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

Abstract **4**

1 Introduction **5**
1.1 Context . . . 5
1.1.1 GreenOps . . . 5
1.1.2 Geographical shifting and Time shifting . . . 5
1.1.3 Carbon-aware workload scheduling . . . 5
1.2 Problem statement . . . 5
1.3 Personal contribution . . . 5

2 Background **6**
2.1 GreenOps landscape . . . 6
2.2 Cloud providers . . . 6
2.2.1 Multi cloud . . . 6
2.2.2 computational sustainability by cloud providers . . . 6
2.3 Kubernetes . . . 6
2.3.1 Kubernetes as a platform . . . 6
2.3.2 Kubernetes extendability . . . 6
2.4 Krateo . . . 6
2.5 State of the Art . . . 6
2.5.1 CASPER . . . 7
2.5.2 CASPIAN . . . 7
2.5.3 Let'sWaitAwhile . . . 7
2.5.4 Other systems . . . 7
2.5.5 SOTA Recap . . . 7

3 Method **8**
3.1 Project division??? . . . 8

4 Design and Implementation **9**
4.1 Assumptions . . . 9
4.2 System Architecture . . . 9
4.3 Krateo PlatformOps integration . . . 9
4.3.1 Helm . . . 9
4.3.2 Krateo . . . 9
4.3.3 Krateo . . . 9
4.4 Cloud providers Kubernetes operators . . . 10
4.4.1 Azure Kubernetes Operator . . . 10
4.4.2 GCP Operator . . . 10
4.4.3 AWS Operator . . . 11
4.5 Open Policy Agent (OPA) . . . 13
4.5.1 Policy as Code paradigm . . . 13
4.5.2 OPA architecture overview . . . 13
4.5.3 OPA and external data . . . 14

4.5.4 OPA integration with Kubernetes . . . 14
4.5.5 OPA policies . . . 16
4.5.6 OPA Policy bundles . . . 17
4.5.7 OPA Gatekeeper . . . 18
4.5.8 Latency policy . . . 19
4.5.9 GDPR policy . . . 19
4.5.10 Mutation policy . . . 20
4.5.11 Data mapping . . . 20
4.5.12 OPA end-to-end workflow . . . 20
4.6 MLOps infrastructure . . . 21
4.6.1 MLOps purpose . . . 21
4.6.2 MLflow . . . 21
4.6.3 KServe . . . 22
4.7 Measurements . . . 23
4.8 End-to-End workflow . . . 23

5 Discussion **24**
5.1 End-to-end integrated test . . . 24
5.2 Theoretic upper bound . . . 24
5.3 Baseline definition . . . 24
5.4 Black hole phenomenon . . . 24
5.5 Preliminary evaluation . . . 24

6 Conclusion **25**
6.1 Future improvements . . . 25

Bibliography **26**

List of Figures

4.1 Minimum set of Azure resources for VM provisioning 10
4.2 Minimum set of GCP resources for VM provisioning 11
4.3 Minimum set of AWS resources for VM provisioning 11
4.4 OPA architecture 14
4.5 Kubernetes mutating webhook and OPA integration 15
4.6 MLOps Architecture 21

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 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

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

2 Background

2.1 GreenOps landscape

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

proposed a standard for data like the FOCUS standard available as per 2025 this standard is not yet adopted by cloud providers

developed the Impact Framework which will be described in section XY

Green Software foundation

trying to push a specification similar to what focus is for FinOps

2.2 Cloud providers

cloud regions

regions vs availability zones Cloud providers usually further divide region into ...

2.2.1 Multi cloud

(why, how to achieve)

advantages

reduces vendor lock-in

2.2.2 computational sustainability by cloud providers

2.3 Kubernetes

2.3.1 Kubernetes as a platform

Kubernetes as a platform to manage things

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

what is krateo. Recognized by Gartner by 2025 companies without a ... (cite)

architecture, components core provider, cdc helm charts as native resources

values.schema.json

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 [4]. 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 [4]. 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 [4]. 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
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

System design and implementation

4.1 Assumptions

In this work, workload has been modeled as Virtual Machines (VMs). This was the first use case taken into account, as it was deemed the ...

However the system was designed with flexibility in mind and can be extended to other type of workload, as described in section XYZ. a limitiation is that the resource that can be provisioned are the only ones that are supported by the cloud provider operator. As a matter of fact, not every cloud resource offered by a cloud provided is guaranteed to be supported by the relative k8s operator

4.2 System Architecture

In this section the System Architecture will be described

Main components of the system:

- Krateo PlatformOps
- Cloud providers Kubernetes operators
- Kubernetes mutating webhook
- Open Policy Agent section
  - OPA Server
  - OPA policies and data
- MLOps infrastructure
  - MLflow
  - KServe

Differences wrt other systems described in the background section production-ready system

4.3 Krateo PlatformOps integration

4.3.1 Helm

what is an helm chart

4.3.2 Krateo

what is krateo developer platform

4.3.3 Krateo

what is krateo developer platform

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

Why K8s synchronization operator from scratch was not ideal wrt using helm charts with krateo

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

4.4 Cloud providers Kubernetes operators

Integrating operators from different cloud providers allowed us to effectively create a multi-cloud system.

Operators: Continuous Reconciliation

They let manage external cloud resources inside a Kubernetes cluster In Kubernetes there is a representation of what is actually provisioned on a public cloud This representations are Custom Resources As illustrated in the following subsections

In particular, we decided to logically replace the custom K8s Operator for the mapping between generic resource to cloud-specific resource with Krateo Core Provider. In short, instead of embedding business logic in a K8s operator, we will leverage Helm templating to generate cloud-provider specific resources.

the minimum set of cloud resources needed for the provisioning of virtual machine on each of the cloud provider must be determined

4.4.1 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

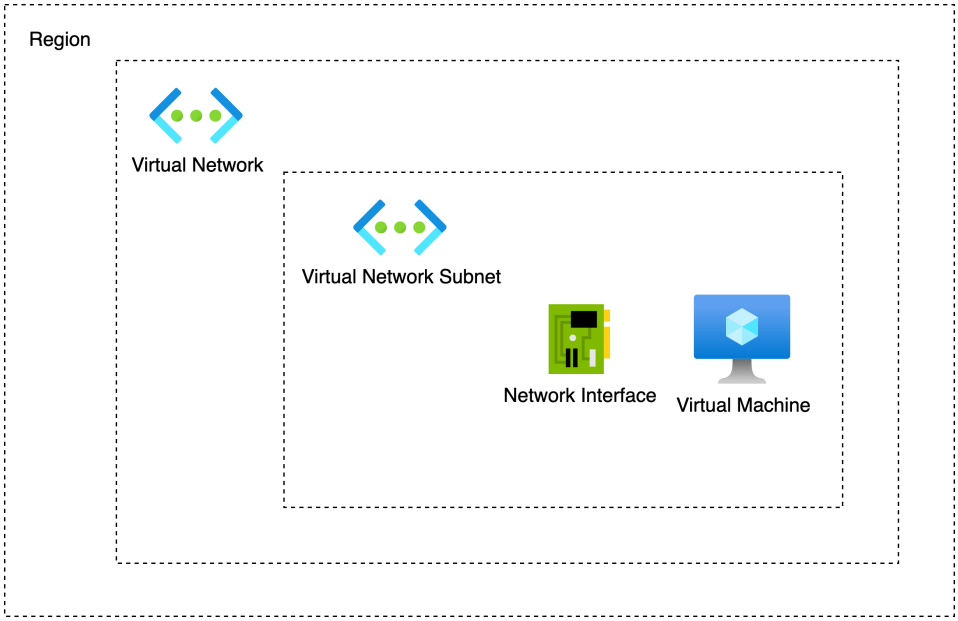

Figure 4.1: Minimum set of Azure resources for VM provisioning

INSTANCE CR example

4.4.2 GCP Operator

minimum set of resources needed for vm deployment

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

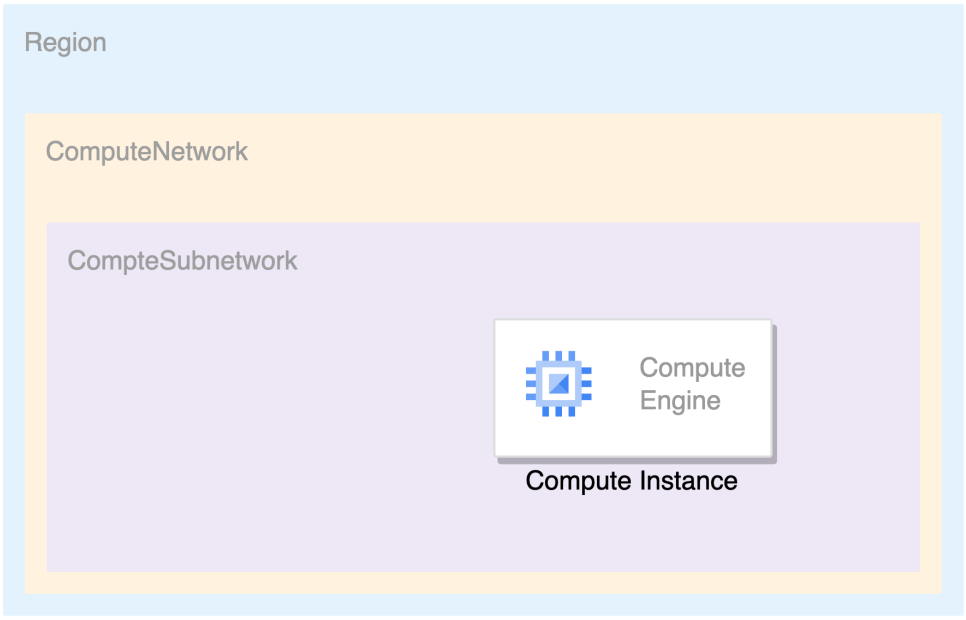

Figure 4.2: Minimum set of GCP resources for VM provisioning

4.4.3 AWS Operator

minimum set of resources needed for vm provisioning

- VPC
- Subnet
- EC2 Instance

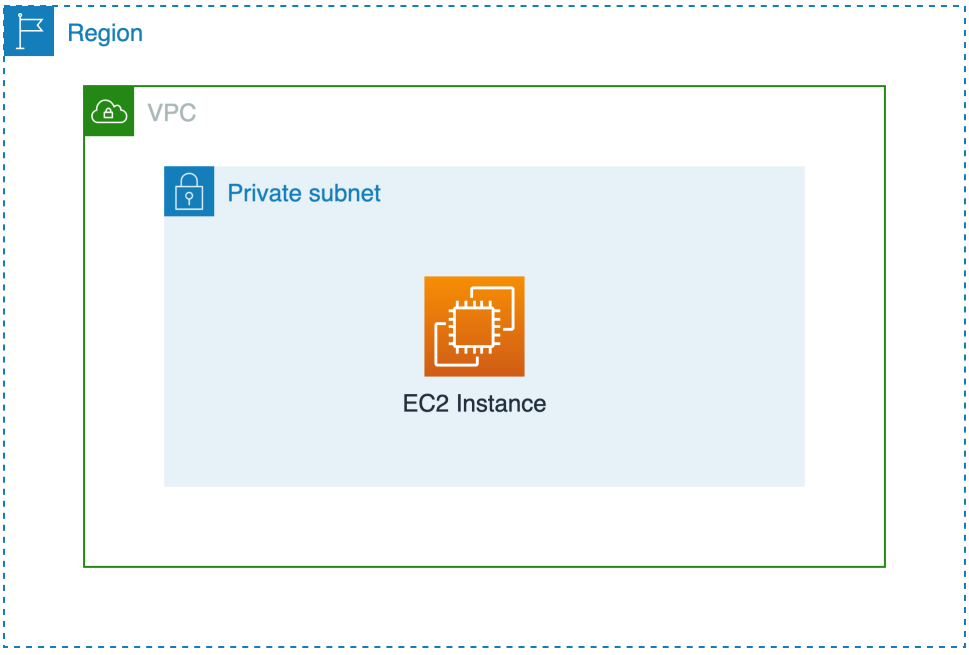

Figure 4.3: Minimum set of AWS resources for VM provisioning

INSTANCE CR example

Our implementation allows us to compatible with any design choices of cloud provider. One example of design choice is not allowing K8s object reference inside CR manifest for instance

interesting since it is an example of using helm lookup functions

while the simplest way, from the devolper standpoint would be to use references to K8s objects just like what you can do with subnet manifest to reference a vpc

HELM LOOKUP LISTING

—

Amazon AMI what is an AMI what are the parameter to determine an AMI

Testing: manual tests were made to check correspondance from scraped ubuntu website and AWS console.

4.5 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**" [1].

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 [1].

4.5.1 Policy as Code paradigm

According to AWS, Policy-as-Code (PaC) is a software automation approach which is similar to Infrastructure-as-Code (IaC) [3]. PaC helps assess company system configurations and validate compliance requirements through software automation [3]. 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 [3].

OPA's generic defintion of policy is: "*A policy is a set of rules that governs the behavior of a software service*" [2]. 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.

4.5.2 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 tighlty 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 4.5.5. These bundles enable version-controlled, centralized policy distribution, ensuring consistency and maintainability across distributed environments.

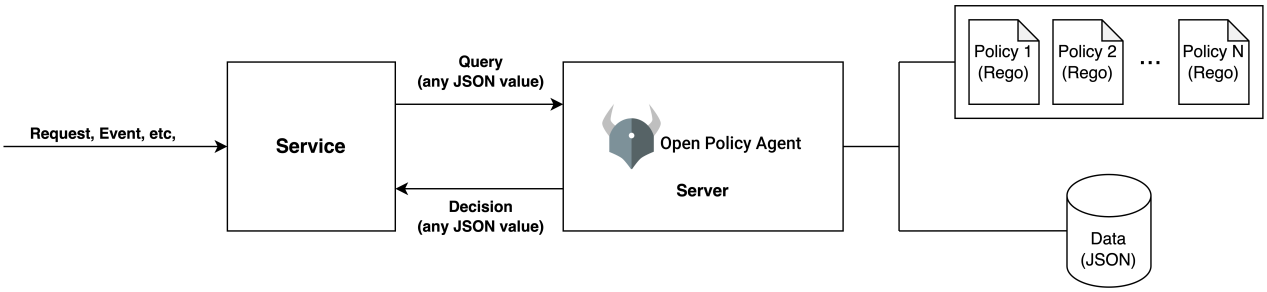

Figure 4.4: OPA architecture

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

4.5.3 OPA and external data

types of external data strategies

http.send() paramters

4.5.4 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 4.5 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

kube-mgmt 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 name conflicts, where OPA needs to check existing names in 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

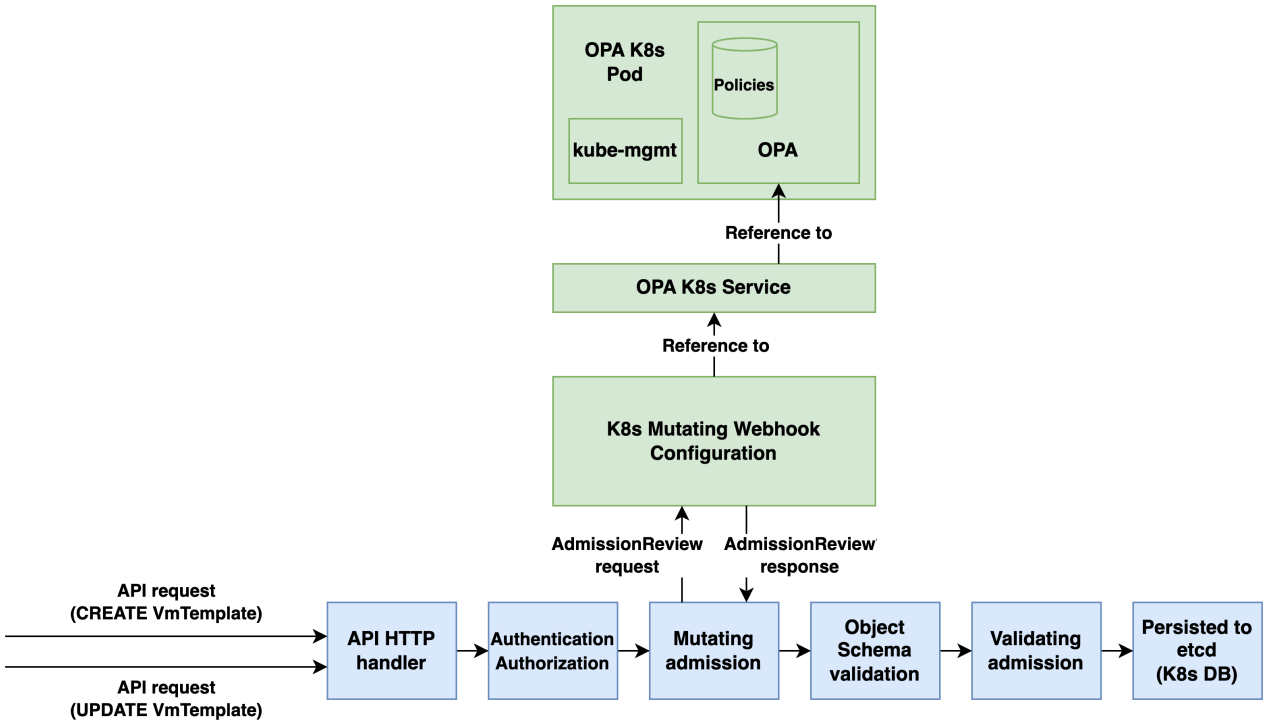

Figure 4.5: Kubernetes mutating webhook and OPA integration

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.

In the current system configuration, the kube-mgmt container is deployed to facilitate resource replication, ensuring that Kubernetes resources, including CustomResourceDefinitions (CRDs), are synchronized with the OPA instance. However, at present, no policies require 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.

4.5.5 OPA policies

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

The root policy 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

Listing 4.1: Rego policy for Pods registry

```
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 4.2: Rego "root" policy (system.main)

```
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}
```

```
{
    "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 4.3: AdmissionReview request

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

Listing 4.4: 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 handles **policy enforcement**, effectively rejecting the CREATE request since it violates the specified rules.

4.5.6 OPA Policy bundles

what is a policy bundle

how to package a bundle budles as OCI images

OPA server is configured to pull bundles from a specified registry repository 1 or more bundles

CI CD gitops

descrizione release.yml

impacchettamento policy

hot reload performed at application level no need for the opa K8s pod to be restarted

big advantage since if we want to add a new policy or update data we just push those changes in a code repository (like on GitHub) and the CI/CD (GitHub action) will bundle and publish the policies as a OCI Container on a Container registry.

4.5.7 OPA Gatekeeper

OPA Gatekeeper is ... could be seen as the go-to solution for kuberntes architecture. this is prbably true for simple use cases. not useful for the problem that must be tackled in this system (mutation and leveraging external data). differences wrt normal OPA deployment. OPA Gatekeeper advantages: no policy bundles but K8s custom resources (name of the CR to be added). for basic mutations is also fasibile and in this case rego code in not needed (there are specific resource called mutators with specific fields to tune in order to modify specific resource fields). OPA Gatekeeper limitations: on mutations and external data

To illustrate the differences between a standard OPA policy and an OPA Gatekeeper policy, we present two examples: (1) a simple Rego policy that enforces a basic constraint, and (2) 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 4.5: Simple OPA Rego Policy

The second example 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.

```
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 4.6: OPA Gatekeeper ConstraintTemplate

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

```
  parameters: {}
```

Listing 4.7: OPA Gatekeeper Constraint

4.5.8 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 of latency matrix (maybe 10x10?) with colors]

[code of the policy]

```
...
"italynorth": {
                "australiacentral": 286,
                "australiacentral2": 278,
                "australiaeast": 279,
                "australiasoutheast": 266,
                ...
                "francecentral": 24,
                "francesouth": 15,
                "germanynorth": 25,
                "germanywestcentral": 20,
                "israelcentral": 50,
                "italynorth": 0,
                ...
},
"japaneast": {
                "australiacentral": 108,
                "australiacentral2": 107,
                "australiaeast": 104,
                "australiasoutheast": 115,
                "brazilsouth": 278,
                "canadacentral": 159,
                "canadaeast": 169,
                "centralindia": 122,
                "centralus": 137,
                "eastasia": 52,
                "eastus": 170,
                "eastus2": 163,
                ...
},
...
```

Listing 4.8: Latancy matrix example

4.5.9 GDPR policy

Another policy configured is the GDPR

set of cloud regions that resides inside countries of the European Union.
speicifc regions for each cloud provider encoded in the data in json (along with latency matrix)

4.5.10 Mutation policy

main policy dedicated to
patch code
emcopded patches

4.5.11 Data mapping

OPA is powerful enough to ...
this mappings are needed since the scheudler knows only...
inside the policy is a good place to do this mapping
CHART image (illustrating data mapping steps)

```
# 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 4.9: Rego data mapping

4.5.12 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()
sort of GitOps since we deploy policies and build from a repo periodic polling of rego policies + hot reaload of policies at application level, no need for pod/container restart
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 **??** represents the configuration of the Kubernetes Mutating Webhook with the intergeation of Open Policy Agent. In particular,
Day 2 operations 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 MLOps infrastructure

mlops venn diagram image to explain what it is

4.6.1 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

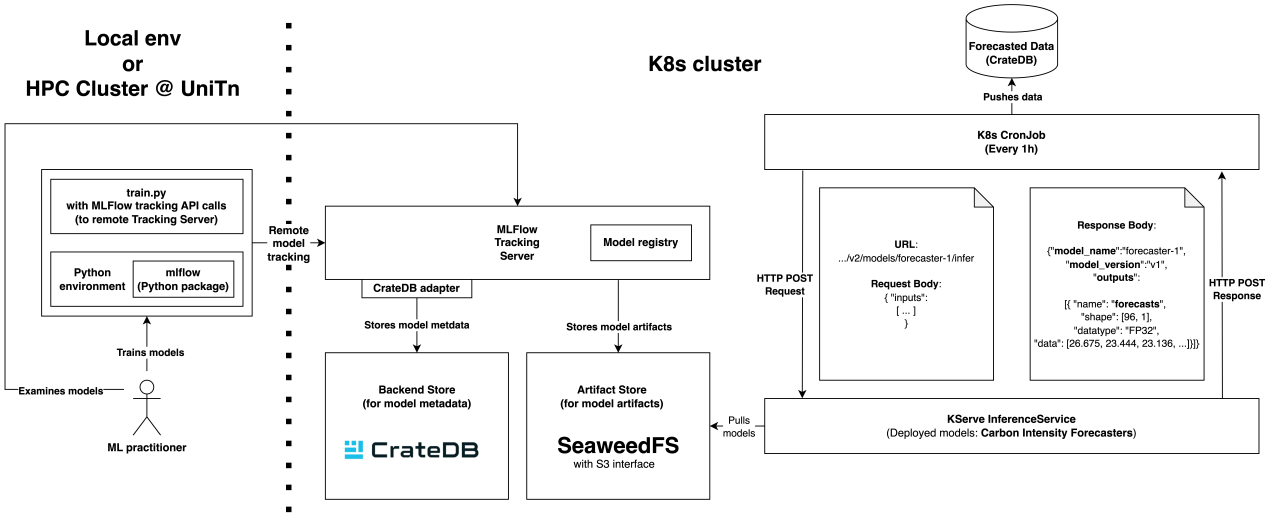

Figure 4.6: MLOps Architecture

MLFlow framework

KServe framework

4.6.2 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 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

4.6.3 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:

```python
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 4.10: Wrapping a PyTorch Model

4.7 Measurements

Impact framework (by green software foundation)

4.8 End-to-End workflow

swim lanes chart figure

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

(dependency graph)

5.2 Theoretic upper bound

(how close can we get, masachussets amherest group)

5.3 Baseline definition

5.4 Black hole phenomenon

(how it is countered)

5.5 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.

6 Conclusion

production-ready system

6.1 Future improvements

day2 operations we are ready for this

other resources we need templates operators

there is one paper (https://ceur-ws.org/Vol-2382/ICT4S2019$_{p}aper_{2}8.pdf)thatuseslocalairtemperatureands$
$\backslash Solarirradiancevariesmorewidelythancarbonintensityacrossglobalregions$".

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

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

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

[3] 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.

[4] 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.