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.

Contents

List of Figures

Abstract

test test test

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

1 Introduction

Intro intro

project divided into 3 parts

data part

ML part

infrastructure part

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 Personal contribution

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

In

exploratory data analysis (EDA) 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 GreenOps landscape

what is greenops

from greenops landscape itself

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

Green Software foundation

green software 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

the foundation also developed the Impact Framework which will be described in section XY

2.2 Cloud providers

2.2.1 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.

2.2.2 Multi cloud

(why, how to achieve)

advantages

reduces vendor lock-in

Why multi-cloud in this context? 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 the purpose of this work, we will consider only the 3 major Public Cloud Provider as of today: AWS, Azure, GCP

2.2.3 computational sustainability by 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 wiith 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".

2.3 Kubernetes

became the de-facto standard for container orchestration

2.3.1 Kubernetes as a platform

Kubernetes as a platform to manage external resources

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.

2.3.2 Kubernetes extendability

Operator paradigm

CRDs

2.4 Krateo PlatformOps

Krateo PlatformOps 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 but acts as a control plane even for resource external to the Kubernetes cluster. A requirement is that the resources needs to be descriptible using a YAML file which represents the desired state of the resource [**?**].

Recognized by Gartner Gartner... by 2025 companies without a ... (cite)

architecture, components Krateo main 3 components

Krateo Composable Operations Krateo Composable Portal Krateo Composable FinOps

The core-provider is a Kubernetes operator that downloads and manages Helm charts. It checks for the existence of a file named values.schema.json 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 Helm chart and it is considered a best practice for Helm charts.

helm charts as native resources

2.4.1 Helm

HELM

what is an helm chart

values.schema.json

as we can see the core provider deploys the cdc

2.5 State of the Art

An extensive analysis of existing systems have been made in order to...

2.5.1 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**

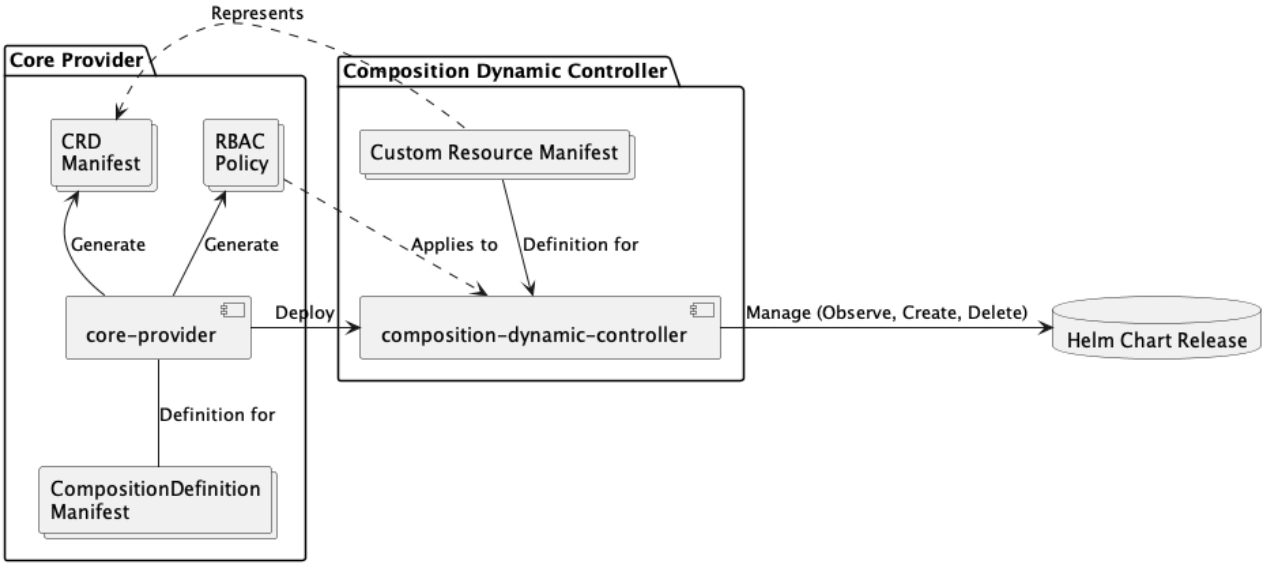

Figure 2.1: Krateo Core Provider architecture

Level Objectives (SLOs), highlighting its potential for practical implementation in large-scale distributed systems [**?**]. However, the system CANNOT BE CONSIDERED A REAL PRODUCTION SYSTEM.

2.5.2 CASPIAN

most important ptobably

2.5.3 Let'sWaitAwhile

test

2.5.4 Other systems

carbonScaler

2.5.5 SOTA Recap

many simulation, no real system no much flexibility

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

3.1 Project division???

Code repositories It is possible to find all the source code related to the project at...

4 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.

4.1 Assumptions

4.1.1 workload definition

In this work, workload has been modeled as **Virtual Machines (VMs)**, representing the primary use case considered during the system's initial design phase. [WHY THIS CHOICE?]

We define, for the purpose of this work, a workload as a Virtual Machine (VM). We can assume/imagine that inside of the VM, some kind of processing/work P will happen. This VM can be scheduled on any Public Cloud Provider since we are looking for a multi-cloud setting: cloud can be seen as a commodity. Alternatively, a subset of eligible Public Cloud Providers can be set as a policy. We can use general-purpose VMs.

The VM has the following characteristics: MinCPU needed MinRAM needed Duration (D): the time duration that a VM must run to accomplish its processing/work Deadline (DL): the time (timestamp) when the VM must have finish its work (maybe this can be omitted for the basic use cases) Max latency (ML): value in ms

If the workload is completely delay-tolerant then the Deadline parameter must not be used.

So a VM could be defined as a tuple like the following: VM = (MinCPU, MinRAM, D, DL, ML)

example

Min CPU: 4 vCPU Min RAM: 4 GB of RAM Duration (D): 1h Deadline (DL): 2022-12-31 23:59:59

—

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)**.

—

A limitation of our general approach is that only resources supported by the cloud provider's Kubernetes operator can be provisioned. 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 managed 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: different cloud providers expose varying levels of support for Kubernetes-native resource provisioning, leading to potential inconsistencies across multi-cloud environments.

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. Another approach could involve the development of custom operators or controllers to manage specific resource types not supported by existing operators. [KRATEO custom operator REST]

4.2 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	Intercepts and modifies API requests before they are persisted, allowing dynamic configuration adjustments with policy enforcement.
OPA Server	Evaluates authorization and policy decisions based on defined constraints and input data from Kubernetes API requests through the webhook.
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 4.1: 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 **??**.

4.3 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.

4.3.1 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 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 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.

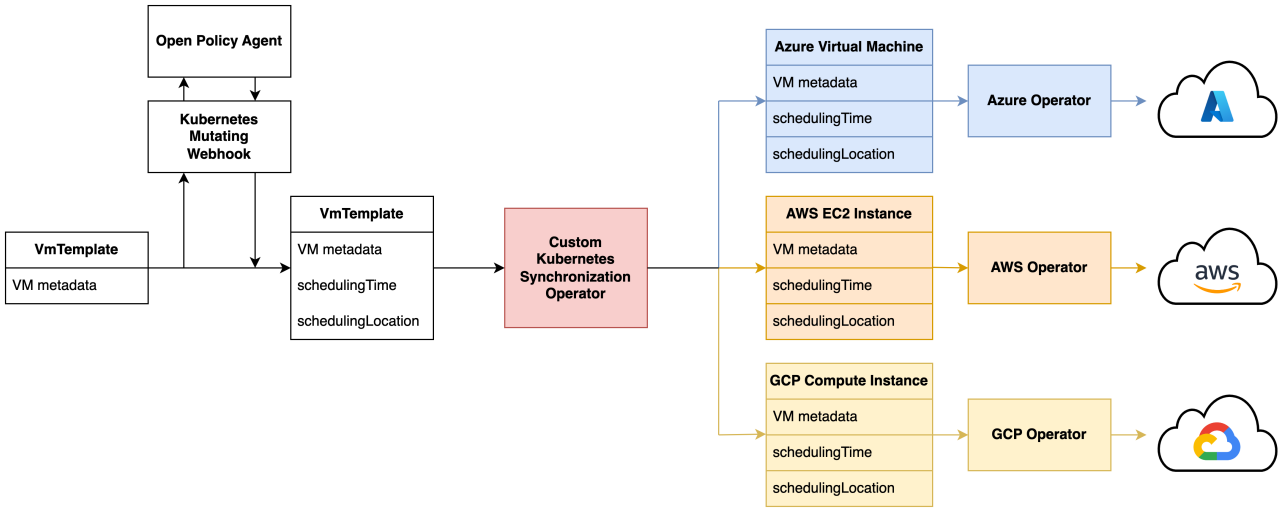

Figure 4.1: Multi-cloud resource management with Custom Kubernetes "Synchronization Operator" approach

4.3.2 Resource management: the Krateo approach

what is krateo developer platform already described in section **??**

Self-service platform for multi-cloud native resources

(generic VM mapped thanks to Krateo components, what is the added value)

generic workload resource definition how to define it

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

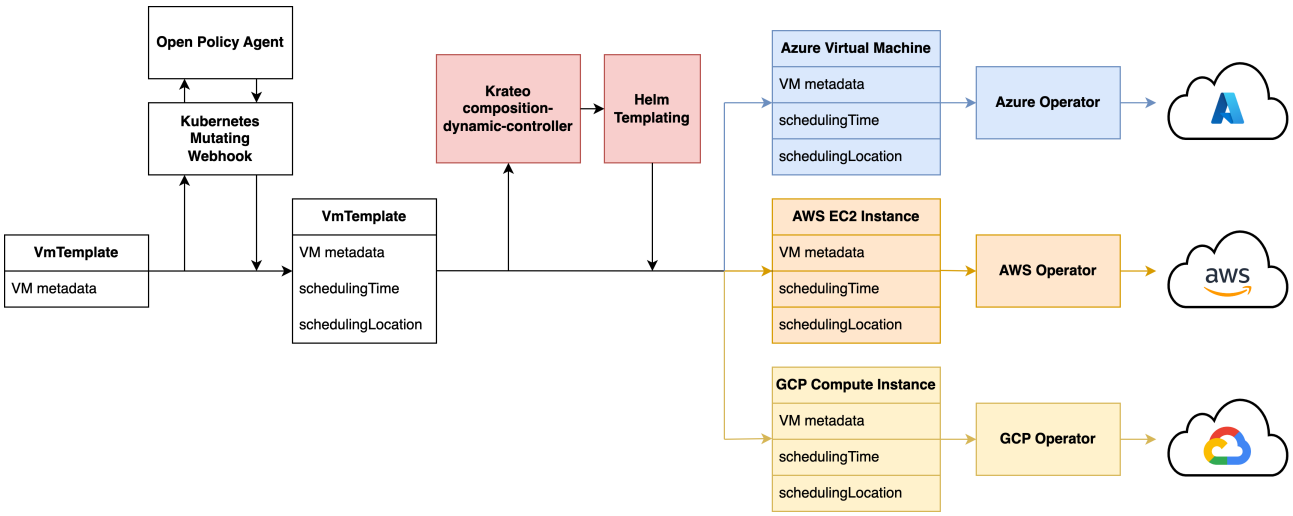

Figure 4.2: Multi-cloud resource management with Krateo PlatformOps approach

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

krateo cdc The composition-dynamic-controller is an operator that is instantiated by the core-provider to manage the Custom Resources whose Custom Resource Definition is generated by the core-provider.

4.4 Kubernetes Mutating Webhook

[from k8s docs]

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 semnatically different: the webhook configuration and the webhook service

[generic webhook image]

4.5 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)**.

4.5.1 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.

4.5.2 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 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.

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 baseline infrastructure for each cloud provider 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 providers has its complexities and nuances

vendor lock-in.

4.5.3 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

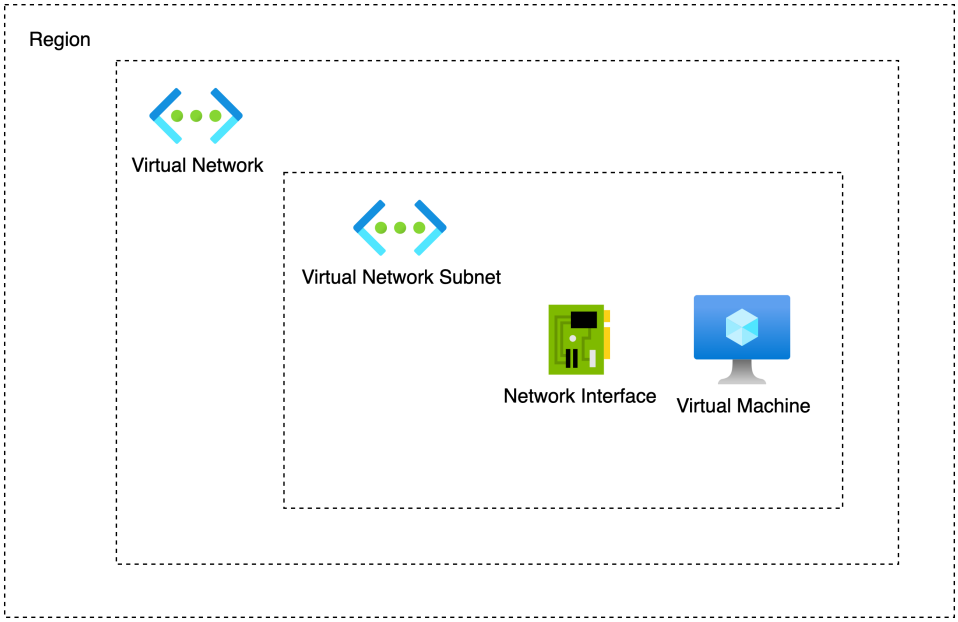

Figure 4.3: Minimum set of Azure resources for VM provisioning

INSTANCE CR example

4.5.4 GCP Operator

minimum set of resources needed for vm deployment

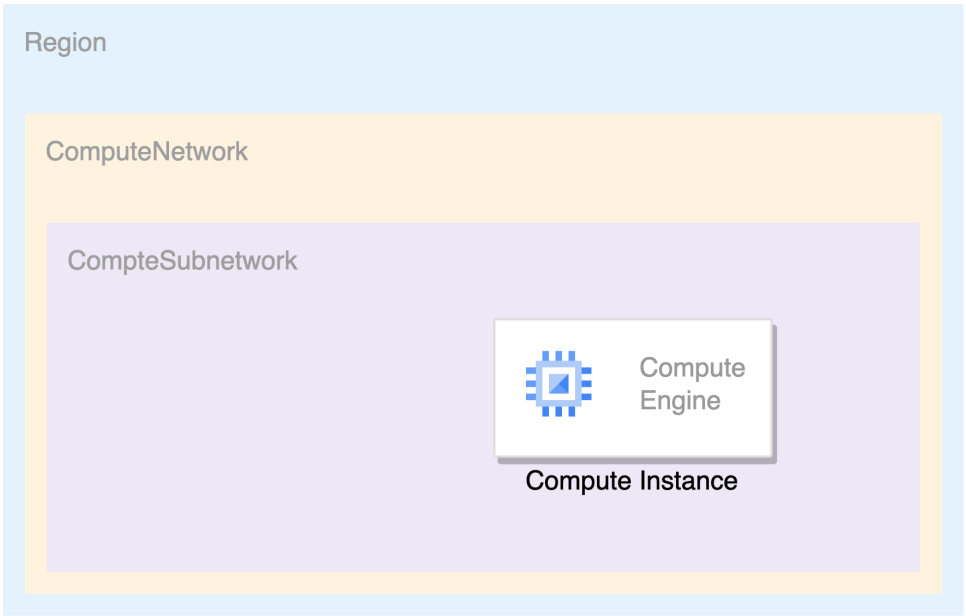

Figure 4.4: Minimum set of GCP resources for VM provisioning

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 resource needed

4.5.5 AWS Operator

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

- VPC
- Subnet
- EC2 Instance

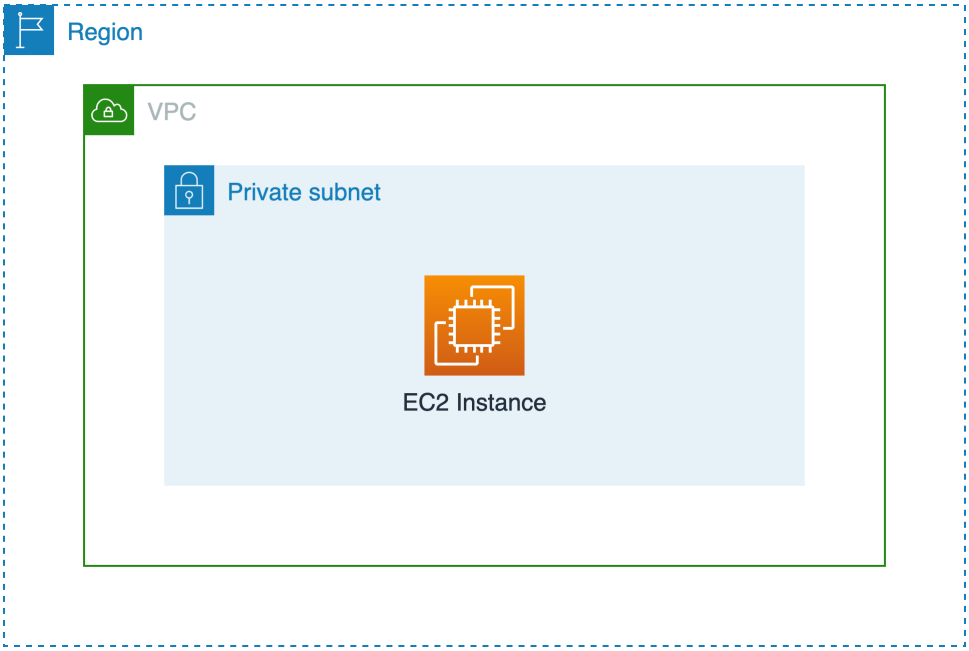

Figure 4.5: Minimum set of AWS resources for VM provisioning

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.

```
1 ...
2 apiVersion: ec2.services.k8s.aws/v1alpha1
3 kind: Instance
4 metadata:
5   name: {{ .Values.vmName }}
6   namespace: {{ .Values.namespace | default "greenops" }}
7 ...
8 spec:
```

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

Listing 4.1: 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 4.2: AWS Operator direct reference example

In this case, the Subnet Custom Resource manifest directly references the VPC Custom Resource using name and namespace since the Operator is designed to support this type of relationship. As explained before, this is determined by Operator design choices.

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).

4.6 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.

4.6.1 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 distributed 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.
- **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.

4.6.2 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.

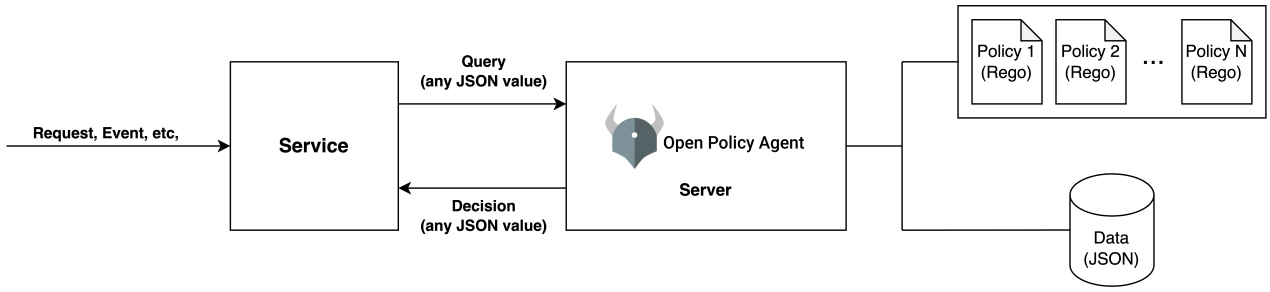

Figure 4.6: OPA architecture

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

4.6.3 OPA and external data

types of external data strategies

http.send() paramters

4.6.4 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

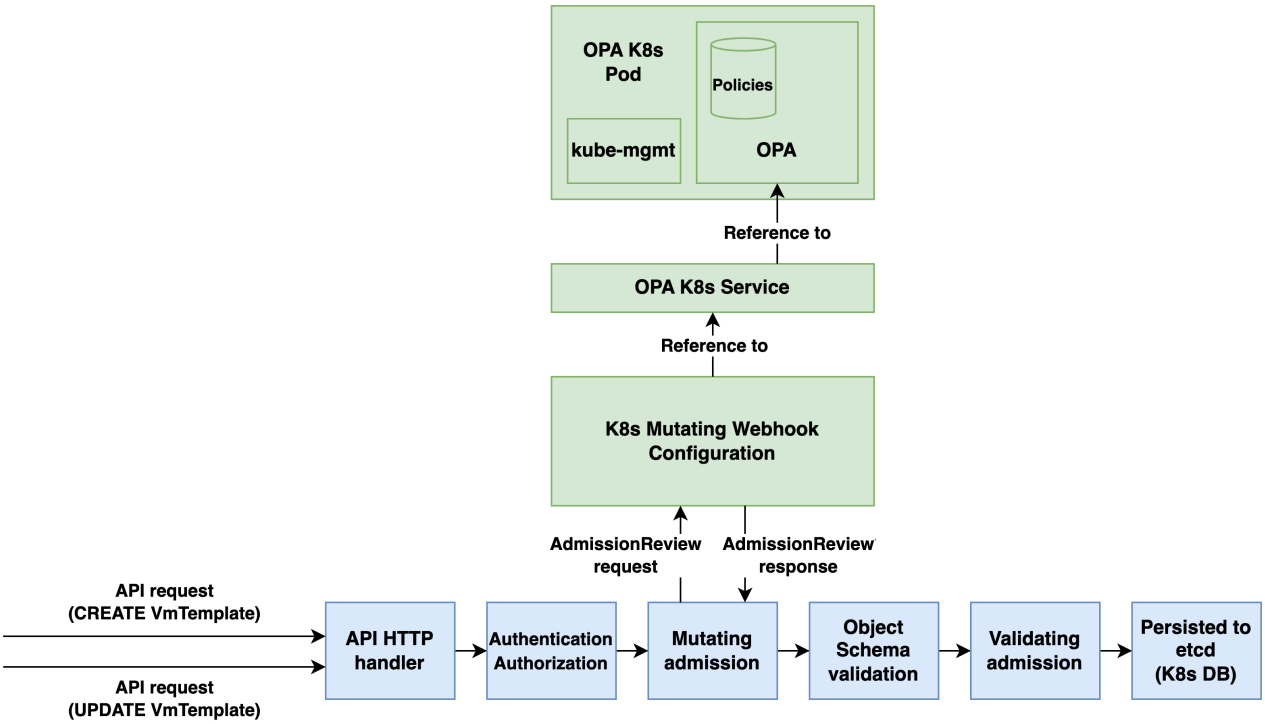

Figure 4.7: Kubernetes mutating webhook and OPA integration

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.

4.6.5 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 4.3: 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 4.4: 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 4.5: AdmissionReview request

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

Listing 4.6: 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.

4.6.6 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 **??**.

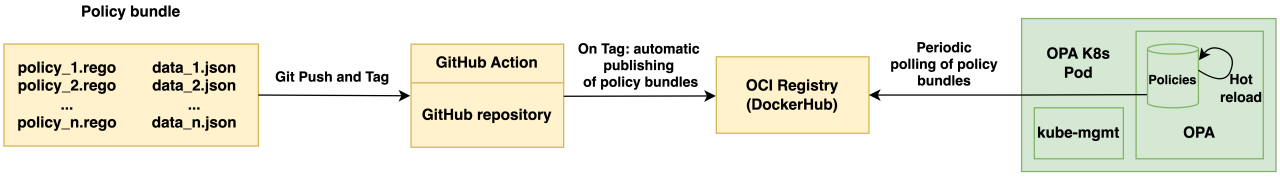

Figure 4.8: OPA policy bundles

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.

4.6.7 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 4.7: Simple OPA Rego Policy

The second example, illustrated in listing **??** utilizes OPA Gatekeeper, which extends OPA 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 4.8: OPA Gatekeeper ConstraintTemplate

```
apiVersion: constraints.gatekeeper.sh/v1beta1
```

```yaml
kind: PodNamespaceConstraint
metadata:
  name: restrict-namespace
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
  parameters: {}
```

Listing 4.9: OPA Gatekeeper Constraint

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 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.

4.6.8 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.

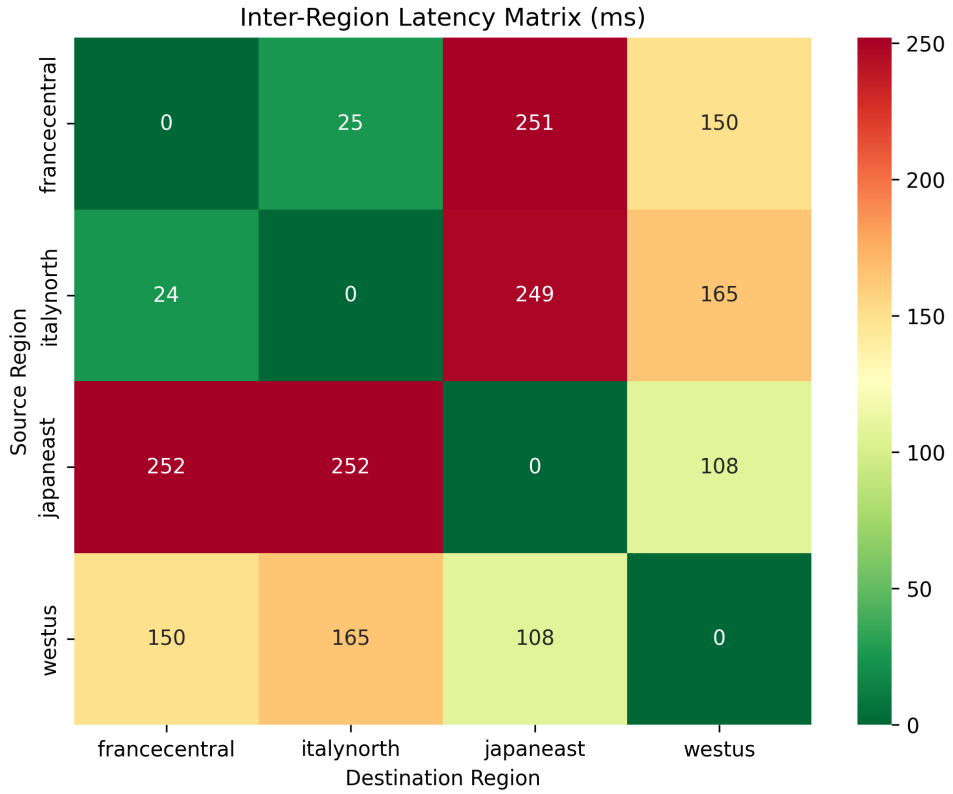

Figure 4.9: Latency matrix example

```
{
    "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 4.10: 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/networking/m

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

Azure network latencies docs: https://raw.githubusercontent.com/MicrosoftDocs/azure-docs/refs/heads/m
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)

google syntetic data

4.6.9 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.

4.6.10 Scheduling outcome policy

main policy

Mutation policy dedicated to

JSON patch, what is a jsonpatch patch code

encoded patches

4.6.11 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

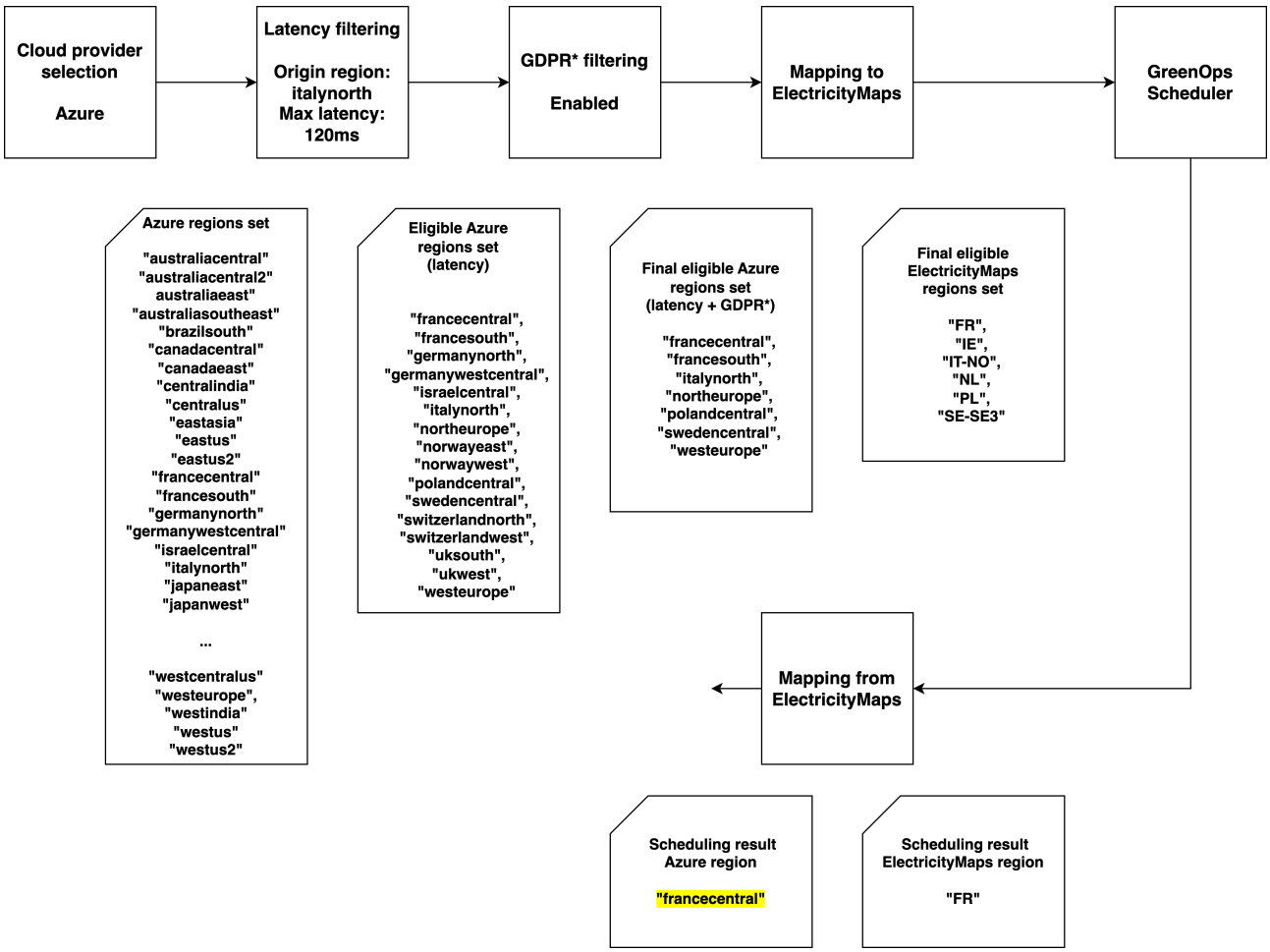

Figure 4.10: OPA Data mapping

```
# 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 4.11: Rego data mapping

4.6.12 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

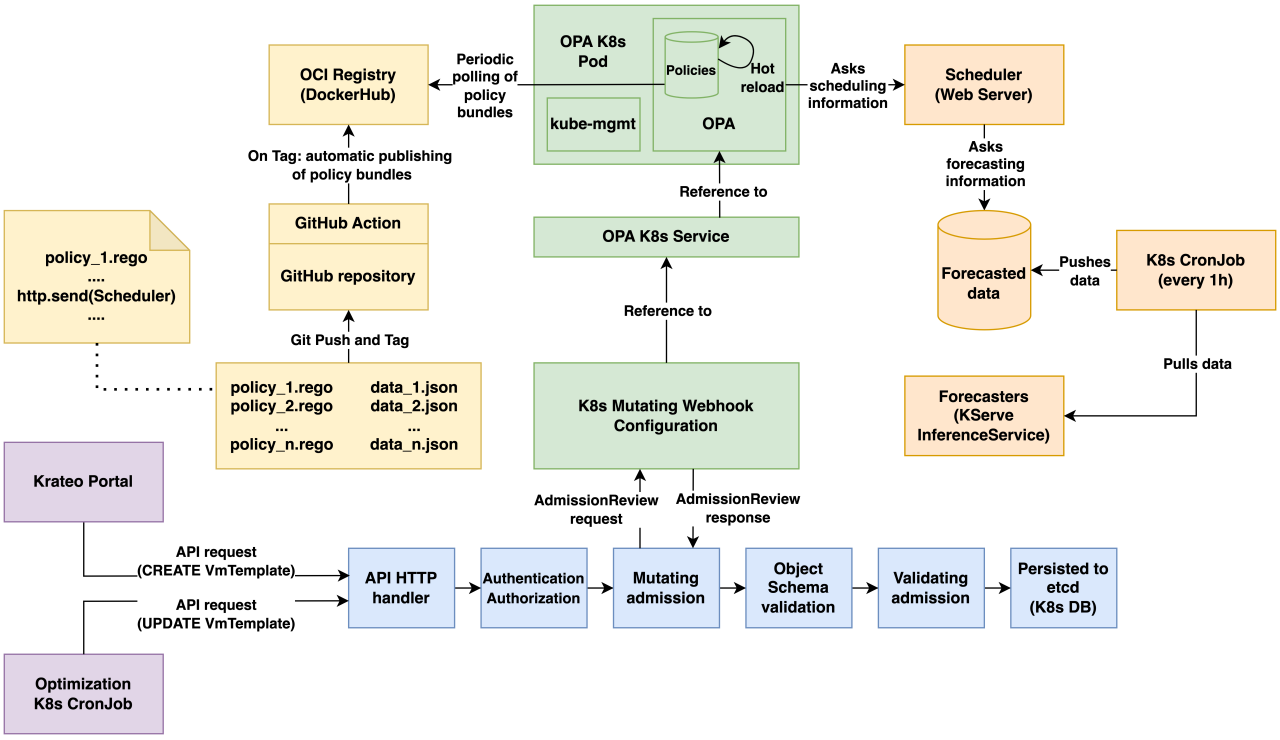

Figure 4.11: Kubernetes mutating webhook + OPA integration

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

4.6.13 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

4.7 MLOps infrastructure

A MLOps infrastrcutrue is not necessarily needed for multi-cloud resource management but since it is believed that ai or ml models will be used in the future more and more to get scheduling and management decisions, it is deemed important to describe a MLOps infrastructure in a Kubernetes environment.

4.7.1 MLOps purpose

MLOps implements DevOps principles, tools and practices into Machine Learning workflows

purpose: industrialize ML models lifecycle

faster model development

faster model selection and deployment to production

- model tracking (experiments, runs)
- model selection (model registry)
- model storage (in buckets)
- model deployment (inference)

allows all the team member to have visibility on the status of the ML models

Instead of having the so-called "AI Inference Mock Server", treated as a black box returning a scheul;ding time and schedulingLocation

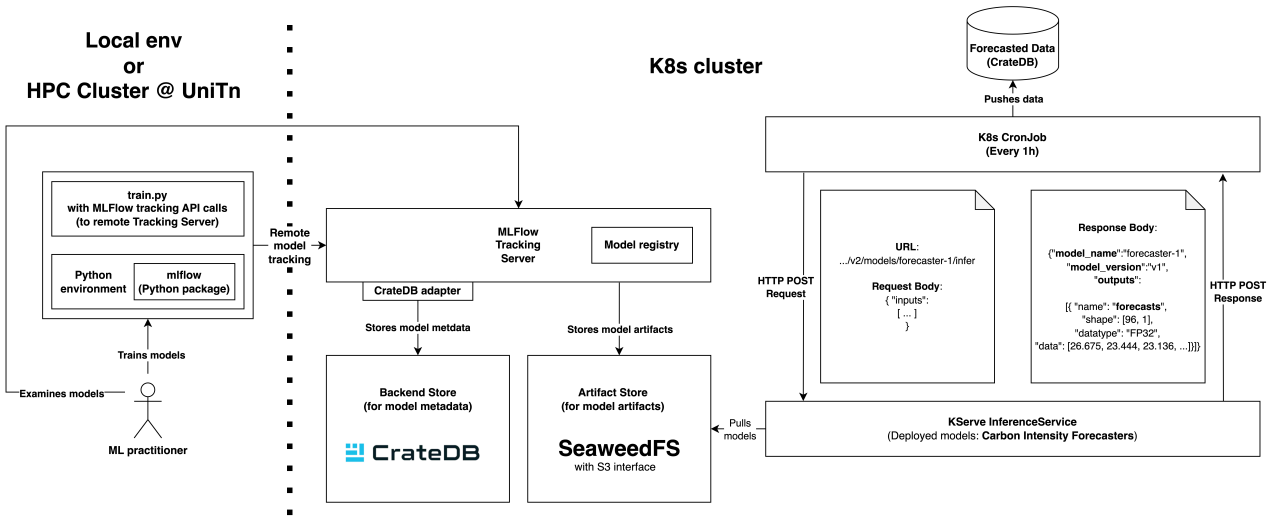

Figure 4.12: MLOps Architecture

MLFlow framework

KServe framework

4.7.2 MLflow

MLflow Tracking Server

mlflow is compatible with many ML frameworks like sklearn, pythorch

what is a model tracking server what is a model registry

Mlflow API calls autolog function infer signature important since store

the end result is a self contained fodler with everythiong needed it allows reproducibility

The training script will also serialise our trained model, leveraging the MLflow Model format.

model/ MLmodel model.pkl conda.yaml requirements.tx

additional challenge: CrateDB is not supported natively by mlflow framework a CrateDB adapter / wrapper is devoloped and mantianed by cratedb community CrateDB as metadata stoere

SeaweedFS as artifact store MINio could be an alternative altough it has a restrictive license [**?**].

MODEL signature

Alternative configuration 1

watchdog watchdog (pyhton package) PoC sidecar container
artifact store not needed

Alternative configuration 2

Another possible configuration could be the adoption of just CrateDB as both Metadata Store and Artifact Store.
This would be possible if CrateDB supports blob storage but not object storage
This solution cannot be implemented yet

4.7.3 KServe

KServe Inference Service
what is inference server / model server
used to deploy the forecaster (ML model)
uses Istio and Knative under the hood but a deep descrption of those is out of the scope of this theses. features: scaling to zero, etc
InferenceService with TorchServe runtime which is the default installed serving runtime for PyTorch models.
Kserve project proposes a standard protocol for inference servers. The version 2 of the KServe Inference Protocol is the Open Inference Protocol.

Open Inference Protocol

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

adopted by NVIDIA
multi model deployment
our strategy: 1 model per region 1 generic model? as fallback if specific model is not available?
Kserve "stack"
Kserve
in kserve 0.14.1 clusterservingruntimes supported are 10 among which torchserve
clusterservingruntimes -¿ kserve-mlserver (supported models: sklearn, xgboost, lightgbm, mlflow)
mlserver
serving runtimes
Seldon MLserver
accorgimenti:

```
import torch

class WrappedModel(torch.nn.Module):
    def __init__(self, original_model):
        super().__init__()
        self.original_model = original_model

    def forward(self, *args, **kwargs):
        return self.original_model(*args, **kwargs)['prediction_outputs']

# Wrap the existing model
```

```
model = WrappedModel(model)

# Now calling model() will return only 'prediction_outputs' (test)
print(model(test_dataset[0]['past_values'].unsqueeze(0)))
```

Listing 4.12: Wrapping a PyTorch Model

4.8 Measurements

4.8.1 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.

4.8.2 Impact framework

Impact framework (by green software foundation)

4.9 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

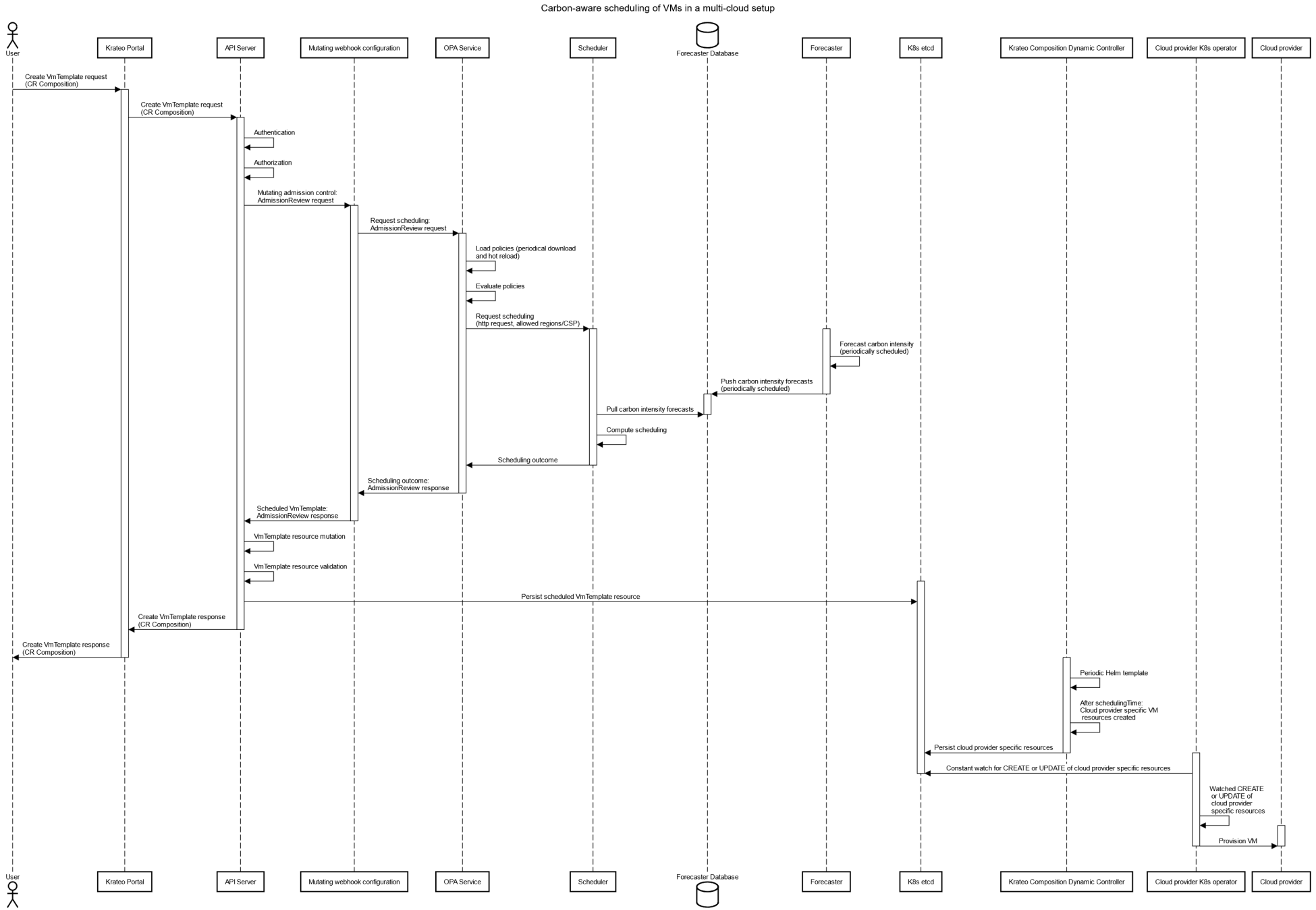

Figure 4.13: Example of a full-page rotated image

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)

5 Discussion

5.1 End-to-end integrated test

A comprehensive end-to-end integrated test has been carried out on a Kubernetes cluster

this was used to validate the system

(dependency graph)

5.2 Theoretic upper bound

(how close can we get, masachussets amherest group)

5.3 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.

5.4 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)

5.5 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. Ethics?

5.6 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.

6 Conclusion

production-ready system

6.1 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

there is one paper (https://ceur-ws.org/Vol-2382/ICT4S2019$_paper_2$8.pdf)$thatuseslocalairtemperatureands$
$\backslash Solarirradiancevariesmorewidelythancarbonintensityacrossglobalregions$".

Maybe it could be an extension of our system in the future.

6.1.1 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