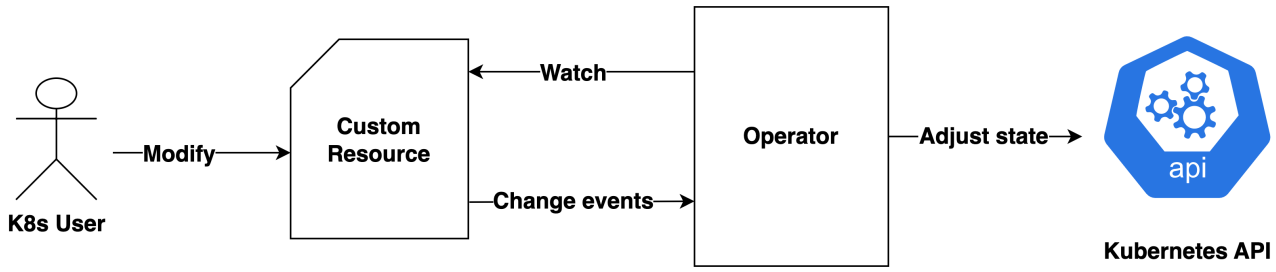

Figure 2.3: Operator paradigm

2.2.3 Helm

Helm is a **package manager** for Kubernetes. Therefore with Helm it is possible to define, install and manage Kubernetes applications in a simpler way compared to a manual management of Kubernetes resources manifests [**?**]. Helm is a graduated project in the CNCF and it is the de-facto standard for Kubernetes package management. The key concept is the **Helm chart**, which is a collection of files that describe a related set of Kubernetes resources. These files are mainly of two types: templates and values. The **templates** are Kubernetes manifest files that are rendered by Helm's **powerful templating engine**. The **values** are the set of variables that are used to render the templates. Upon an Helm chart installation, Helm will render the templates "injecting" the values and deploy the resources in the Kubernetes cluster. One major advantage that Helm provides is the complete management of the lifecycle of the resources. As a matter of fact, Helm allows to easily **upgrade**, **rollback** and **uninstall** the Kubernetes resources deployed with a Helm chart reducing time and errors in such operations [**?**]. Without Helm, the user would have to deal with each single Kubernetes resource manifest file and manually apply changes to them. Finally, users can benefit of Helm charts already developed by the community and leverage chart distribution within their organization using Helm repositories.

2.3 Krateo PlatformOps

Krateo PlatformOps (Krateo) is an **open-source Kubernetes-based platform** that aims to provide a unified interface for managing any desired resource on any infrastructure [**?**]. Krateo runs as a Kubernetes deployment inside a Kubernetes cluster but **acts as a control plane** even for resource external to the Kubernetes cluster. The only requirement for this management is that the resources need to be logically descriptible using a YAML file which represents the desired state of the resources [**?**]. Krateo is composed of three main parts:

- Krateo Composable Operations
- Krateo Composable Portal
- Krateo Composable FinOps

For the purpose of this work, we will focus on the **Krateo Composable Operations** part, which is the core of the Krateo platform and is responsible for managing the lifecycle of resources in a Kubernetes cluster [**?**]. Krateo Composable Operations is composed in turn by several components. Due to their core importance in our system, as described in section XYZ we will briefly describe the functionalities of the **Krateo core-provider** and the **Krateo composition-dynamic-controller**.

2.3.1 Krateo core-provider

The Krateo core-provider is a Kubernetes operator that has the duty of downloading and managing Helm charts. It first checks for the existence of a file named *values.schema.json* in the chart folder and uses it to generate a Kubernetes Custom Resource Definition (CRD), accurately representing the

possible values that can be expressed for the installation of the chart [**?**]. The file *values.schema.json* is a JSON schema that describes the structure of the *values.yaml* file for the related Helm chart and it is considered a standard best practice for Helm charts. It basically provides a way to validate the *values*.yaml file before the Helm chart is installed (i.e., to check if the values are in the correct format). In other words, the Krateo core-provider operator is responsible for deploying the Helm chart as a **native Kubernetes resource**, which allows for the management of the Helm chart lifecycle through Kubernetes APIs [**?**]. As a matter of fact, out of the box, Kubernetes does not provide a way to manage Helm charts natively and the Krateo core-provider is one tool that allows to do so. The Kubernetes Custom Resource Definition introduced by the Krateo core-provider is called ***CompositionDefinition***. It is a CRD that represents the Helm chart and its values (a Helm Chart archive (.tgz) with a JSON Schema for the values.yaml file) [**?**]. Upon a CompositionDefinition manifest application to the Kubernetes cluster, the Krateo core-provider generates the CRD from the schema defined in the *values.schema.json* file included in the chart. It then deploys an instance of the Krateo composition-dynamic-controller, setting it up to manage resources of the type defined by the CRD [**?**].

2.3.2 Krateo composition-dynamic-controller

The Krateo composition-dynamic-controller is the Kubernetes operator that is instantiated by the Krateo core-provider to manage the Custom Resources whose Custom Resource Definition is generated by the core-provider. In practice, when a Custom Resource (CR) is created, the instance of composition-dynamic-controller checks if a Helm release associated with the CR already exists in the cluster [**?**]. If this is not the case, it performs an *helm install* operation using the values specified in the CR to create a new Helm release. This will practically deploy all the resources defined in the Helm chart using **Helm's templating engine**. However, if the Helm release does already exist, it instead executes an *helm upgrade* operation, updating the release's values with those specified in the CR, effectively updating the resources in the cluster. Finally, when the CR is deleted from the cluster, the instance of the composition-dynamic-controller performs an *helm uninstall* on the release, removing all the resources defined in the Helm chart from the cluster [**?**].

The architecture of the Krateo core-provider and composition-dynamic-controller is depicted in Figure **??**.

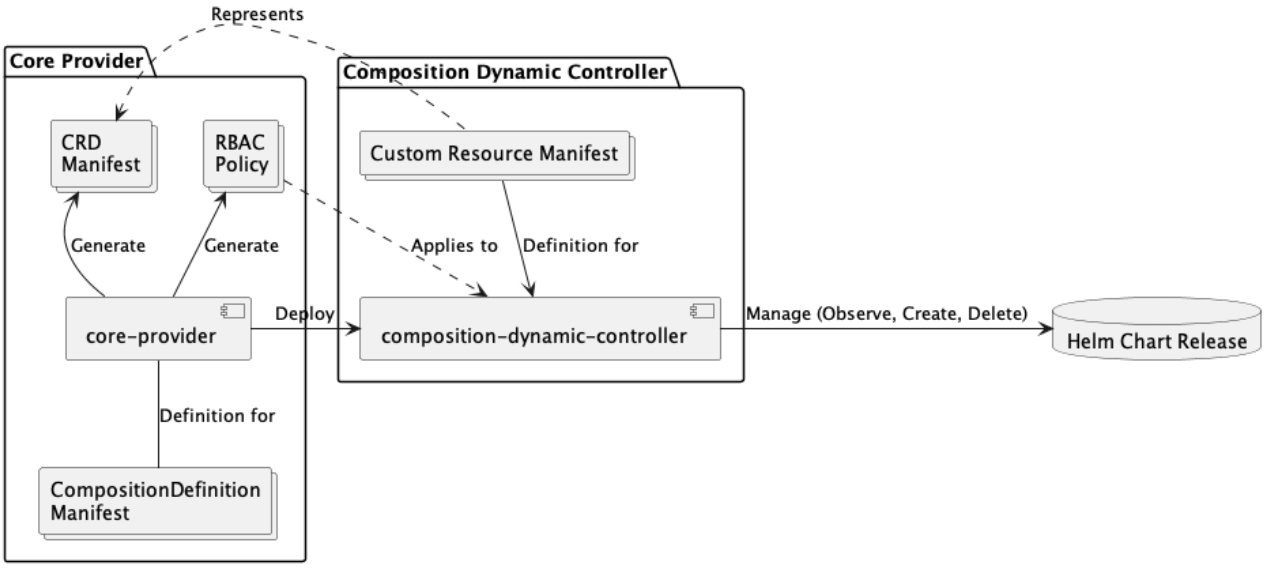

Figure 2.4: Krateo core-provider and composition-dynamic-controller architecture [**?**]

2.4 Multi cloud resource management

The idea of a **dynamic management of workloads leveraging a multi-cloud paradigm** is not new. In this section we will provide an overview of some of the existing works in the literature that have tackled the problem of multi-cloud resource management.

2.4.1 Dynamic Virtual Machine placement

The work of Simarro et al. [**?**] back at the dawn of cloud computing (2011) proposed a multi-cloud architecture for the dynamic placement of Virtual Machines (VMs). The main objective of the system was cost optimization but this paradigm provides reliability and flexibility as well. The scheduling part is comprised of a "**cloud broker**" that is responsible for VM placement **transparent to users** providing a single uniform interface to the cloud resources. Users can provide to the system a "**service description template**" to specify the number of VMs to provision and some constraints. The cloud broker architecture is composed of two major components: the **scheduler** and the **cloud manager** [**?**]. The former is responsible for placement decision across multiple cloud providers, while the latter is responsible for the actual management of the VMs in the cloud providers. More precisely, the cloud manager is represented by the OpenNebula (ONE) virtual infrastructure manager [**?**]. OpenNebula is an open-source platform that aims to provide a unified management interface for multiple virtualization technologies and cloud providers [**?**].

2.4.2 Cloud service brokers

A CSB is a system that acts as an **intermediary** between cloud service providers and consumers, providing a **unified interface** to manage cloud resources across multiple providers [**?**]. Cloud service brokers (CSBs) were described and categorized by Wadhwa et al. [**?**] in their work of 2013. The emerging market of cloud computing led to the proliferation of cloud services and providers, and by consequence the need for mechanisms to manage costs, capacity and resources [**?**].

An interesting CSB example in the literature is the **STRATOS** system by Pawluk et al. proposed in 2012. The work can be considered a pioneer in the field of multi-cloud resource management since it can be framed in the first years of cloud computing [**?**] but the proposed paradigms and concepts are relevant today. STRATOS tried to **avoid the assumption of resource homogeneity** and represented an initial attempt to provide a "**cross-cloud resource provisioning**" system [**?**]. The proposed architecture enables the specification of high-level objectives that can be assessed in a standardized manner across different providers. The decision-making process is fully automated, shifting the decision point from deployment to runtime [**?**]. Users first submit a topology document, triggering the Cloud Manager to communicate with the Broker for topology instantiation. The Broker (implemented in Java) then conducts the initial resource acquisition decision (RAD), optimizing the allocation of resources across multiple providers (configured beforehand) [**?**]. Experiments indicate that distributing workloads across different cloud providers can reduce the overall cost of the topology. The approach taken by the authors primarily focuses on two objectives: **cost efficiency** and **avoiding vendor lock-in**. The application environment was deployed on public cloud platforms, specifically AWS and Rackspace [**?**].

2.4.3 AI-based resource management in cloud computing

More recent works have focused on the development of systems that leverage AI techniques for the optimization of resource management in cloud computing environments.

[Machine learning (ML)-centric resource management in cloud computing: A review and future directions]

[HUNTER: AI based holistic resource management for sustainable cloud computing]

2.4.4 Policy-driven resource management systems

[SLA-driven dynamic cloud resource management]

[Deadline-aware Dynamic Resource Management in Serverless Computing Environments]

[Policy-Based Cloud Management Through Resource Usage Prediction]

[Amazon Web Services Cloud Compliance Automation with Open Policy Agent]

—

2.5 GreenOps landscape

what is greenops

greenops landscape

In the context of cloud-native sustainability, the Technical Advisory Group (TAG) Environmental Sustainability is a XYZ that supports and advocates for environmental sustainability initiatives in cloud native technologies.

2.5.1 Green Software foundation

green software foundation very important actor in the field of green software

sci: https://sci.greensoftware.foundation/

proposed a standard for data like the FOCUS standard available trying to push a specification similar to what focus is for FinOps as per 2025 this standard is not yet adopted by cloud providers "real time cloud" proposed standard (not yet adopted): https://github.com/Green-Software-Foundation/real-time-cloud?tab=readme-ov-file

the foundation also developed the Impact Framework which will be described in section XY could be potentially integrated in our system

https://patterns.greensoftware.foundation/catalog/cloud/choose-region-closest-to-users [which patterns are used in our system]

2.6 Carbon-aware systems for resource management

Having provided an overview of the existing works in the field of multi-cloud resource management and having introduced the GreenOps landscape, we now focus on the state of the art in the field of carbon-aware systems for resource management.

The work of 2023 by Sukprasert et al. named "*Spatiotemporal Carbon-aware Scheduling in the Cloud: Limits and Benefits*" [**?**] is a comprehensive analysis on the limits and benefits of the employment of geographical shifting and time shifting for cloud workloads. The authors highlight the fact that different workloads have different characteristics and therefore different degrees of flexibility. Those include, for instance: **execution deadlines**, **data protection laws**, and **latency requirements**. Therefore, carbon savings are constrained by a complex set of factors that need to be taken into account when designing a carbon-aware system [**?**].

2.6.1 CASPER

CASPER (Carbon-Aware Scheduling and Provisioning for Distributed Web Services) is a carbon-aware scheduling and provisioning system whose primary purpose is to minimize the carbon footprint of distributed web services [**?**]. The system is defined as a multi-objective optimization problem that considers two factors: the **variable carbon intensity** and the **latency constraints** of the network [**?**]. By evaluating the framework in real-world scenarios, the authors demonstrate that CASPER achieves significant reductions in carbon emissions (up to 70%) while meeting application **Service Level Objectives (SLOs)**, highlighting its potential for practical implementation in large-scale distributed systems [**?**]. The authors of CASPER highlight the importance of considering the workload characteristics such as memory state, **latency** and **regulatory constraints such as GDPR** [**?**]. The system is not adopting time-shifting since it is dealing with web services that are by their nature non-stopping workloads. The architecture is tied to scheduling K8s resources inside K8s clusters and does not consider external resource management.

2.6.2 CASPIAN

[TO BE ADDED]

2.6.3 Let'sWaitAwhile

[TO BE ADDED]

2.6.4 CarbonScaler

[TO BE ADDED]

https://dl.acm.org/doi/abs/10.1145/3626788 CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency

2.6.5 Microsoft's Carbon-Aware Kubernetes

[TO BE ADDED] [https://devblogs.microsoft.com/sustainable-software/carbon-aware-kubernetes/]

2.6.6 A Low Carbon Kubernetes Scheduler

[TO BE ADDED] https://ceur-ws.org/Vol-2382/ICT4S2019$_p aper_2$8.*pdf*

It provisions entire K8s clusters on the selected regions. Interesting feature: they use local air temperature and solar irradiance as tiebreaker for 2 datacenters with similar carbon intense grid. The claim is that: "Solar irradiance varies more widely than carbon intensity across global regions" and that "Local air temperature surrounding a datacentre affects the amount of energy needed for cooling".

—

Data-driven Algorithm Selection for Carbon-Aware Scheduling https://hotcarbon.org/assets/2024/pdf/hotc
final23.pdf

Carbon aware computing at Google https://www.performance2021.deib.polimi.it/www.performance2021.dei
content/uploads/2021/11/Carbon-Aware-Computing-Year: 2021 Description: It states that Carbon-aware computing is: exploiting flexibility in when and where and how computing is done to reduce carbon emissions. Some examples of flexible workloads are: video processing, training large-scale machine learning models, simulation pipelines. The components they recognized to be necessary are: accurate carbon intensity data (Tomorrow), Scalable infrastructures (Cloud), Virtualizations and migration mechanisms (VMs), Well identified flexible load, data-driven methodology. At Google: the total amount of work that needs to get done per day is quite predictable.

2.6.7 State of the Art analysis outcome

many simulation compared to real-world scenarios no much flexibility for what concern variaty of resource managed usually tied to one type of resource (e.g., VMs, K8s pods)

usually either time shifting or geographical shifting

3 Design and Implementation

This chapter presents the design and implementation of our system, focusing on the integration of the various components and the overall architecture. The system is designed to be modular, scalable, and extensible, enabling the integration of additional components as needed.

3.1 Assumptions

In this section we present the assumptions made during the design and implementation of the system.

3.1.1 Workload definition

In this work, workloads has been modeled as **Virtual Machines (VMs)**, representing the **primary use case** considered during the system's initial design phase. It is possible to define as interrupible workloads, those workloads that can be stopped and restarted without losing the work done. For this first iteration of the system only **non-interruptible workloads** are considered. This choice was driven by the fact the Virtual Machines are both a common and widely used cloud and one of the simplest cloud resources to provision and therefore ideal for the first iteration of the system. Having said that, for the purpose of this work, a formalization can be done.

A VM is defined as a tuple:

$$VM = (MinCPU, MinRAM, D, DL, ML)$$

where:

- $MinCPU$ is the minimum number of virtual CPUs required.
- $MinRAM$ is the minimum RAM required (in GB).
- D is the duration for which the VM must run to complete its processing task P (in hours).
- DL is the deadline timestamp by which the VM must complete execution.
- ML is the maximum allowed latency in milliseconds. If latency is not a constraint, then a high value can be set (e.g., $ML = 2000$).

This VM can be scheduled on any Public Cloud Provider since we are interested in a multi-cloud system where the cloud can be effectively seen as a commodity. Alternatively, a subset of **eligible Public Cloud Providers** can be set at runtime by the user. We will refer to **general-purpose VMs** and not specialized ones like GPU instances or high-performance computing instances.

As an example, we can define a VM with the following specifications:

$$VM_{example} = (4, 4, 2, \texttt{"2025-03-20T23:59:59Z"}, 100)$$

where:

- $MinCPU = 4$ vCPUs
- $MinRAM = 4$ GB
- $D = 1$ hour
- $DL =$ `"2025-03-20T23:59:59Z"` (i.e., the processing task P inside the VM must complete before this timestamp)

The system is designed to be cloud-agnostic, however for the purpose of this work, the system is currently configured to support three major cloud providers: **Microsoft Azure**, **Google Cloud Platform (GCP)**, and **Amazon Web Services (AWS)**.

3.1.2 System limitations

A limitation of our general approach is that only resources supported by the cloud provider's Kubernetes operator can be provisioned in a seamless way. Not all cloud resources available in a provider's portfolio are guaranteed to have corresponding Kubernetes Custom Resources (CRs). This introduces certain constraints:

- Limited resource availability: if a specific resource type (e.g., a GPU-accelerated instance or a database service) is not supported by the cloud provider Operator, it cannot be provisioned using the current system.
- Dependence on Operator updates: cloud providers may extend or modify the set of resources supported by their Kubernetes operators over time.
- Vendor-specific implementations: for the same class of resources (e.g., virtual machines), the structure and fields of the CRs may vary a lot between cloud providers.

Despite these constraints, the system architecture remains highly adaptable, and future enhancements could incorporate additional or alternative provisioning mechanisms. An example of an alternative implementation could be the **direct API interactions** with cloud providers to bypass operator limitations. As a matter of fact, usually cloud provider Operators are leveraging these APIs under the hood to interact with the cloud provider's services. Another approach could involve the development of custom operators or controllers to manage specific resource types not supported by existing operators. For instance, Krateo PlatformOps provides the so-called "oasgen-provider" that aims to fill the gaps of missing or incomplete Kubernetes operators. This module is a K8s controller that is able to generate Kubernetes CRDs and the related controllers from OpenAPI specifications [**?**].

3.2 System Architecture

The following table provides an overview of the main components of the system and their respective functions.

Component	Function
Krateo PlatformOps	Provides an abstraction layer for infrastructure orchestration, enabling declarative resource management and integration with cloud providers with templates.
Cloud Providers Kubernetes Operators	Manages the provisioning and reconciliation of cloud resources within Kubernetes, ensuring the actual state matches the desired state.
Kubernetes Mutating Webhook Configuration	Intercepts and modifies API requests before they are persisted, allowing dynamic configuration adjustments with policy enforcement.
OPA Server	Evaluates policy decisions based on defined constraints and input data from Kubernetes API requests through the webhook configuration.
OPA Policies and Data	Define the rules and contextual information used by OPA to make policy decisions, namely scheduling information
GreenOps Scheduler	Determines the optimal scheduling region and scheduling time for VMs, acting as an external data source for OPA policies.
MLflow	Allows the tracking, logging, versioning and storing of machine learning experiments for reproducibility and model lifecycle management.
KServe	Provides scalable and Kubernetes-native model serving capabilities, enabling deployment of machine learning models for inference.

Table 3.1: Main components of the system and their respective functions.

All the components listed in the above table must be deployed inside a Kubernetes cluster. The only exception are the OPA Policies and data which lies outside the cluster as described in section **??**.

3.3 Krateo PlatformOps integration

Krateo PlatformOps is utilized in this system for **multi-cloud resource management**, allowing for the declarative orchestration of cloud resources across different cloud providers leveragin Kubernetes as a control plane. This section highlights the differences between two approaches for resource synchronization:

- The **Custom Kubernetes "Synchronization Operator"** approach
- The **Krateo PlatformOps** approach

It is deemed interesting to describe both approaches in order to identify the several trade-offs between implementing a custom synchronization operator and leveraging a template-based abstraction for cloud resource provisioning.

3.3.1 Resource management: the Custom Kubernetes "Synchronization Operator" approach

When dealing with multi-cloud workloads with Kubernetes as a control plane, a synchronization and mapping mechanism (broker) is required to bridge the gap between:

- **Generic Kubernetes Custom Resources**, which represent generic provider-agnostic workloads.
- **Cloud provider-specific Custom Resources**, which correspond to the actual cloud resources provisioned through the respective Kubernetes operators provided by Azure, AWS or GCP (in our case).

A custom Kubernetes Operator would be responsible for the mapping and synchronization of the above resource types. This approach is based on the principle of **Continuous Reconciliation**, where the operator continuously monitors and adjusts the system to maintain consistency between the desired and actual states. Candidate solutions for the development of the K8s Operator includes: Operator SDK, Kubebuilder, or writing the operator from scratch using the Kubernetes client libraries. For the purpose of this work, Kubebuilder was used to develop a Proof of Concept (PoC) for the custom synchronization operator.

Responsibilities of the Custom Operator

In our specific case, the operator should continuously watch the generic CRs in the Kubernetes cluster to check if critical scheduling fields have been set:

- **schedulingRegion**: Defines where the workload should be placed.
- **schedulingTime**: Specifies when the workload should be deployed.

These fields, if set, indicate a geographical placement and timing for the workload have been determined by the GreenOps Scheduler. If these fields are not yet present, the operator must wait for scheduling decisions before proceeding. Therefore, inside its Reconcile() loop, the operator should:

1. Continuously check if scheduling fields (schedulingRegion, schedulingTime) are set.
2. Trigger the creation of the provider-specific resource when the schedulingTime is approaching.
3. Track the provisioning status by marking the generic CR with a field indicating that the cloud-specific resource has been created.

Post-Creation Considerations

Once the cloud provider-specific resource is created, two main questions arise:

- What happens if the provider-specific CR is modified manually?
- What happens if the VM configuration is modified directly on the cloud provider (outside Kubernetes)?

Related to the first question, an example scenario could be: changing the VM instance type (VM size) inside the Kubernetes cluster. In this case, the operator needs to decide whether to revert unauthorized changes or allow them and update the generic CR accordingly. For the second question, an example could be: changing the VM size directly on the cloud provider's console. In this case, the operator should detect the drift and update the generic CR to reflect the external changes.

Resource linking

A mechanism must be in place to link the generic CR to the cloud provider-specific CR. Possible approaches include:

- UUID-based linking: A universally unique identifier ensuring each resource is mapped correctly.
- Kubernetes Object Metadata (ObjectMetadata.Name & ObjectMetadata.Namespace): This approach may be preferable within a single Kubernetes cluster, avoiding the need for an external ID system.

Termination Logic

The operator must handle the deletion of cloud resources correctly in a variety of scenarios:

- When the provider-specific CR is deleted from Kubernetes, the corresponding cloud resource is de-provisioned and the custom operator should ensure the deletion process is handled gracefully, avoiding orphaned generic CRs.
- If the provider-specific CR is deleted directly on the cloud provider, the operator should detect the change and update the generic CR accordingly.
- In the event of a generic CR deletion, the custom operator should ensure the provider specific resource is removed, troggering a deletion process on the cloud provider side (de-provisioning).

Managing cloud provider-specific fields

Each cloud provider has unique resource configurations and constraints that must be managed. Some differences are purely syntactic (e.g., AWS uses instanceType, whereas Azure uses vmSize). Others require additional provider-specific metadata (e.g., Azure requires a resourceGroup field which represent a logical container for resources in Azure). A custom synchronization operator must encode this logic explicitly, making it more complex to maintain especially when supporting several cloud providers.

Limitations of a Custom "Synchronization Operator"

Another major challenge with a custom synchronization operator is that some cloud providers do not support time scheduling metadata within their Custom Resources. In particular, no cloud operator among the ones we used for the system (AWS, Azure, GCP) provides a dedicated field for scheduling time. This means that the Kubernetes operator itself must handle time scheduling logic, delaying CR creation until the scheduled time. If the operator, upon the creation of a generic CR, immediately creates the cloud-provider specific CR (without a "waiting logic"), the cloud provider Operator will trigger and provision the VM immediately, ignoring scheduling constraints. Due to these limitations and complexities, we explored and leveraged an **alternative template-driven approach** using Krateo PlatformOps, described in the next section.

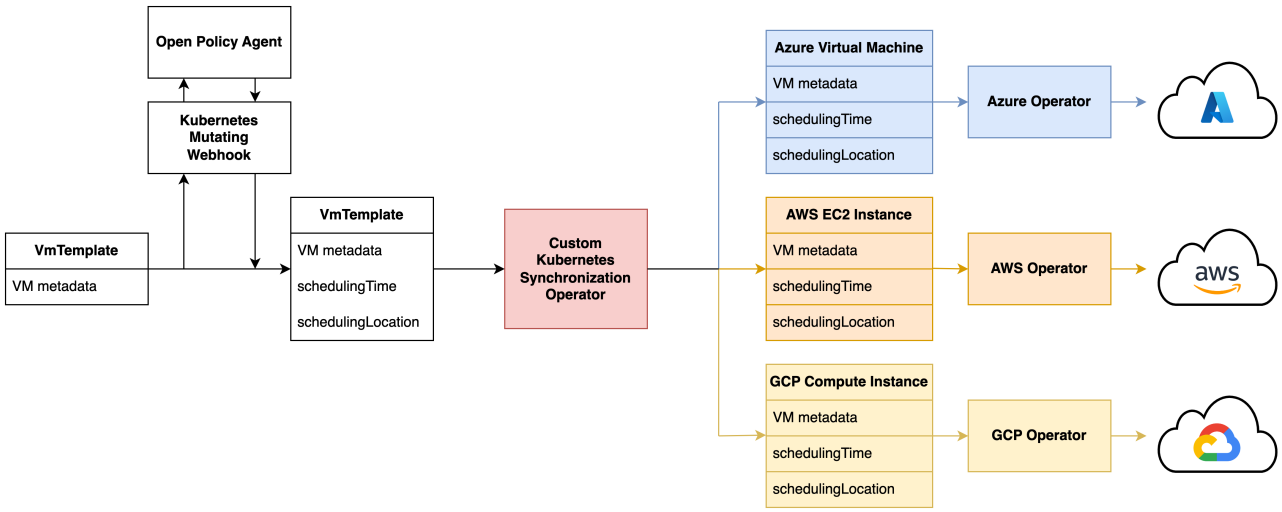

Figure 3.1: Multi-cloud resource management with Custom Kubernetes "Synchronization Operator" approach

3.3.2 Resource management: the Krateo approach

3.3.3 Strategic Shift: From Custom Operator to Krateo Core Provider

In our approach, we opted to replace a custom Kubernetes operator ("Synchronization Operator"), originally designed to handle the **mapping** from generic to cloud-specific resources, with **Krateo Core Provider**. This decision was motivated by the need for greater flexibility and maintainability in defining multi-cloud infrastructure components. As a matter of fact, the custom operator was originally

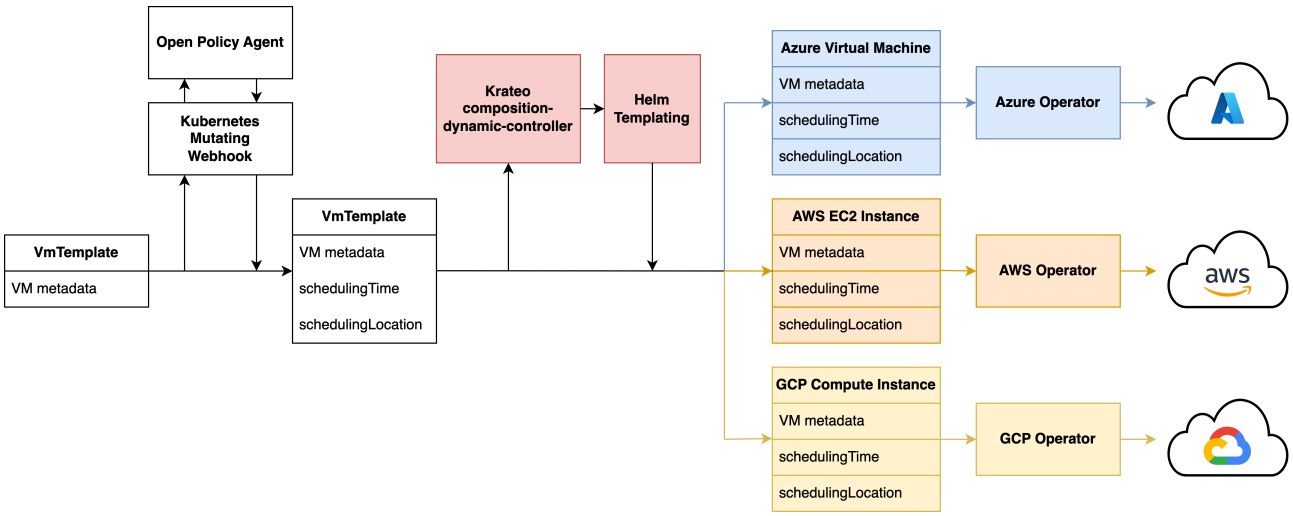

Figure 3.2: Multi-cloud resource management with Krateo PlatformOps approach

designed to handle only virtual machines (VMs) mappings and extensions to support additional cloud resources would have required significant code changes and maintenance overhead for each additional resource type added.

Therefore instead of embedding business logic directly within a custom Kubernetes operator, in the current system implementation, we leverage the capabilities of **Helm templating** to dynamically generate cloud-provider-specific resources. More precisely, another Krateo component, the **Krateo composition-dynamic-controller** is leveraging Helm templating under the hood to generate Kubernetes resources starting from helm templates. This approach, further described in the following sections, offers several advantages:

Simplified resource management: Helm enables a standardized way define resources without maintaining complex operator logic. Greater extensibility: By externalizing the logic from the operator, future modifications and integrations with additional cloud providers become easier. Reduced maintenance overhead: Operators typically require constant updates and refinements, while Helm-based resource generation minimizes the need for frequent code changes.

```yaml
# @param {string} vmName Name of the VM
vmName: test-vm

# @param {integer} cpu  Number of CPU cores
cpu: 1

# @param {integer} memory Number of GB of RAM
memory: 2

# @param {string} [schedulingTime] Scheduling Time for the VM
schedulingTime: 2025-05-05T00:00:00Z

# @param {string} [schedulingLocation] Scheduling Location for the VM
schedulingLocation: italynorth

# @param {string} duration Duration of the Workload
duration: 3h

# @param {string} deadline Deadline of the Workload
deadline: 2025-12-31T10:00:00Z

# @param {integer} maxLatency Maximum Latency of the Workload
maxLatency: 100
```

Listing 3.1: values.yaml

```
{
  "type": "object",
  "$schema": "http://json-schema.org/draft-07/schema",
  "required": [
    "vmName",
    "cpu",
    "memory",
    "duration",
    "deadline",
    "maxLatency"
  ],
  "properties": {
    "vmName": {
      "type": [
        "string"
      ],
      "description": "Name of the VM",
      "default": "test-vm"
    },
    "cpu": {
      "type": [
        "integer"
      ],
      "description": "Number of CPU cores",
      "default": "1"
    },
    "memory": {
      "type": [
        "integer"
      ],
      "description": "Number of GB of RAM",
      "default": "2"
    },
    "schedulingTime": {
      "type": [
        "string"
      ],
      "description": "Scheduling Time for the VM",
      "default": "2025-05-05T00:00:00Z"
    },
    "schedulingLocation": {
      "type": [
        "string"
      ],
      "description": "Scheduling Location for the VM",
      "default": "italynorth"
    },
    "duration": {
      "type": [
        "string"
      ],
      "description": "Duration of the Workload",
      "default": "3h"
    },
    "deadline": {
      "type": [
        "string"
      ],
      "description": "Deadline of the Workload",
      "default": "2025-12-31T10:00:00Z"
```

```
    },
    "maxLatency": {
      "type": [
        "integer"
      ],
      "description": "Maximum Latency of the Workload",
      "default": "100"
    }
  }
}
```

Listing 3.2: values.schema.json

Helm template engine (how to map to cloud provider specific resources, why is better)

Multi-cloud VMs list and mapping We need to know which VMs (instance types) are actually available on Azure, AWS, GCP. What to actually use to fetch the list and where to store it How frequent the elements changes and how to update the list (this is done with policy updates (updates to contextual data)) How to match generic workload to candidate VMs instance types (that adhere to the generic workload specs), what is the mapping logic. For example: Requested VM (generic): (2 vCPU, 16 GiB) Chosen VM instance: Azure Standard E2as v4 What are the steps of the process to identify the correct VM instance?

Use Cloud Providers API / CLI and store the records in a local DB. Regular scheduled updates. For our PoC we could maybe use a small subset of all the available instance types.

Current approach: Map (CPU, RAM) =¿ InstanceType

This mapping is done through Helm templating when creating

compostion definition

3.4 Kubernetes Mutating Webhook Configuration

3.4.1 Kubernetes Admission Control

In the context of **Kubernetes Admission Control**, in addition to standard, compiled-in admission plugins, Kubernetes supports the use of additional admission plugins that are effectively extensions of the system and run as **webhooks** configured at runtime [**?**]. This means that the admission control logic can be extended dynamically without the need to recompile the Kubernetes API server or other Kubernetes components. Changes are applied at runtime to the running Kubernetes cluster, making the system more flexible and adaptable.

These plugins can be used to enforce custom policies and perform additional validation and mutation of Kubernetes objects before they are persisted in the cluster.

K8s mutating webhook is used to modify K8s custom resources with the data from policies. The K8s mutating webhook intercepts the CREATE or UPDATE API request, asks about policies (with also scheduling decision) and mutates the resource.

we can distinguish between two types of webhooks: validating and mutating.

we can distinguish two entities semantically different: the webhook configuration and the actual webhook /server which perform mutation or validation

it could be a server that you need to implement to perform your own custum mutation or validation logic it could be a service ready to use out of the box like OPA server

service can be either a deployment with the related service inside the cluster like seen in the image **??**

or can be deployed outside the cluster

Figure **??** shows an example of a Kubernetes Mutating Webhook. The webhook server is deployed as a Kubernetes Deployment (with 1 Pod), with a corresponding Service to expose it within the cluster. The webhook server is responsible for receiving The AdmissionReview Request, applying custom logic, and returning an AdmissionReview Response to the Kubernetes API server describing the mutation to be applied to the resource. The custom logic in this simple example is just to add a label "mutated: true" to the resource.

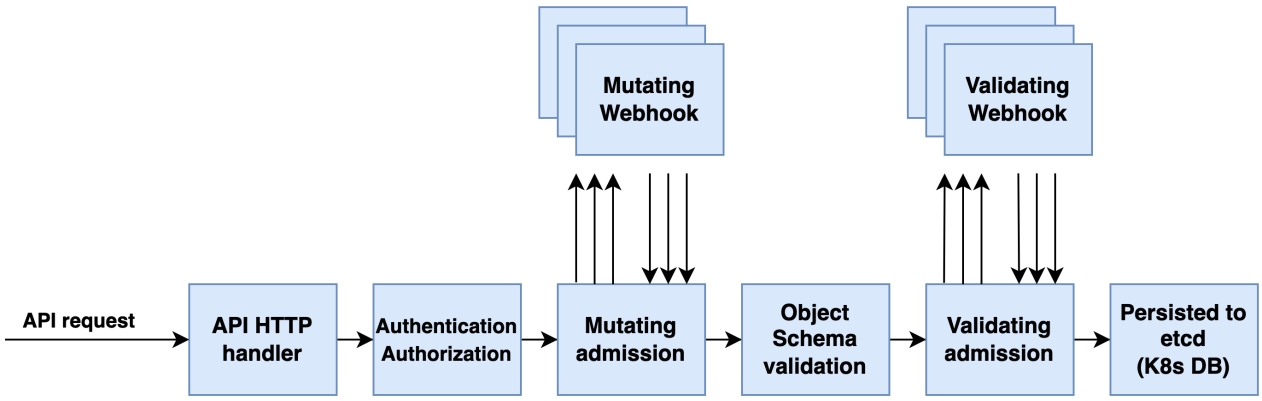

Figure 3.3: Kubernetes Admission Control

3.5 Multi-Cloud Integration through Kubernetes Operators

The integration of operators from different cloud providers has enabled the development of an effective **multi-cloud system**, allowing seamless orchestration and provisioning of cloud resources across various cloud platforms. Namely, the system leverages Kubernetes operators from **Microsoft Azure**, **Google Cloud Platform (GCP)**, and **Amazon Web Services (AWS)**.

3.5.1 Role of Kubernetes Operators

Kubernetes operators work on the principle of Continuous Reconciliation, ensuring, in this case, that the desired state of the system, as defined by users, aligns with the actual state of provisioned cloud resources. In particular, Operators act as controllers that monitor, adjust, and manage external cloud resources within a Kubernetes-native environment. Inside the K8s lie the representation of the cloud resources, which are managed by the operators. Key characteristics of operators include:

- Managing external cloud resources within a Kubernetes cluster, providing a **unified interface** for multi-cloud deployments.
- Maintaining a **real-time representation** of provisioned cloud resources within Kubernetes.
- Using Custom Resources (CRs) to define cloud-specific resources in a **declarative** manner.

It must be noted that different cloud provider adopts different design choices for their Kubernetes operators and more in general for their overall infrastructure management. Therefore, for the creation of logically similar resources, like a virtual machine, the structure and the field of the resources can be different. These resources typically include:

- Compute resources (e.g., VM instances, virtual machine templates)
- Networking components (e.g., virtual networks, subnets, security groups)
- Storage allocations (e.g., persistent volumes, cloud disks)
- Access management (e.g., resource groups, roles, authentication credentials)

For the purpose of this work we defined a **baseline infrastructure** for each cloud provider taken into account in order to have a common ground for the system to work. This baseline infrastructure is composed by the minimum set of resources needed for a VM provisioning. Each public cloud provider has its complexities and nuances when it comes to managing cloud resources. In the following sections, we provide an overview of the minimum set of resources needed for VM provisioning on each cloud provider, as well as some of the specific configurations required for each resource.

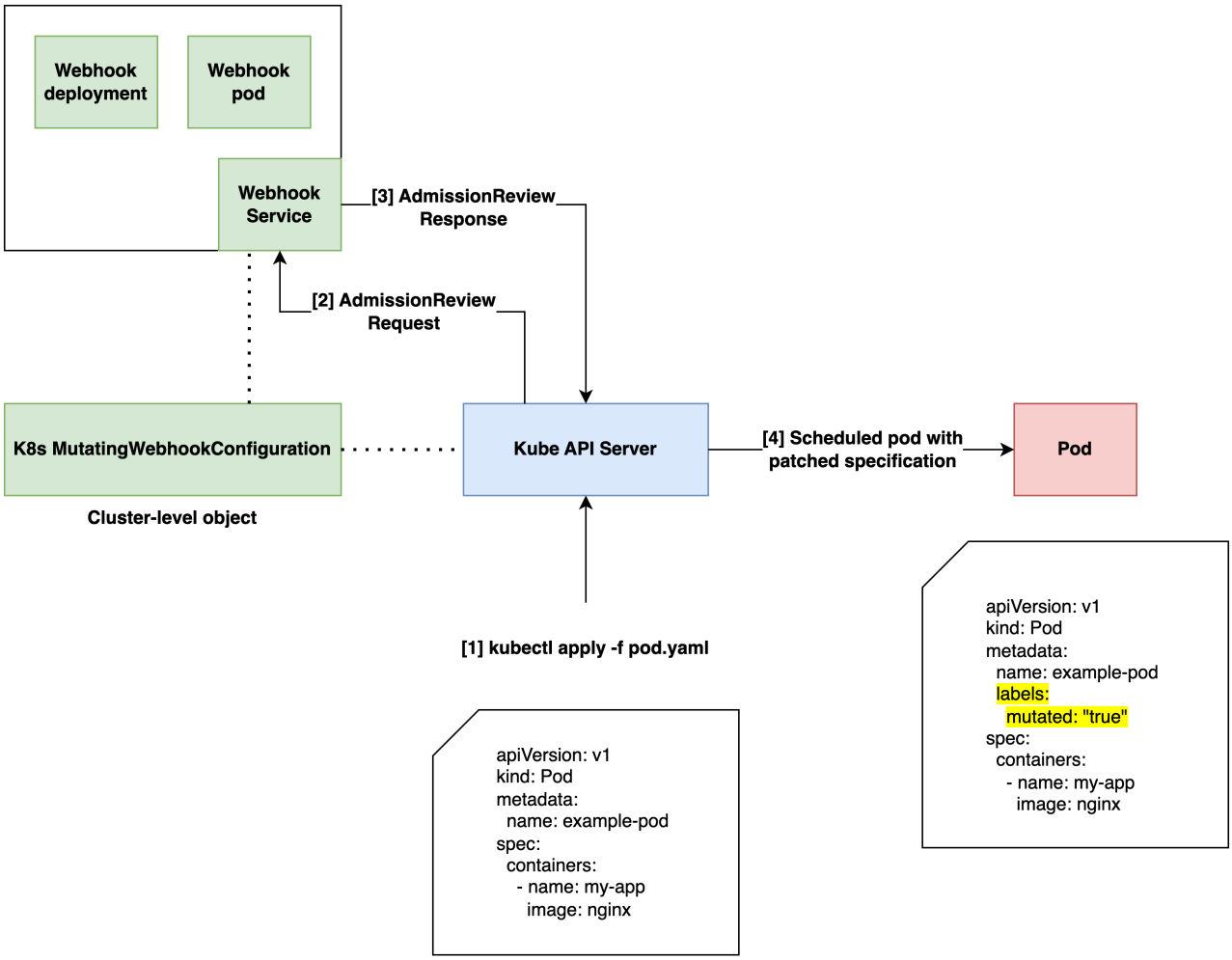

Figure 3.4: Kubernetes Mutating Webhook example [**?**]

3.5.2 Azure Kubernetes Operator

Microsoft Azure provides a Kubernetes operator called **Azure Service Operator v2** (ASO). Currently, ASO supports more than 150 different Azure resources. minimum set of resources needed for vm provisioning on Azure throgh Azure service operator is is:

- Virtual Network
- Virtual Network Subnet
- Network Interface
- Virtual Machine

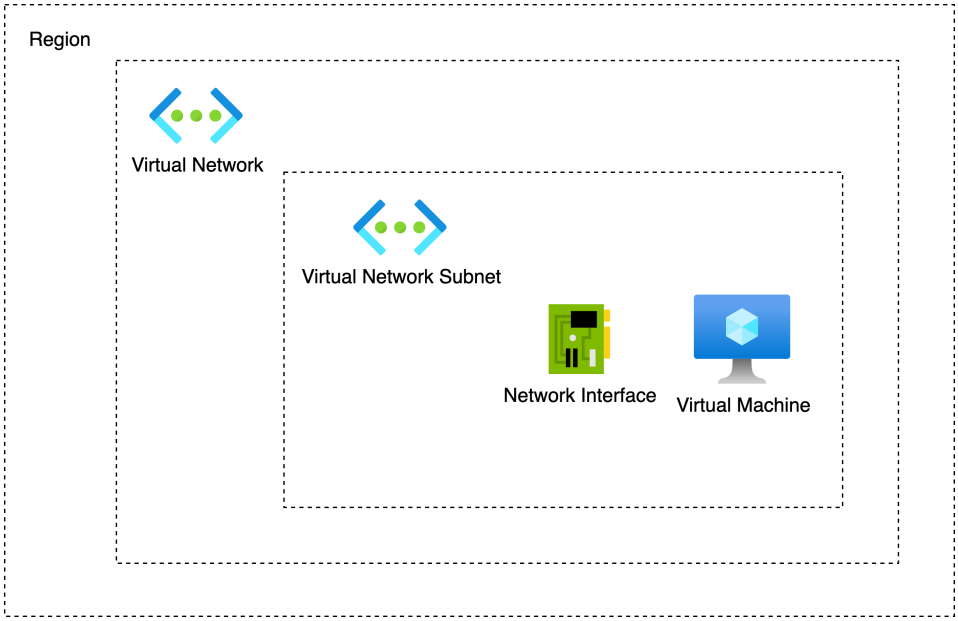

Figure 3.5: Minimum set of Azure resources for VM provisioning

INSTANCE CR example

3.5.3 GCP Operator

minimum set of resources needed for vm deployment

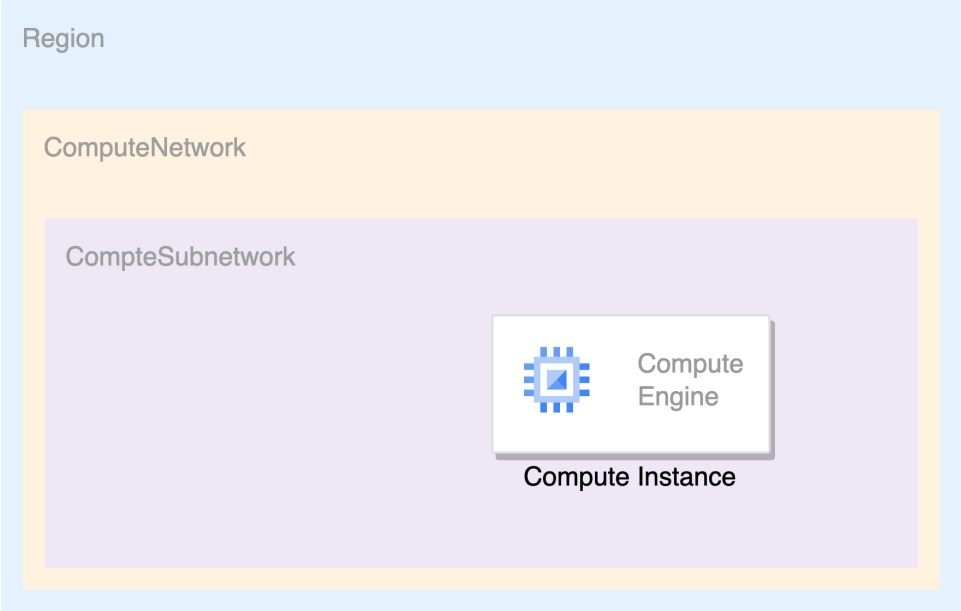

Figure 3.6: Minimum set of GCP resources for VM provisioning

INSTANCE CR example
some fields are based on regions some fields are based on zones
networkinterface is directly defined in the instance manifest, no additional networkinterface resource neededv to be created

3.5.4 AWS Operator

this is a collection of operators that are part of the AWS controllers for Kubernetes (ACK) project.
The minimum set of resources needed for vm provisioning

- VPC

- Subnet
- EC2 Instance

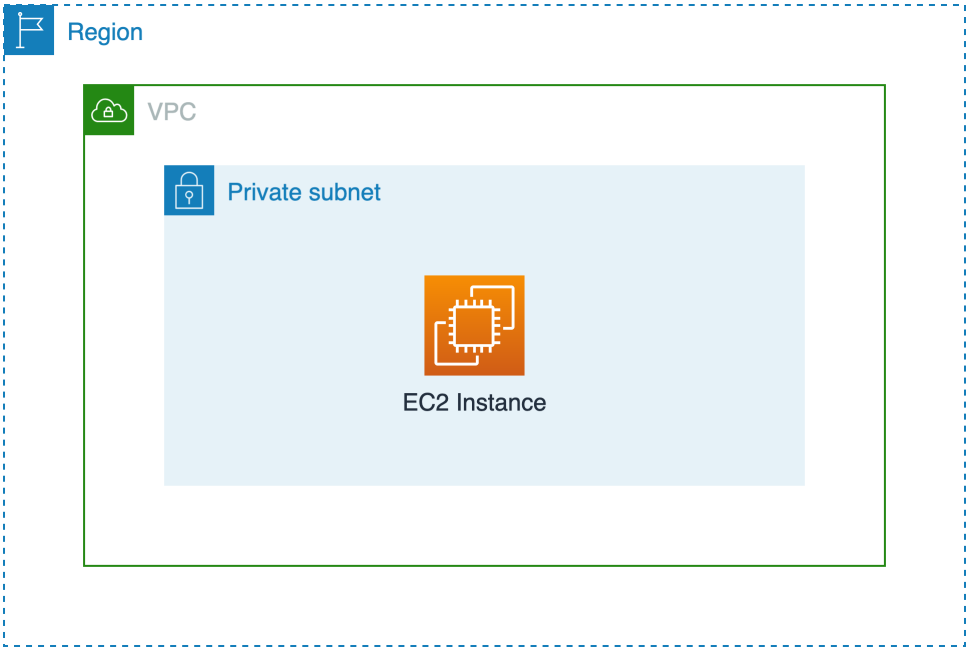

Figure 3.7: Minimum set of AWS resources for VM provisioning

As described at the beginning of this section, the implementation approach adopted in our system ensures compatibility with diverse cloud provider design choices. Cloud providers may impose different constraints and best practices when managing Kubernetes-native resources, and the system is designed to adapt to these variations seamlessly.

One notable design choice observed with the AWS operator is the restriction on referencing some Kubernetes objects inside a Custom Resource (CR) manifest. This limitation means that developers cannot directly link a resource (e.g., a Virtual Machine) to another Kubernetes object (e.g., a Subnet) using built-in object references.

To overcome this limitation, our system leverages **Helm's *lookup function***, which dynamically retrieves Kubernetes object details at runtime. This method allows us to fetch required parameters without directly referencing Kubernetes objects in the CR, ensuring compatibility with the AWS operator's design constraints. The following example demonstrates how the lookup function can be used to resolve subnet IDs dynamically and inject them into the CR manifest.

```
...
apiVersion: ec2.services.k8s.aws/v1alpha1
kind: Instance
metadata:
  name: {{ .Values.vmName }}
  namespace: {{ .Values.namespace | default "greenops" }}
...
spec:
  ...
  subnetID: {{ (lookup "ec2.services.k8s.aws/v1alpha1" "Subnet" (.Values.namespace |
      default "greenops") (printf "%s-subnet" .Values.vmName)).status.subnetID }}
...
```

Listing 3.3: Helm Lookup example: dynamically resolving SubnetIDs

The Helm lookup function can be used to look up resources in a running cluster and its synopsis is: "lookup apiVersion, kind, namespace, name -¿ resource or resource list" [**?**]. In the listing **??**, the

Helm lookup function retrieves the subnetID from a Subnet Custom Resource dynamically, based on the VM name and namespace. Then, the subnetID is injected into the Instance Custom Resource manifest, ensuring that the VM is provisioned in the correct subnet.

An example by the same AWS Operator where instead a direct reference to a resource is allowed is the one illustrated in listing **??**.

```
...
apiVersion: ec2.services.k8s.aws/v1alpha1
kind: Subnet
metadata:
  name: {{ .Values.vmName }}-subnet
...
spec:
  vpcRef:
    from:
      name: {{ .Values.vmName }}-vpc
      namespace: {{ .Values.namespace | default "greenops" }}
...
```

Listing 3.4: AWS Operator direct reference example

In thr case of Listing **??**, the Subnet Custom Resource manifest directly references in a conveneint way the VPC Custom Resource using its name and namespace since the Operator is designed to support this type of relationship. As explained before, this is determined by Operator design choices but our system is able to handle both scenarios.

Provider specific configurations

An Amazon Machine Image (AMI) is a pre-configured image that provides the necessary software environment to set up and boot an Amazon EC2 instance [**?**]. In other words, AMIs serve as a blueprint for launching virtual machines (VMs) in AWS.

When launching an instance, specifying an AMI is **mandatory**. The AMI must be compatible with the chosen EC2 instance type, ensuring that the selected image supports the required hardware and software configurations.

The following attributes define an AMI:

- Region: AMIs are region-specific
- Operating System: Determines the base OS (e.g., Ubuntu, Windows, RHEL) installed on the AMI.
- Processor Architecture: e.g., x86, ARM
- Root Device Type: Specifies whether the AMI uses an EBS-backed volume (Elastic Block Store) or Instance Store for storage.
- Virtualization Type: Defines whether the AMI supports paravirtual (PV) or hardware virtual machine (HVM) instances.

For the purpose of this research, only **Ubuntu-based AMIs** have been considered for provisioning virtual machines. Official Ubuntu AMIs were collected from a dedicated Ubuntu repository. In order to select the most suitable AMI for a given VM, the system leverages Helm template engine to dynamically select the appropriate AMI ID based on the region and other parameters specified in the VmTemplate Kubernetes Custom Resource (CR).

3.6 Open Policy Agent (OPA)

Open Policy Agent (OPA) is an open-source general-purpose **policy engine** that enables unified policy enforcement across cloud-native environments. OPA provides a declarative language called Rego enabling a paradigm known as "**Policy as Code**" [**?**].

Open Policy Agent can be integrated as a sidecar container, host-level daemon, or library to perform policy decisions for a plethora of use cases: microservices, Kubernetes admission control, CI/CD pipelines, API gateways and more [**?**].

In the context of our system, OPA and the Policy-as-Code paradigm are used to enforce policies for workload scheduling: encoding the output of a scheduling decision coming from an external GreenOps Scheduler and ensuring compliance with additional policies related to latency requirements and legal constraints.

3.6.1 Policy as Code paradigm

According to AWS, Policy-as-Code (PaC) is a software automation approach which is similar to Infrastructure-as-Code (IaC) [**?**]. PaC helps assess company system configurations and validate compliance requirements through software automation [**?**]. The perceived value of this type of automation in the software development lifecycle has grown significantly in modern enterprises. This large adoption is probably driven by the inherent consistency and reliability it provides, ensuring standardized enforcement of policies and reducing human error [**?**].

OPA's generic defintion of policy is: "*A policy is a set of rules that governs the behavior of a software service*" [**?**]. OPA provides a high-level declarative language called **Rego** to define policies in a flexible manner. One of OPA's key strengths is its **domain-agnostic design**, allowing it to enforce policies across various systems and environments. This makes it highly adaptable to different use cases, ranging from access control to infrastructure security. Some representative examples of policies that OPA can enforce include:

- Restricting which image registries can be used for deploying new Pods in a Kubernetes cluster.
- Controlling whether a specific user is permitted to perform delete operations on certain resources.
- Enforcing network security policies, such as blocking external access to sensitive services.
- Ensuring infrastructure compliance, for example, by verifying that new cloud resources to be provisioned follow predefined security configurations.
- Enforcing that new deployed servers must have the prefix "server-" in their name.

Therefore, the use cases covered span from role-based access control to container image security and beyond.

Another important aspect of OPA is that it effectively **decouples** policy decision-making from policy enforcement, enabling organizations to implement consistent and scalable authorization across their systems [**?**]. In practice, this means that when a software module needs to make a policy decision, it queries OPA, supplying relevant data as input. In other words, policy decisions are **offloaded** to OPA rather than being hardcoded within individual services. This approach offers several key advantages:

- **Centralized policy management**: policies are defined in a single location, ensuring uniform enforcement across all services of an organization.
- **Improved maintainability**: updating policies does not require modifying, recompiling or redeploying application code, reducing complexity and deployment overhead.
- **Greater flexibility**: policies can be dynamically updated (e.g., with CI/CD approaches) based on evolving security and compliance requirements.
- **Scalability**: since OPA and application modules are not tightly coupled.

3.6.2 OPA architecture overview

As mentioned in the introduction to this section, one common approach to integrating OPA into a software system is by deploying it as a host-level daemon. The latter is essentially a lightweight server

that processes policy queries via HTTP requests. This setup allows services to offload policy decision-making to OPA in a scalable and efficient manner since the two entities are not tightly coupled.

A standard OPA deployment consists of three main components:

- **OPA Server**: The core service that evaluates policy queries and returns decisions based on defined rules, contextual data and input data.
- **OPA Policies**: Rules written in the Rego language that define the logic to be enforced.
- **Data**: Optional contextual information, typically structured in JSON format, that policies use to make informed decisions along with input data.

To facilitate deployment and management, Rego policies and associated contextual data are packaged into **policy bundles**, as described in section **??**. These bundles enable version-controlled, centralized policy distribution, ensuring consistency and maintainability across distributed environments.

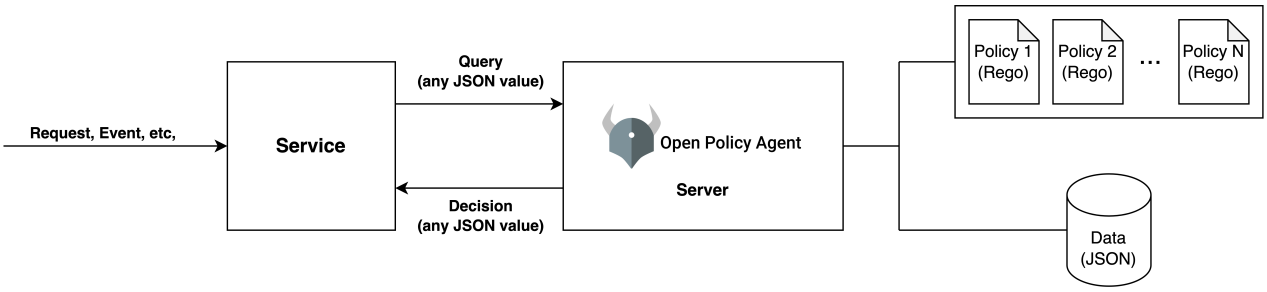

Figure 3.8: OPA architecture

OPA accepts arbitrary structured data as input. and Like query inputs, your policies can generate arbitrary structured data as output.

3.6.3 OPA and external data

types of external data strategies

http.send() paramters

3.6.4 OPA integration with Kubernetes

In Kubernetes admission control, policy enforcement is handled by the **Kubernetes API server** itself. OPA makes the policy decisions when queried by the admission controller, but the actual enforcement (namely allowing or denying requests) is executed by Kubernetes' built-in admission control mechanisms. This workflow is represented in figure **??** where **AdmissionrReview request** and **AdmissionReview response** are respectively input and output of the whole OPA section. The API Server sends the entire Kubernetes object in the webhook request to OPA. The Kubernetes API server will use the received AdmissionReview response for its decision.

In a Kubernetes deployment, an OPA Pod typically consists of the following containers:

- OPA server container
- **kube-mgmt** container

The kube-mgmt container functions as a **sidecar container** within a Kubernetes Pod. The sidecar container pattern is a common Kubernetes design paradigm in which auxiliary containers run alongside the main application container within the same Pod. These additional containers serve to enhance, extend, or support the primary application's functionality without modifying its core logic [**?**]. The primary responsibility of kube-mgmt is to replicate Kubernetes resources into the OPA instance (OPA container). This operation is essential for OPA to access and evaluate policies based on real-time cluster state, enabling dynamic policy enforcement. By synchronizing these resources, kube-mgmt ensures that OPA has an up-to-date view of relevant Kubernetes objects. This is especially useful to enforce policies that deals with naming conflicts, where OPA needs to check existing names in

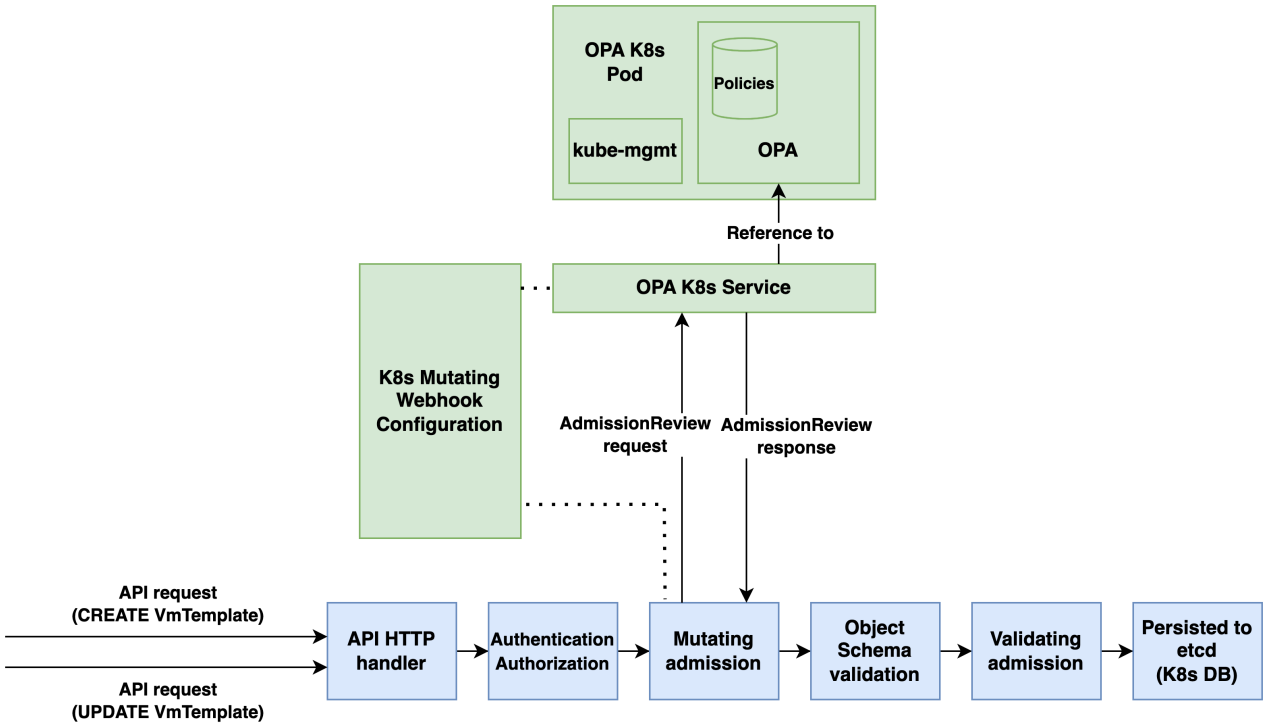

Figure 3.9: Kubernetes mutating webhook and OPA integration

the cluster for the decision [**?**]. Additionally, it allows for loading policies directly from the Kubernetes cluster by retrieving them in the form of ConfigMaps. This feature is particularly useful when policies need to be dynamically updated based on the current state of the cluster [**?**]. However, in the system described in this thesis, this latter feature is not employed in the current implementation as we are using policy bundles for policy distribution.

In the current system configuration, the kube-mgmt container is deployed to facilitate resource replication, ensuring that Kubernetes resources, namely VmTemplate resources, are synchronized with the OPA instance. However, at present, no policy requires interrogation of VmTemplate resources that are already present in the system. Looking ahead, future policies could leverage VmTemplate resource information to enforce naming conflict resolution, quota management, or additional constraints.

3.6.5 OPA policies

As OPA official documentation describes, when the Kubernetes AdmissionReview request from the webhook arrives, it is binded to the OPA input document and generates the default, "root", decision: *system.main*

The root policy, in the case of Kubernetes admission control, is responsible for generating the AdmissionReview response in accordance with the Kubernetes API specifications. It is the duty of the policy developer to write Rego code that produces a well-formed AdmissionReview response, ensuring that the OPA server can then correctly communicate its decision to the Kubernetes admission controller.

It is deemed useful to show one of the simplest and common example of a OPA policy in the **Kubernetes admission control context**. That is: to ensure all images for Kubernetes Pods come from a trusted registry, namely *unitn.it*.

It is important to note that, in this case, due to the simplicity of the policy, no additional contextual data in JSON format is required.

policy compilation policy are compiled compile time errors like merge errors if data is clashing for instance

```
deny contains msg if {
    input.request.kind.kind == "Pod"
    image := input.request.object.spec.containers[_].image
    not startswith(image, "unitn.it/")
    msg := sprintf("image '%v' comes from untrusted registry", [image])
}
```

Listing 3.5: Rego policy for Pods registry

```
package system

import data.kubernetes.admission

main := {
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": response,
}

default uid := ""

uid := input.request.uid

response := {
  "allowed": false,
  "uid": uid,
  "status": {"message": reason},
} if {
  reason := concat(", ", admission.deny)
  reason != ""
}

else := {"allowed": true, "uid": uid}
```

Listing 3.6: Rego "root" policy (system.main)

```
{
    "apiVersion": "admission.k8s.io/v1",
    "kind": "AdmissionReview",
    "request": {
        "kind": {
            "group": "",
            "kind": "Pod",
            "version": "v1"
        },
        "object": {
            "metadata": {
                "name": "myapp"
            },
            "spec": {
                "containers": [
                    {
                        "image": "bitnami/node:22",
                        "name": "nodejs"
                    }
                ]
            }
        }
    }
}
```

Listing 3.7: AdmissionReview request

```
{
    "apiVersion": "admission.k8s.io/v1",
    "kind": "AdmissionReview",
    "response": {
        "allowed": false
        "status": {
            "message": "image 'bitnami/node:22' comes from untrusted
               registry"
        }
    }
}
```

Listing 3.8: AdmissionReview response

Therefore, in this specific case, the creation of the Kubernetes Pod will be **denied**. OPA is responsible for **decision-making**, determining that the request do not complies with the defined policies, while the Kubernetes API server, using the AdmissionReview response generated by OPA, handles **policy enforcement**, effectively rejecting the CREATE request since it violates the specified rules.

3.6.6 OPA Policy bundles

An OPA policy bundle is a collection of policies and optional associated contextual data. More precisely, a bundle is a standardized way to package policies, facilitating version control and distribution [**?**]. As a matter of fact, a single policy bundle can be potentially used by multiple OPA instances. A policy bundle mainly consists of:

- **Rego policy files** defining the logic.
- **Data files** (in JSON or YAML format) containing contextual information required for policy evaluation (e.g., cloud region mappings).

Policy bundles can be distributed through a variety of mechanisms such as remote HTTP servers (e.g., NGINX) and object storage services (e.g., Amazon S3, Google Cloud Storage, Azure Blob Storage) [**?**]. One of the most convenient approaches is packaging them as **OCI (Open Container Initiative) images** [**?**] and this is the approach adopted in the system described in this thesis.

Once packaged as OCI images, policy bundles can be pulled by OPA servers from a container registry at predefined time intervals. This allows policy updates to be deployed in OPA **without requiring manual intervention** or **service restarts**, ensuring that enforcement mechanisms remain up to date with the latest compliance requirements identified and implemented by the organization. This is crucial for instance when dealing with **critical security policies** that need to be updated frequently, maybe in response to the discoveries of new CVEs. In the context of our system, such timely updates are not essential but the OPA is designed to be able to handle them if needed. To ensure continuous policy enforcement while maintaining high operational efficiency, a CI/CD approach is adopted for policy management in the context of our system. As a matter of fact, policies are maintained in a **version-controlled hosted repository** (i.e., on GitHub), where updates like tagging ("git tag") trigger an automated pipeline (e.g., using GitHub Actions) responsible for building, packaging into a OCI image, and publishing the policy bundle to a container registry (e.g., Docker Hub). One of the major advantages of this approach is the ability to dynamically update policies without requiring OPA pods to restart as there is an **hot-reload** of policies done at application level by OPA ("loaded on the fly") [**?**]. This is particularly useful in production environments where service availability is critical and downtime must be minimized. The overall process of policy distribution is illustrated in figure **??**.

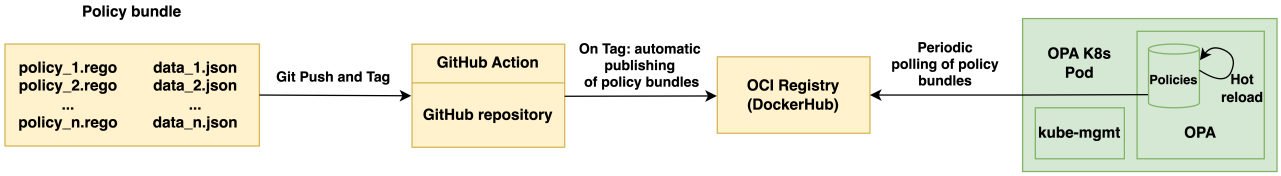

Figure 3.10: OPA policy bundles

By leveraging OCI images for policy distribution and implementing a fully automated CI/CD pipeline, our system ensures that policy enforcement remains consistent, up to date, and highly available across all OPA instances. This approach aligns with modern DevOps practices, enabling organizations to maintain a high level of security and compliance without compromising operational efficiency.

3.6.7 OPA Gatekeeper

OPA Gatekeeper is a Kubernetes-native policy engine that extends OPA with **Custom Resources (CRs)** and controllers to enforce policies across a Kubernetes cluster. It integrates natively with Kubernetes and provides a declarative approach to defining and enforcing policies using Kubernetes Custom Resources (CRs). This makes it an excellent choice for basic and standard policy enforcement scenarios, such as RBAC (Role-Based Access Control), security compliance, and resource constraints. However, while OPA Gatekeeper is well-suited for simple use cases, it presents **limitations** when addressing complex policy requirements, particularly when policies involve **mutations** or require access to **external data sources**. These limitations make it unsuitable for the specific challenges tackled in this system. Therefore, after an initial investigation and Proof of Concept implementation, we decided to use the standard OPA server for policy enforcement mainly due to the flexibility it provides in handling diverse scenarios.

To illustrate the differences between a standard OPA policy and an OPA Gatekeeper policy, we present two examples:

- a simple Rego policy that enforces a basic constraint on Pod creation in a Kubernetes cluster.
- the corresponding policy implemented as an OPA Gatekeeper **ConstraintTemplate** and **Constraint** Kubernetes resources.

The first example demonstrates a standalone Rego policy, which can be evaluated directly by an OPA instance. While this approach is flexible and allows for fine-grained policy definition, it requires manual integration into the system, including policy distribution and enforcement setup.

```
package kubernetes.admission

deny[msg] {
  input.request.kind.kind == "Pod"
  input.request.object.metadata.namespace == "restricted"
  msg := "Pods cannot be created in the 'restricted' namespace."
}
```

Listing 3.9: Simple OPA Rego Policy

The second example, illustrated in listing **??** utilizes OPA Gatekeeper, which extends Kubernetes with Kubernetes-native Custom Resource Definitions (CRDs), enabling declarative policy management. By using a ConstraintTemplate, policies can be enforced dynamically through Kubernetes, making them easier to distribute and manage. In other words, with this kind of setting, OPA policy bundles are not employed in the same way as in the standard OPA server. Instead, policies are defined as Kubernetes resources, allowing for more straightforward policy enforcement and management within a Kubernetes environment.

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: podnamespaceconstraint
spec:
  crd:
    spec:
      names:
        kind: PodNamespaceConstraint
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package kubernetes.admission
        deny[msg] {
          input.review.object.metadata.namespace == "restricted"
          msg := "Pods cannot be created in the 'restricted' namespace."
        }
```

Listing 3.10: OPA Gatekeeper ConstraintTemplate

In the example, the policy is defined as a ConstraintTemplate, which is then instantiated as a Constraint Custom Resource of kind defined in the ConstraintTemplate. The ConstraintTemplate

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: PodNamespaceConstraint
metadata:
  name: restrict-namespace
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
  parameters: {}
```

Listing 3.11: OPA Gatekeeper Constraint

specifies the Rego policy logic, while the Constraint defines the target resources and parameters for policy enforcement. Therefore a ConstraintTemplate can be used by multiple Constraints, allowing for policy reuse.

OPA Gatekeeper also provides additional Kubernetes Custom Resources called *mutators* (Assign, AssignMetadata, AssignImage, ModifySet) that allow modifying resource fields without writing Rego code. These mutators are useful for simple transformations, such as setting default labels or annotations. However simultaneous mutation of multiple fields leveraging external data is not supported [**?**]. This limitation, in the context of our system, determined the choice of the standard OPA server for policy enforcement.

It must be noted that OPA Gatekeeper limitations could be potentially addressed in future releases, making it a more viable option for complex policy enforcement scenarios. However, for the current system requirements, the standard OPA server was deemed more suitable due to its flexibility.

3.6.8 Latency policy

A representative example of a policy aligned with Service Level Objectives (SLOs) or Service Level Agreements (SLAs) is the latency policy described in this section. Given an **origin region** and a **maximum latency threshold** (expressed in milliseconds), the objective is to determine a **set of eligible regions** where the inter-regional latency between the origin and each region in the set is equal to or below the specified threshold. Enforcing such constraints helps mitigate the so-called "**black hole phenomenon**" in the GreenOps use case, where all virtual machines (VMs) would otherwise be scheduled in a region with generally low carbon intensity, without considering additional constraints or performance requirements. By incorporating similar performance-aware policies, organizations can achieve a balance between environmental impact, performance, and service reliability. The proposed flexible system enables organizations to fine-tune these factors according to their specific requirements or those of their users. This policy demonstrates the flexibility of OPA in handling diverse compliance scenarios. It is the responsibility of the policy developer to design an appropriate strategy for encoding relevant information into **well-structured JSON data models**, e.g., a latency matrix. Proper structuring ensures efficient policy evaluation, maintainability and extendability.

Figure **??** illustrates an small example (4 regions subset) of a latency matrix, where each cell represents the latency between two regions. The matrix can be encoded in JSON format as illustrated in listing **??**, allowing for easy integration with OPA policies. The "Latency policy" then uses this matrix to determine eligible regions based on the origin region and maximum latency threshold.

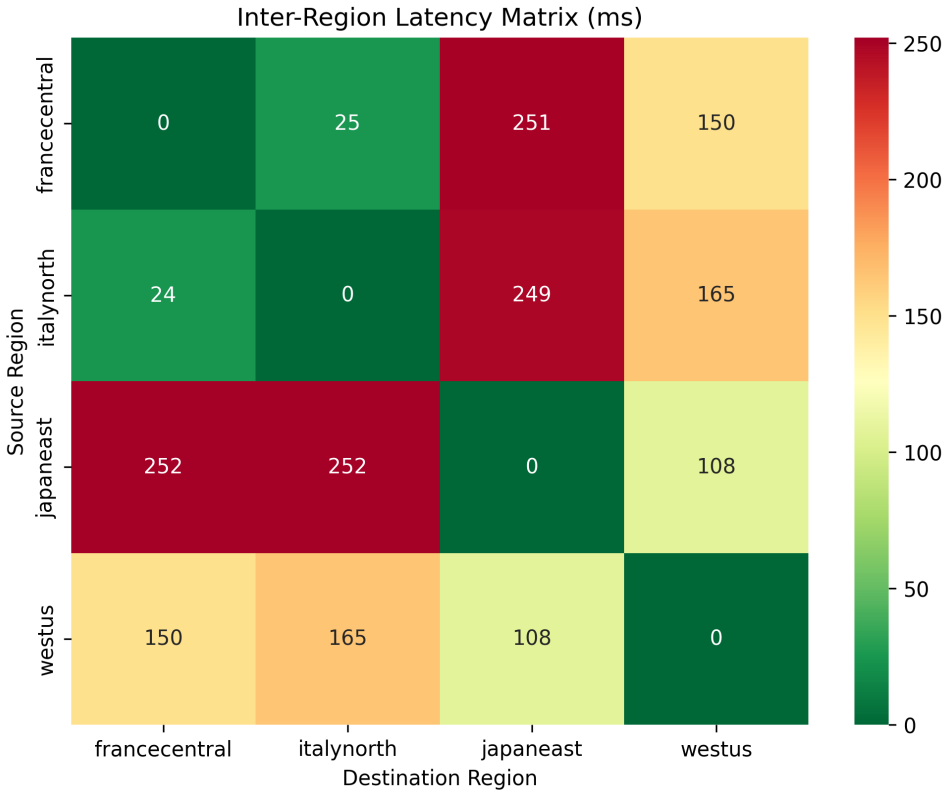

Figure 3.11: Latency matrix example (Azure regions subset)

```
{
  "italynorth": {
      "italynorth": 0,
      "japaneast": 249,
      "francecentral": 24,
      "westus": 165
  },
  "japaneast": {
      "italynorth": 252,
      "japaneast": 0,
      "francecentral": 252,
      "westus": 108
  },
  "francecentral": {
      "italynorth": 25,
      "japaneast": 251,
      "francecentral": 0,
      "westus": 150
  },
  "westus": {
      "italynorth": 165,
      "japaneast": 108,
      "francecentral": 150,
      "westus": 0
  }
}
```

Listing 3.12: Latancy matrix example encoded in JSON format

data

Azure provides monthly Percentile P50 round trip times between Azure regions: (https://learn.microsoft.com/en-us/azure/networking/azure-network-latency?tabs=Americashttps://learn.microsoft.com/en-us/azure/networki

network-latency/azure-network-latency-thumb.pnglightbox

Azure network latencies docs: https://raw.githubusercontent.com/MicrosoftDocs/azure-docs/refs/heads/main/ar
network-latency.md

Merged Azure network latencies: https://docs.google.com/spreadsheets/d/1kxtPw9ZSnAv1vQ6IDwzw-mXdHoKUFb8AjlsjACnvDqE/edit?usp=sharing

AWS: https://www.cloudping.co/grid (not official, not using VMs but Lambda functions) how it is calculated

google syntetic data no offical data

3.6.9 GDPR policy

Another policy configured in the system is the "GDPR Policy", which ensures that virtual machines (VMs) are deployed in cloud regions that reside in countries of the European Union. The policy is based on the principle of **set intersection**. One set consists of the eligible regions determined by other constraints, such as latency requirements. The other set includes cloud provider regions that are physically located within European Union (EU) countries. The intersection of these two sets defines the final list of allowed deployment regions, restricting workloads to EU-based data centers. Since each cloud provider has its own regional distribution, the list of EU-compliant regions is provider-specific and is encoded as contextual data in JSON format. This allows for flexibility and easy updates when cloud providers introduce new regions.

It must be noted that this policy is **not intended to be a comprehensive GDPR compliance solution**, but rather a basic example of how OPA can enforce **data residency requirements in a multi-cloud environment**. Organizations with more stringent GDPR compliance needs should consider additional measures.

3.6.10 Scheduling outcome policy

main policy

Mutation policy dedicated to

JSON Patch is a format for describing changes to a JSON document which avoid the need to send the entire document when only a part of it has changed. Effectively, only deltas are sent back to the requester which are themselves JSON documents. The format is defined in RFC 6902 from the IETF [**?**].

As an example, a single patch operation

OPA, assuming the role of a webhook server, has the duty of ..

3.6.11 OPA Data mapping

OPA is flexible enough to handle data mapping between different data models, enabling seamless integration with external systems. In our GreenOps system, data mapping is essential for translating between ElectricityMaps regions and cloud provider regions.

At some point in the system this mapping needs to be done. this mappings are needed since the scheudler knows only about ElectricityMaps regions, and do not possess the knowledge of cloud provider regions. Therefore, a mapping is needed to translate the ElectricityMaps regions to cloud provider regions and vice versa.

inside the policy is a good place to do this mapping

first filter is the selection of the provider this determine the whole set of regions belonging to that provider

eligible regions: this filter can be only done with cloud provider specific latencies

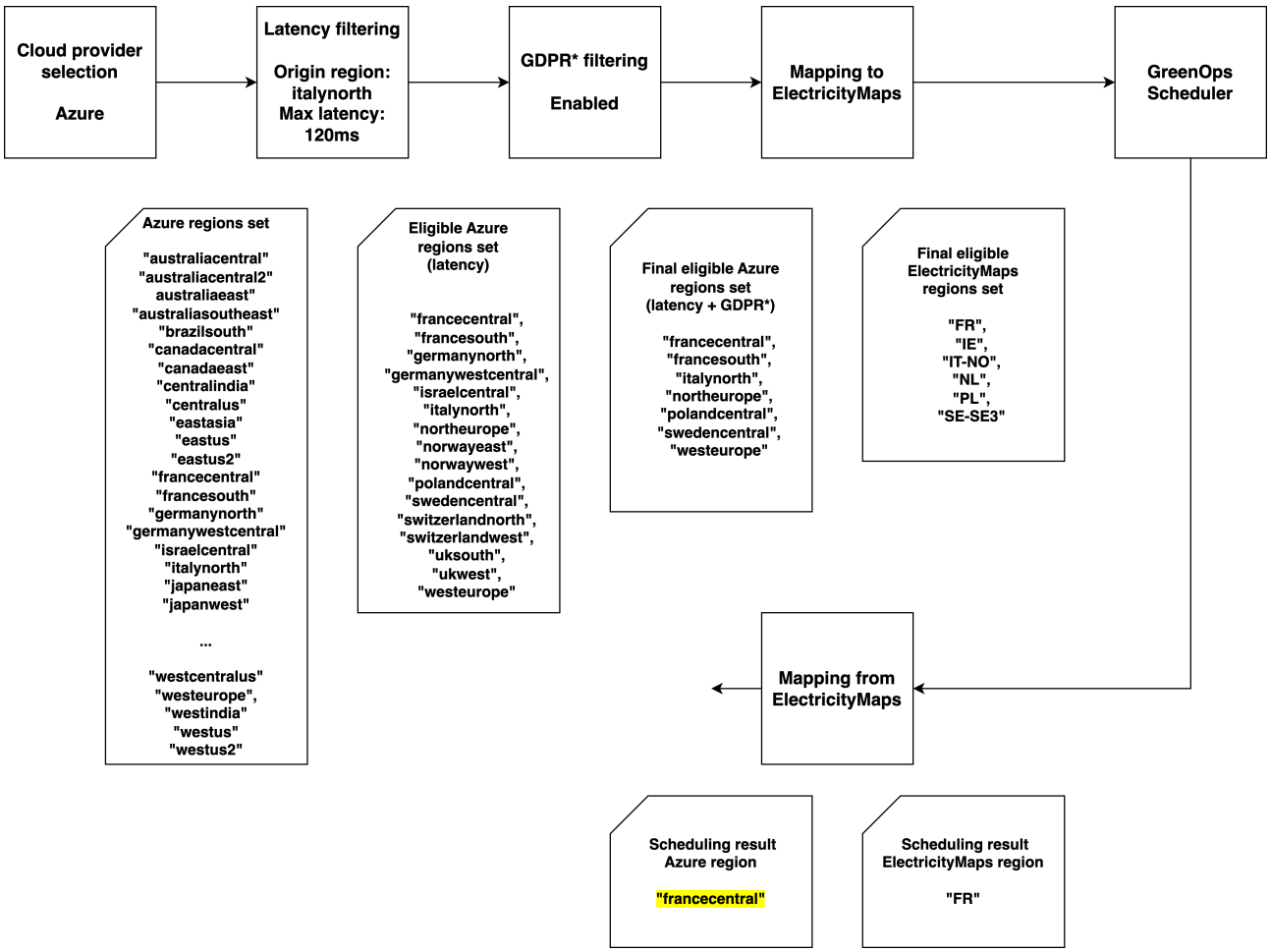

Figure 3.12: OPA Data mapping

```
# Utility functions to map between cloud provider regions
# and ElectricityMaps regions

map_to_electricitymaps(eligible_regions, provider) = em_regions if {
    em_regions := {
        region.ElectricityMapsName |
        some eligible_region;
        some region;
        eligible_region = eligible_regions[_];
        region = data[provider].cloud_regions[_];
        region.Name == eligible_region
        region.ElectricityMapsName != ""
        region.ElectricityMapsName != "Unknown"
    }
}

map_from_electricitymaps(em_region, provider) = cloud_region if {
    some region;
    region = data[provider].cloud_regions[_];
    region.ElectricityMapsName == em_region;
    cloud_region := region.Name
}
```

Listing 3.13: Rego data mapping

3.6.12 OPA end-to-end workflow

(K8s mutating webhook)

OPA flow:

- admission review (contains max_latency, origin_region)
- policy contains cloud provider (or chose for the user)
- policy calculate subset of eligible regions
- policy will ask scheduling information to the scheduler (using http.send()
relationship with k8s mutating webhook
rego policies
scheduler has notions of electricity maps regions only
OPA is used also as a data mapping layer both at request time and at response time

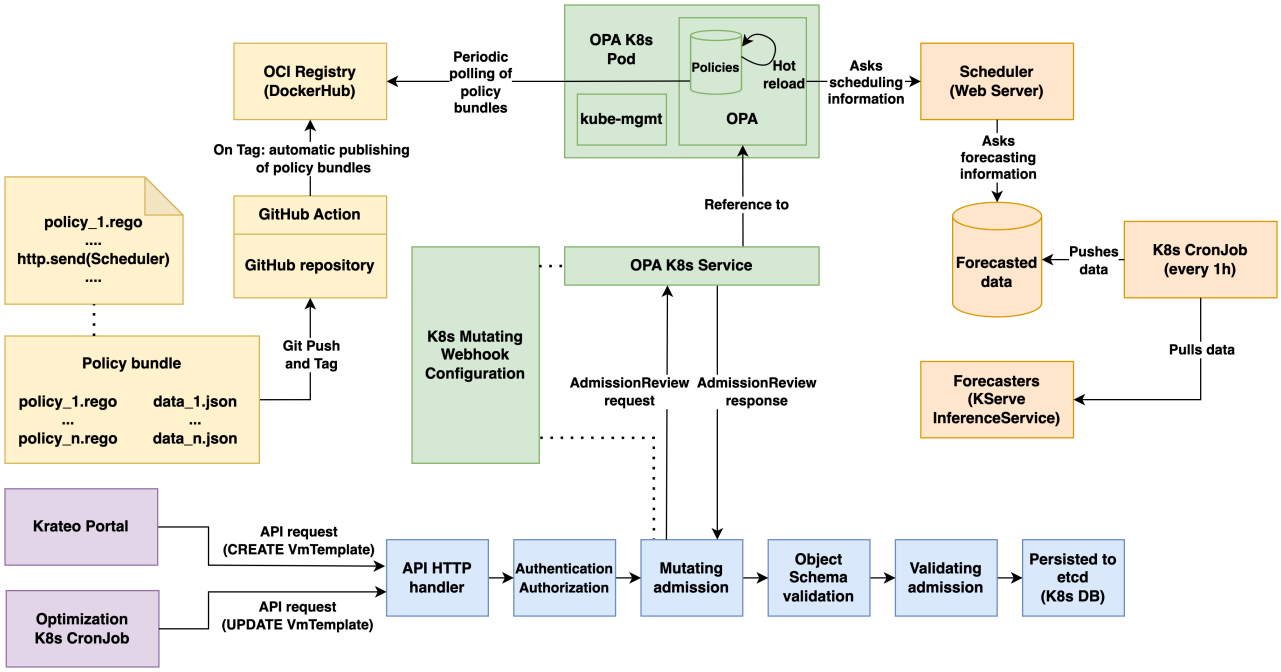

Figure 3.13: General architecture

Figure **??**represents the configuration of the Kubernetes Mutating Webhook with the intergeation of Open Policy Agent. In particular,

Day 2 operations what are day 2 operations: in this case (VMs) resize a VM (scale up or down) based on the load The mutating webhook configuration is set on the CREATE and UPDATE operations

UPDATE operation trigger K8s Cronjob that attach a label to the custom resoruce

3.6.13 OPA advanced features

It is deemed useful to mention some of the advanced features of OPA that were not employed in the system described in this thesis but could be potentially useful in future developments or in other contexts where OPA is used.

- bundle signing - delta bundles

3.7 MLOps infrastructure

MLOps is the abbreviation of "**Machine Learning Operations**", and it broadly refers to a set of methods designed to improve workflow procedures and automate machine learning deployments. It enables the reliable and efficient management, maintenance and deployment of models at scale [**?**]. A MLOps infrastructure is not necessarily required for multi-cloud resource management, but it is believed that AI models will be utilized in the future more and more to get **scheduling and management decisions**, as evidenced by recent studies discussed in section **??**. It is therefore deemed important to describe the MLOps infrastructure deployed in a Kubernetes environment and leveraged by the system described in this thesis.

3.7.1 MLOps purpose

In a way, MLOps implements DevOps principles, tools and practices into typical Machine Learning workflows. Its main purpose is to effectively industrialize the machine learning models lifecycle, enabling faster model development, selection, and deployment to production compared to traditional manual approaches. Some principles of MLOps can be summarized as follows [**?**]:

- **Automation**: automate the entire model lifecycle, from training to deployment.
- **Versioning**: track and version models (with related data and code).
- **Reproducibility**: ensure that various model versions can be reproduced at any time.
- **Monitoring**: monitor models in production to ensure they are performing as expected.
- **Scalability**: scale models to handle increased workloads in a seamless manner.
- **Collaboration**: enable collaboration between data scientists, data engineers, and operations teams.

In the context of the proposed system, the MLOps infrastructure is used to deploy and manage the forecasting models that predict the carbon intensity of the electricity grid in different world regions. In particular, **MLflow** is used for model tracking, model selection, and model storage, while **KServe** is used for model deployment.

3.7.2 MLflow

MLflow is an open-source platform designed to facilitate and enhance the overall machine learning lifecycle. In particular, it offers tooling for **experiment tracking**, **model management** and **model storage**. It is compatible with various ML frameworks, including scikit-learn, PyTorch, TensorFlow, and XGBoost. In particular, in the case of our system, PyTorch is the ML framework used for the forecasting models making MLflow a suitable choice for model management. For what concerns model deployment, MLflow does not provide a built-in solution, but it is compatible with other tools like KServe which will be described in section **??**.

MLflow Tracking Server

MLflow Tracking Server is the central component of the MLflow platform which is responsible for logging and storing model training data, parameters, and metrics. It enables ML practitioners to track experiments, compare results, and reproduce models easily in a collaborative environment. MLflow Tracking Server revolves around the concepts of **experiments**, which are collections of **runs**. Each run represents a single model training session, and it contains information such as parameters, metrics, and artifacts (e.g., model files). Selected models can then be registered in the MLflow Model Registry, which is essentially an optional subset of the selected models that are ready for deployment. Model selection can done with the aid of a user interface that allows ML practitioners to visualize and compare the results of different experiments, making it easier to select the best model for deployment. MLflow Tracking Server, as the name suggest, is a server that is constantly waiting for new data to be logged. Said data comes from the various training scripts that are executed by the data scientists or machine learning engineers which are run in training environments. In particular the training scripts must be instrumented with the **MLflow API calls** to log the data in the remote MLflow Tracking Server. This setting is very flexible since the training scripts can be run in any environment (e.g.,

local environment, Universities' HPC clusters), as long as they have access to the MLflow Tracking Server.

It is deemed important to mention some of the MLflow API calls that are used in the training scripts: the *infer_signature* and *autolog* functions.

The *infer_signature* function captures the **input and output schema of the model**, which is important for the deployment of the model. The *autolog* function is used to automatically log the parameters and metrics of the model training session, without the need to manually log them. In particular, each supported ML framework has its own autolog function which is used to automatically log the parameters and metrics of the model training session.

The final phase of a model training session is the automatic creation of a **self-contained directory**, named after the experiment and run IDs, that contains all the necessary files to deploy the model. In particular, the folder contains: the serialized model with model weights, the configuration files (conda.yaml, requirements.txt), and the MLmodel file that contains additional configuration, among which, the model signature. The self-contained folder is then automatically uploaded to the MLflow Tracking Server as an artifact. The following is an example of the structure of the self-contained directory created by MLflow after a PyTorch model training session.

```
model/
├── MLmodel
├── conda.yaml
├── python_env.yaml
├── requirements.txt
└── data/
    ├── model.pth
    └── pickle_module_info.txt
```

MLflow deployment configuration

In the context of our system, MLflow is deployed on a Kubernetes cluster with a loosely coupled architecture as can be seen in Figure **??**, where the MLflow Tracking Server is decoupled from its storage: the metadata store (backend store) and the artifact store. This configuration is the most common in production environments, as it allows for better scalability and environment flexibility. The metadata store chosen for the system is **CrateDB**, while the artifact store is the **SeaweedFS** object storage service.

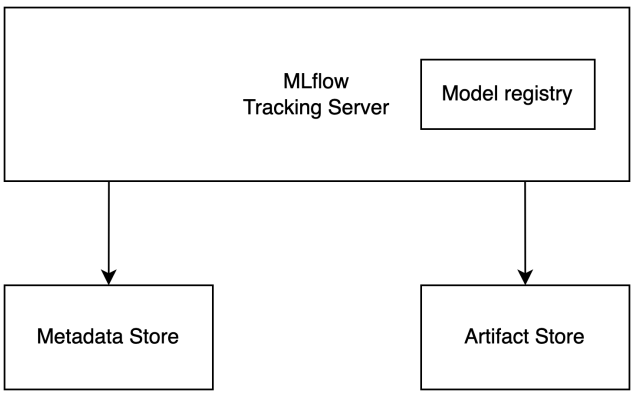

Figure 3.14: MLflow deployment configuration

During the design phase, **alternative configurations** were considered. One of the alternative configurations was to use a single CrateDB instance as both the metadata store and the artifact store. This configuration was not implemented due to the lack of native support of object storage in CrateDB. A second theorized approach was to use a sidecar container with the duty of watching

the MLflow Tracking Server local directory in the container filesystem (e.g., using *watchdog* Python package), packaging the model artifacts as an OCI image, and uploading them to an image registry. This approach could be implemented to offer an alternative to the MLflow Tracking Server artifact store in environment where adding a new storage service is not feasible.

3.7.3 KServe

KServe is an open-source model inference platform that extends Kubernetes with a set of Custom Resource Definitions (CRDs) to **deploy** and scale machine learning models in production environments. KServe can be deployed in several ways, one of which is the "Serverless mode" which is built on top of Istio and Knative, leveraging their powerful capabilities such as automatic scaling. There are two main CRs that can be used to set up a model serving environment: ***ServingRuntimes*** (or ***ClusterServingRuntimes***) and ***InferenceServices***. The former are abstractions that define model serving environments, specifying the templates for Kubernetes Pods capable of serving particular model formats. The latter are actually leveraging the available ServingRuntimes to deploy the models in the system. KServe provides several out-of-the-box ClusterServingRuntimes for common model formats, such as TensorFlow, PyTorch, and XGBoost, which can be used to deploy models without the need to define and configure the runtimes themselves. In particular, in our specific use case, since the forecasting models are PyTorch models, packaged by MLflow as described in the previous section, InferenceServices with "kserve-mlserver" ClusterServingRuntime are used. As a matter of fact, this runtime is the one that supports models packaged with MLflow. Listing **??** shows an example of the InferenceService Custom Resource used to deploy a model in the system.

```yaml
apiVersion: "serving.kserve.io/v1beta1"
kind: "InferenceService"
metadata:
  name: "forecaster-1"
  namespace: "model-inference"
spec:
  predictor:
    serviceAccountName: sa-s3creds
    model:
      modelFormat:
        name: mlflow
      protocolVersion: v2
      storageUri: s3://mlartifacts/forecaster-1
```

Listing 3.14: InferenceService Custom Resource example

Open Inference Protocol

Interoperability is key in a fast-moving environment as the one of machine learning and AI. Therefore KServe has introduced the Open Inference Protocol specification to standardize the communication between inference servers and clients. The Open Inference Protocol has been adopted by several inference servers, including NVIDIA's Triton and Seldon MLserver

API	**Verb**	**Path**
Inference	POST	v2/models/[/versions/<model_version>]/infer
Model Ready	GET	v2/models/<model_name>[/versions/]/ready
Model Metadata	GET	v2/models/<model_name>[/versions/<model_version>]
Server Ready	GET	v2/health/ready
Server Live	GET	v2/health/live
Server Metadata	GET	v2

Model deployment

Our specific use case requires the deployment of multiple models, each corresponding to a different region. This is true if we want to achieve better model performance compared to a single model that tries to predict the carbon intensity of all the regions. The current strategy adopted for the system is the following: one model per region is deployed, and a generic model is used as a fallback if the specific model is not available.

This translates into the deployment of multiple InferenceServices, each corresponding to a specific region, and one InferenceService that acts as a fallback. This setting introduces a quite large amount of overhead in terms of resources, since each InferenceService set up a whole new set of resources (e.g., large Kubernetes Pods with underlying model serving environments) for each model. Figure **??** illustrates the configuration of the InferenceServices used to deploy the forecasting models in the system.

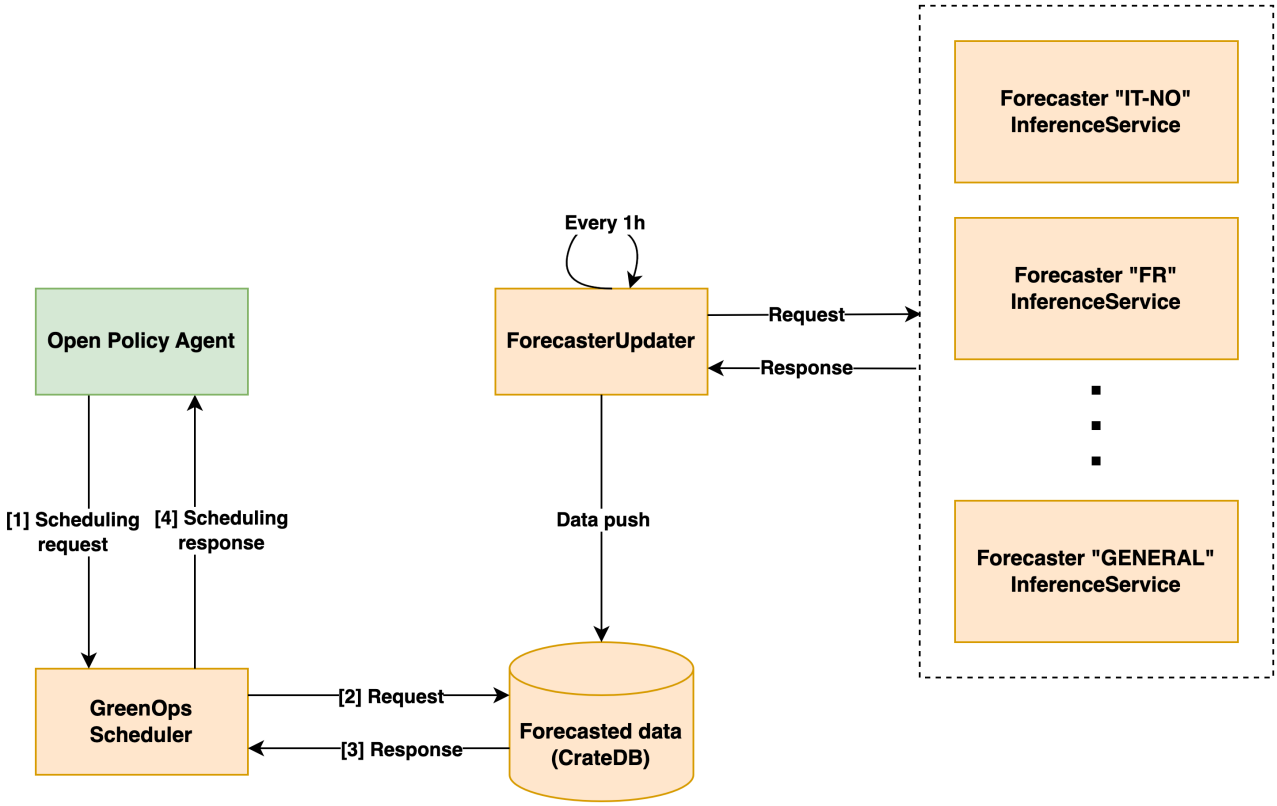

Figure 3.15: Model deployment configuration

Model compatibility

In scenarios where ML models produce outputs in formats not directly compatible with the serving runtime, some workarounds are necessary, such as model wrapping. This process involves adapting the model's input and output interfaces to align with the expected formats of the serving runtime, ensuring a correct model deployment. For instance, in the case of the forecasting models, the output format was not directly compatible with the serving runtime: the output was a custom defined class specific to the model, instead of a single tensor. Therefore, a simple model wrapping was needed to make the model compatible with the serving runtime, effectively operation a selection of output fields from the model output and returning them as a single tensor.

3.7.4 MLOps general architecture

Figure **??** illustrates the general architecture of the MLOps infrastructure deployed in the Kubernetes environment. The architecture consists of two main components, described in the previous sections: MLflow and KServe. In particular, MLflow is used for model tracking, model selection, and model

storage, while KServe is leveraged for model deployment.

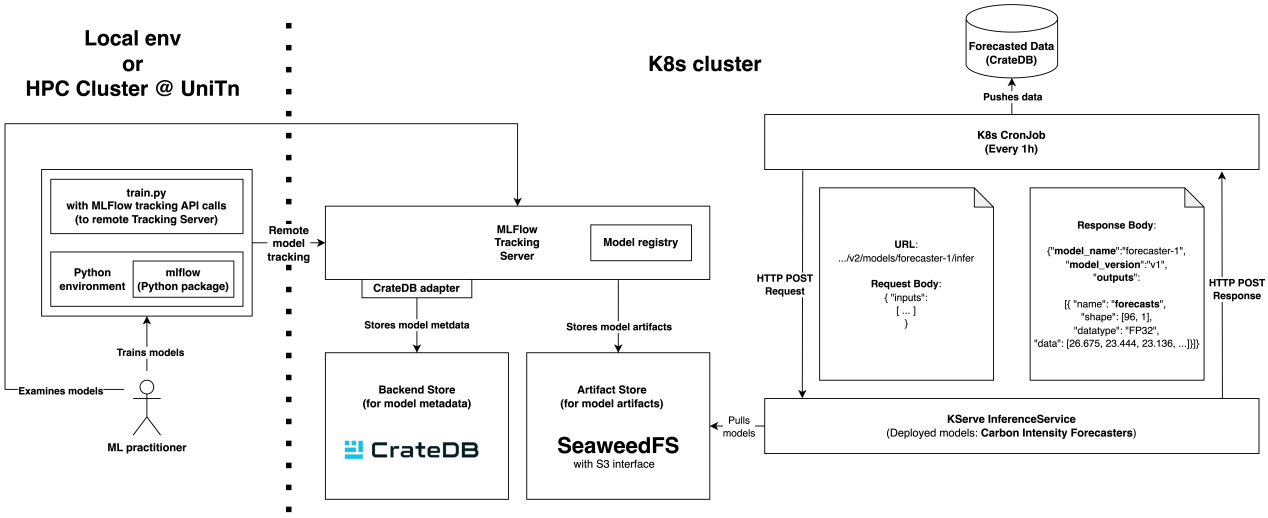

Figure 3.16: MLOps Architecture

3.8 Measurements

3.8.1 System / performance metrics

how to measuere cloud resource systems

especially useful for "day 2 operations" scaling down a VM

Metrics collected could be: CPU usage memory usage. These metrics are especially useful for the 2nd use case for instance: scaling down a VM.

Prometheus exporters (https://prometheus.io/docs/instrumenting/exporters/) + Prometheus scrapers for data collection. Generic Prometheus exporters and scrapers already used for Krateo Composable FinOps leveraging specific K8s Custom Resources. These exporters are generic and can scrape arbitrary metrics configured in specific CRs (for instance, collecting VMs CPU consumption though Azure APIs). From the Krateo Composable FinOps document: - "we transform all optimizations into a set of Kubernetes Custom Resources (CRs) to act upon newly found cost-related deficiencies. This allows us to use Kubernetes operators (explicitly coded to interact with cloud services) to monitor these metrics and act automatically to apply changes to remote resources."

- "forward the optimization to the Krateo operator that manages the services that need to be optimized, for example, the Azure Operator to modify the size of a Virtual Machine;"

- "the optimization is automatically encoded in a CR for the finops-operator-vm-manager, which then analyzes it and decides how to manage the Virtual Machine. For example, it could scale up or down the virtual machine, stop it for the night, etc."

From my current understanding, only Azure is available for now on the finops-operator-vm-manager. This operator is only able to: start; stop; deallocate; scale-up; scale-down. So finops-operator-vm-manager operates on already provisioned virtual machines and it applies optimizations.

—

Cloud providers API ("hypervisor" / host level) to get CPU usage, memory usage Azure monitoring REST API: https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/rest-api-walkthrough?tabs=resthttps:
us/rest/api/monitor/metrics/list?view=rest-monitor-2023-10-01tabs=HTTP https://learn.microsoft.com/en-us/azure/virtual-machines/monitor-vm

Cloud providers agents Azure Monitor Agent (https://learn.microsoft.com/en-us/azure/azure-monitor/agents/azu
monitor-agent-overview) Google Ops Agent: "the primary agent for collecting telemetry from your Compute Engine instances" (https://cloud.google.com/monitoring/agent/ops-agent)

Standard agents/deamons manually installed the VMs (e.g. Prometheus node exporter, many metrics restricted by public cloud providers not straightforward to automate the deployment

—

Challenges Public Cloud Providers do not provide data about carbon intensity of a VM instance Public Cloud Providers do not provide data about power consumption of a VM instance

Power consumption metrics scaphandre: NOT supported by public cloud providers. "Public cloud providers do not expose the underlying RAPL sensors that scaphandre and other measurement tools rely on to track consumption". (https://github.com/hubblo-org/scaphandre/issues/142) https://hubblo-org.github.io/scaphandre-documentation/index.html

kepler: really interesting and quite mature project but works only with K8s resources inside the cluster (Nodes, Pods). Therefore not good for our use first case. https://sustainable-computing.io/design/architecture

manually estimate power consumption based on CPU utilization, memory usage. Could be a very difficult task. there is the TEADS metodology like

—

Carbon metrics: there is no adopted standard, there is not something similar to FOCUS yet; there is a proposal for a specification (work in progress, not supported yet by Public Cloud Providers): https://github.com/Green-Software-Foundation/real-time-cloud https://github.com/Green-Software-Foundation/real-time-cloud/blob/main/Cloud$_{R}egion_{M}etadata_{s}pecification.md$

Public Cloud Providers monthly reports (probably not useful in this case) Export Azure carbon optimization emissions data (Preview) (probably not fine-grained as we want)

Cloud Carbon Footprint Uses cloud provider billing (AWS Cost and Usage Reports with Amazon Athena, GCP Billing Export Table using BigQuery, Azure Consumption Management API). Using these services costs. Electricity Maps API integration is supported (for live grid carbon intensity)

aether calculation engine (https://aether.green/docs/methodologies/) only AWS and GCP supported, Azure not yet uses AWS CloudWatch and Google monitoring very small project no live Grid Carbon Intensity Coefficient, they extrapolate data from "governative" data reports https://aether.green/docs/r carbon-intensity-coefficient

carbond agent) (https://gitlab.com/sustainable-computing-systems/carbond) installed on the machine

Example of a manual approach (not scalable due to tech specs research): https://devblogs.microsoft.com/sus software/how-can-i-calculate-co2eq-emissions-for-my-azure-vm/

The critical point here is to get/calculate the energy consumed by a cloud instance, since there are a huge number of technical configurations to find, retrieve and use for calculations.

3.8.2 Impact framework

Impact framework (by green software foundation)

3.9 End-to-End workflow

[TO BE CHANGED] A user request for a workload arrives in Kubernetes / Krateo. The request shape is VM1 = (MinCPU=4vCPU, MinRAM=4GiB, D=12h) As we can see the request is generic: it does not contain a specific cloud provider or a specific cloud region. A K8s Custom Resource (CR) representing the workload is created. mutating webhook intercepts the CREATE API request. K8s mutating webhook retrieve and evaluate policies and in particular, the AI model inference is called and should return a decision with the region and scheduling time (time-shifting and geographical shifting). OPA will use this decision to mutate the VM specification, adding the provider, the schedulingRegion and schedulingTime fields. Krateo core provider / cdc

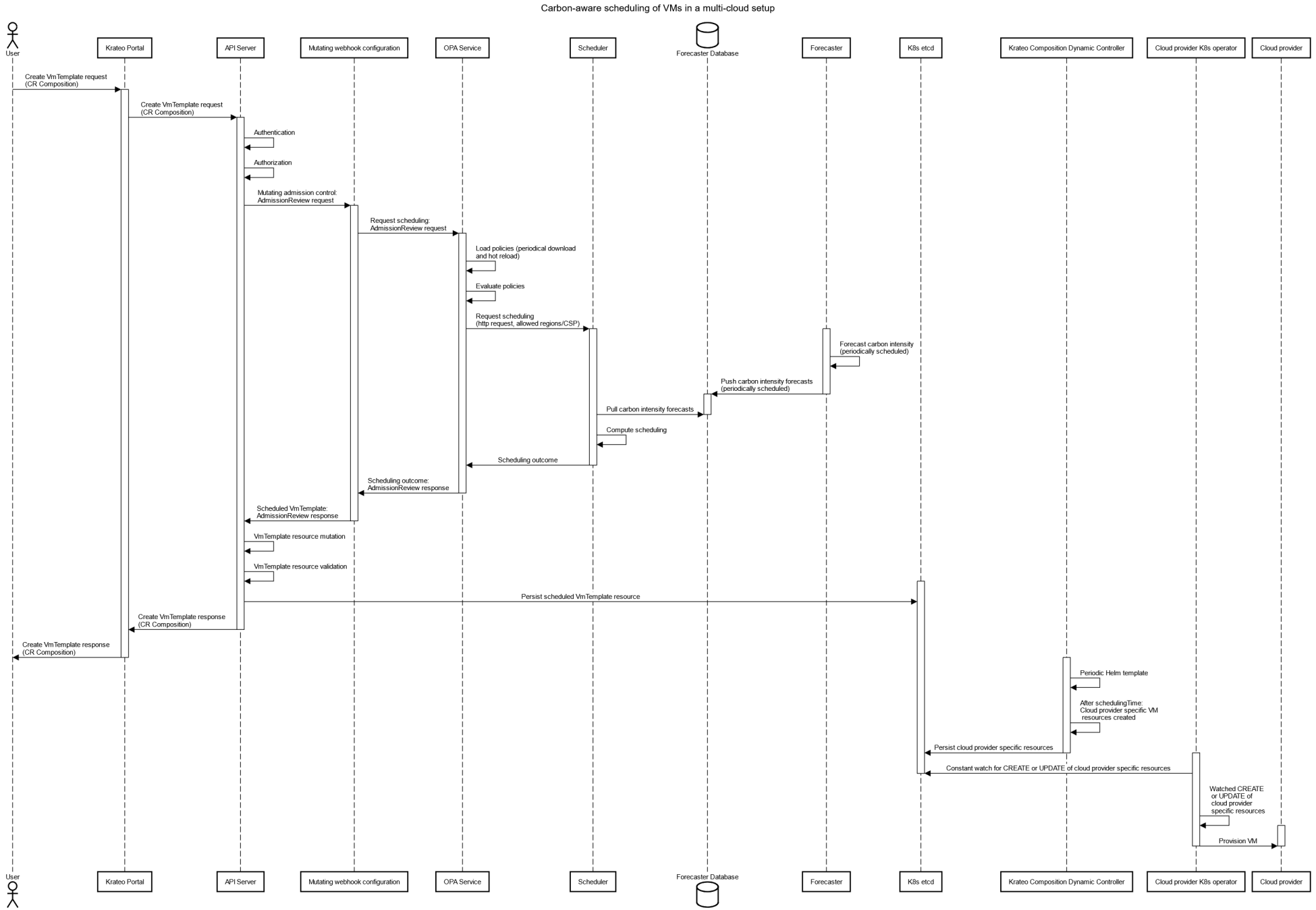

Figure 3.17: Example of a full-page rotated image

Table for recap of all tools used

Kubernetes - Krateo Helm Helm charts Helm templating Helm lookup function

- VmTemplate Krateo Composition Definition - Azure K8s Operator - GCP K8s Operator - AWS K8s Operator - K8s mutating webhook configuration - OPA server - opa policies - OPA bundles - MLflow tracking server (+ metadata store artifact store) - Forecaster (deployed as KServe InferenceService)

4 Discussion

4.1 End-to-end integrated test

final result A comprehensive end-to-end integrated test has been carried out on a Kubernetes cluster this was used to validate the system

4.2 GreenOps system evaluation

4.2.1 Theoretic upper bound

(how close can we get, masachussets amherest group)

4.2.2 Baseline definition

We should prepare one or more baseline schedulings that will be used as a baseline and compared with a carbon-aware scheduling proposed by our system.

4.2.3 Black hole phenomenon

How to deal with the so-called "Black hole" phenomenon? That is, if 100 workload scheduling arrives at some point, there is the possibility that the outcome of the system we are building is: "schedule all workloads in Norway" where Norway is the region with least carbon intensity at that moment. This phenomenon came up also in a previous meeting but it is not clear if this could be a problem etc.. A probable differentiator could be the max latency field of the workload request. Other service requirements could contribute to this as well.

(how it is countered)

4.2.4 Side effects

Maybe out of scope of this work, side effects, big picture. What happens if a big percentage of companies that relies on cloud services starts to adopt carbon-aware scheduling of their workloads? We tend to image cloud providers or even cloud regions as an infinite pool of resources, and at a certain level it is almost like that. But could carbon aware scheduling have larger, not foreseen, side effects? Is this a responsibility of who schedules? Shall schedulers be responsible for the load on regions? Like self-imposing some sort of limits/caps.

4.2.5 Preliminary evaluation

for the purpose of this theses

boavizta API simulation

assumptions - analysis limited to only cloud VM, (aligned with the scope of this theses) - data related to GCP is not data from boavizta (even if gcp is supported in our current system) but mapped from azure and aws

limitations - whole countries, not regions

not easily integratable in a real production system due to its quite restrictive license (AGPL 3) it is still usable for research purposes like in this case.

5 Conclusion

production-ready system

5.1 Future improvements

day2 operations we are ready for this

Scaling down a VM (example of Day 2 operations) From: (4 vCPU, 8 GiB RAM) To: (2 vCPU, 4 GiB RAM)

This use case is meaningful for workloads with durations in the order of at least days. Otherwise, for short-lived workload this use case does not make sense. And in the case of workloads with days as duration, time and geographical shifting is not that relevant.

This use case will leverage system and performance metrics.

support for other resources we need templates operators

5.1.1 Multi model serving

"The original design of KServe deploys one model per InferenceService. But, when dealing with a large number of models, its 'one model, one server' paradigm presents challenges for a Kubernetes cluster."

kserve model mesh instead of several InferenceService there is a lot of overhead in the current configuration

how much is better to use more models instead of one generic model

Bibliography

[1] Amazon machine images (amis). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html. Last access: 10/01/2025.

[2] Azure data center information. https://gist.github.com/lpellegr/8ed204b10c2589a1fb925a160191b974. Last access: 15/02/2025.

[3] Electricity maps. https://app.electricitymaps.com/. Last access: 15/02/2025.

[4] Helm. https://helm.sh/. Last access: 01/02/2025.

[5] Helm template functions and pipelines. https://helm.sh/docs/. Last access: 01/12/2024.

[6] How kubernetes works. https://www.cncf.io/blog/2019/08/19/how-kubernetes-works/. Last access: 10/02/2025.

[7] Json patch. https://jsonpatch.com/. Last access: 10/02/2025.

[8] Krateo composition dynamic controller documentation. https://docs.krateo.io/key-concepts/kco/composition-dynamic-controller. Last access: 17/01/2025.

[9] Krateo core provider documentation. https://docs.krateo.io/key-concepts/kco/core-provider. Last access: 17/01/2025.

[10] Krateo oasgen provider. https://docs.krateo.io/key-concepts/kog/oasgen-provider. Last access: 15/02/2025.

[11] Krateo platformops documentation. https://docs.krateo.io/. Last access: 15/02/2025.

[12] kube-mgmt. https://github.com/open-policy-agent/kube-mgmt. Last access: 05/01/2025.

[13] Kubernetes dynamic admission control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/. Last access: 15/02/2025.

[14] Kubernetes mutating webhook demo (image reference). https://www.youtube.com/watch?v=Eb9pMSCTDjI. Last access: 13/02/2025.

[15] Kubernetes sidecar containers. https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/. Last access: 05/01/2025.

[16] Opa documentation. https://www.openpolicyagent.org/docs/latest/. Last access: 28/12/2024.

[17] Opa gatekeeper external data. https://open-policy-agent.github.io/gatekeeper/website/docs/externaldata. Last access: 05/12/2024.

[18] Opa philosophy. https://www.openpolicyagent.org/docs/latest/philosophy/. Last access: 28/12/2024.

[19] Opa policy bundles. https://www.openpolicyagent.org/docs/latest/management-bundles/. Last access: 16/01/2025.

[20] Open container initiative. https://opencontainers.org/. Last access: 12/12/2025.

[21] Opennebula. https://opennebula.io/discover/. Last access: 27/01/2025.

[22] A practical guide to getting started with policy as code. https://aws.amazon.com/it/blogs/infrastructure-and-automation/a-practical-guide-to-getting-started-with-policy-as-code/. Last access: 10/01/2025.

[23] What is mlops? https://ubuntu.com/blog/what-is-mlops. Last access: 15/02/2025.

[24] Przemyslaw Pawluk, Bradley Simmons, Michael Smit, Marin Litoiu, and Serge Mankovskii. Introducing stratos: A cloud broker service. 06 2012.

[25] Jose Luis Lucas Simarro, Rafael Moreno-Vozmediano, Ruben S. Montero, and I. M. Llorente. Dynamic placement of virtual machines for cost optimization in multi-cloud environments. In *2011 International Conference on High Performance Computing Simulation*, pages 1–7, 2011.

[26] Abel Souza, Shruti Jasoria, Basundhara Chakrabarty, Alexander Bridgwater, Axel Lundberg, Filip Skogh, Ahmed Ali-Eldin, David Irwin, and Prashant Shenoy. Casper: Carbon-aware scheduling and provisioning for distributed web services. In *Proceedings of the 14th International Green and Sustainable Computing Conference*, IGSC '23, page 67–73. ACM, October 2023.

[27] Thanathorn Sukprasert, Abel Souza, Noman Bashir, David Irwin, and Prashant Shenoy. Spatiotemporal carbon-aware scheduling in the cloud: Limits and benefits. In *Companion Proceedings of the 14th ACM International Conference on Future Energy Systems*, e-Energy '23 Companion, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Bimlesh Wadhwa, Aditi Jaitly, and Bharti Suri. Cloud service brokers: An emerging trend in cloud adoption and migration. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 140–145, 2013.