Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

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

Supervisor

Prof. Sandro Luigi Fiore

Student

Leonardo Vicentini

Co-supervisors

Dott. Diego Braga

Dott. Francesco Lumpp

Academic year 2023/2024

Acknowledgements

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

Contents

List of Figures

Abstract

In the last 20 years, cloud computing has steadily become the backbone of modern digital infrastructure, enabling scalable and flexible deployment of heterogeneous services across geographically distributed data centers around the world. It can be safely said that cloud computing can be labelled as a disruptive technology since it has revolutionized the way IT services and applications are designed, developed, delivered and consumed. Moreover, cloud computing is playing a crucial role in the current AI revolution, enabling the training of large-scale machine learning models (such as Large Language Models) in a distributed manner and facilitating the deployment of AI-based services (such as chatbots, AI-powered search engines, etc.). The central role of cloud computing is also testified on the economic and financial side: indeed, according to recent market research for the year 2024 the total revenues of public cloud infrastructure services reached $330 billions [60]. However, the rapid adoption and growth of cloud computing has also raised concerns about its environmental impact. The reason is that the big computing infrastructures require a huge amount of electricity to operate. By consequence, this energy consumption is responsible for an important amount of carbon emissions. Recent efforts in computational sustainability have been made and among these the GreenOps operational paradigm arose. It must be also said that majors public cloud providers (such as Amazon Web Services, Microsoft Azure, Google Cloud Platform) are already taking steps to reduce their carbon footprint in a variety of ways. These spans from internal optimization of data centers, to bringing awareness to the customers on their individual carbon footprint.

The work described in this thesis is part of a larger project which has the primary goal of developing a system for **reducing the carbon footprint of workloads in the cloud**. Said system is mainly based on the idea of exploiting **time-shifting** and **geographical shifting** of cloud workloads to time periods and geographical regions with low carbon intensity. As a matter of fact, nowadays, the carbon intensity of electricity varies significantly depending on the time of day and the geographical region. Therefore, it is deemed interesting to leverage these fluctuations in electricity grid carbon intensity to schedule workloads in a way that reduces the carbon footprint of the workloads compared to a traditional scheduling approach. Currently, targeted workloads are the ones that are not time-sensitive but instead are quite delay-tolerant. In addition, workloads are also characterized by other requirements, such as Quality of Service (QoS) requirements (e.g., deadlines, maximum latency constraints, etc.) and cost constraints (e.g., budget constraints, cost optimizations, etc.). Indeed a **policy-driven approach** is used to encode these requirements along with scheduling outcomes. The system is designed to be used in a multi-cloud setting, where workloads can be scheduled on different cloud providers. Adopting a multi-cloud paradigm offers several advantages, including user-centric flexibility, avoidance of vendor lock-in, and the ability to exploit multiple regions because cloud providers may have varying geographical coverage. Some of the guiding principles adopted in the design of the system are: flexibility, extensibility, cloud-agnosticism. In order to implement these principles, the use of **Kubernetes as a platform for multi-cloud resource management** is considered.

The project is carried out in collaboration with Krateo SRL, a company that is developing a modular platform for resource management and with Electricity Maps, a company that provides historic and real-time data on the carbon intensity of electricity grids around the world. The former provided support for platform integration and supplied the infrastructure used for testing while the latter was the data provider.

This thesis addresses the following research question: "How can flexible and dynamic resource management be achieved to satisfy a varied set of requirements in the context of multi-cloud environments?"

To tackle this wide and complex problem, this research envisions, designs, and implements a policy-driven architecture based on the Kubernetes ecosystem. In particular, we propose and describe a multi-cloud resource management system that is mainly built upon Kubernetes, Krateo PlatformOps and Open Policy Agent (OPA).

Some of the objectives that the system aims to achieve are the following: multi-cloud compatibility, policy-driven resource management, extensibility and flexibility. Multi-cloud compatibility is achieved by providing a vendor-agnostic solution that can operate across multiple cloud providers. Enabling the flexible definition of policies that encode requirements and constraints of workloads (e.g., Service Level Agreements, Quality of Service, economic cost reduction, carbon footprint reduction) is how policy-driven resource management is accomplished. Extensibility and flexibility are obtained by designing a modular system that can be adapted for any kind of cloud resource and arbitrary requirements.

In particular, the use case taken into consideration in the context of this thesis is the scheduling of virtual machines (VMs) in a multi-cloud environment to minimize their carbon footprint while maintaining an exemplificative set of requirements and constraints. As a matter of fact, the traditional management of resources and workloads in the cloud does not usually consider the environmental impact, leading to suboptimal energy usage and excessive carbon emissions. However, users and organizations, even if interested in reducing carbon emissions and energy consumption, may not have the necessary systems to do so. In addition, they may be reluctant to give up performance and cost requirements in favor of environmental sustainability. Therefore, the proposed system enables VM scheduling in a way that minimize carbon footprint while satisfying other requirements and constraints.

The research methodology follows a **hands-on approach** to developing a practical, production-ready solution, integrating and leveraging existing open-source technologies and tools. The phases of the methodology followed for this thesis are the following:

1. **Literature and Technology Review**: The first phase of the research methodology is to review the existing literature and technologies related to the problem statement. This includes exploring the state-of-the-art of multi-cloud resource management and carbon-aware resource management. A study on public cloud providers and the multi-cloud paradigm is also conducted. In addition, the review includes identification and study of the existing open-source tools and technologies that can be used to develop the solution (e.g., Kubernetes, Krateo PlatformOps, Open Policy Agent).

2. **System Architecture Design**: The second phase of the methodology is to design the system architecture, including the components, interactions, and data flows. This includes also the development of various Proof-of-Concepts in order to test the feasibility of the various identified candidate solutions and to understand tools' strengths, limitations, trade-offs.

3. **System Implementation**: The implementation phase involves the integration of the system components, the development of policies and policy bundles pipelines, the configuration of cloud resource templates, the set up of the MLOps infrastructure.

4. **System Deployment and End-to-End Testing**: The final phase of the methodology is to deploy the system on a real infrastructure and perform end-to-end testing to validate the system's functionality in a real-world multi-cloud scenario.

This thesis represents a technical and research contribution within a larger project supervised by Prof. Sandro Luigi Fiore. The project is divided into three main parts: data analysis, machine learning, and cloud infrastructure. The contribution of this thesis is on the cloud infrastructure part and includes:

- The design of the system architecture, including the identification of components, interactions, and data flows.

- The study and development of a policy-driven approach to workload scheduling that also enables the encoding of requirements and constraints.
- The implementation of the system, including the integration of the system components, the development of policies and policy bundles pipelines, the configuration of cloud resource templates, the set up of the MLOps infrastructure.
- The deployment of the system on a real infrastructure and the performance of end-to-end testing to validate the system's functionality in a production-like multi-cloud scenario.

For what concerns the integration with the other parts of the project, the system developed in this thesis is designed to be integrated with the machine learning part of the project which can be summarized as a series of machine learning models that predict the carbon intensity of electricity grids and a scheduler that uses these predictions to schedule workloads. In order to carry out the end-to-end testing, this scientific work is deployed on the infrastructure leveraging tools described in section 2.7 and referred as MLOps infrastructure.

1 Background

In this chapter, we provide an overview of the main concepts, paradigms and technologies that are relevant for the purpose of this work. We start by introducing the concept of **Public Cloud Providers** and the **Multi-Cloud paradigm**. We then provide an overview of the **GreenOps landscape**, with a focus on the the **Computational Sustainability** initiatives by Public Cloud Providers. A brief overview of **Kubernetes**, in particular focusing on the concept of **Kubernetes as a platform** and the **Helm** package manager is deemed necessary. After that, we describe **Krateo PlatformOps**, an open-source Kubernetes-based platform that is a fundamental part of our system. Then we introduce **Open Policy Agent** and the **Policy as Code** paradigm. A pair of MLOps tools are then introduced: **MLflow** and **KServe**. An analysis of **cloud resource metrics** and their uses is then provided. We deemed particularly valuable to provide an overview of the existing works in the field of **multi-cloud resource management** to highlight recurrent patterns and design choices. Finally, we present some works on **carbon-aware systems for resource management**.

1.1 Public cloud providers

The Cloud Computing definition by the National Institute of Standards and Technology (NIST) [41] states that "*Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [41]. **Public Cloud Providers** or Cloud Service Providers (CSPs) are companies that have as their core business the provisioning of cloud computing services. These services, which are growing in number and complexity over time, range from computing resources to storage, networking, databases, machine learning solutions, and more. Public cloud is a deployment model that allows organizations to consume cloud services without having to build and maintain their own physical infrastructure (e.g., private cloud), therefore reducing capital expenditure (CapEx) and shifting to an operational expense model (OpEx). The public cloud model is actually a multi-tenant environment where physical resources, maintained and operated by the cloud provider, are shared among multiple customers (tenants) with the means of virtualization and isolation techniques. These resources are offered in the form of various services at different levels of abstraction (e.g., Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service(SaaS)) and are provided on-demand on a pay-as-you-go basis. Currently, as of 2024, the public cloud market is dominated by three main hyperscalers: **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud Platform (GCP)**. An hyperscaler is generally a company (Cloud Service Provider) that operates a data center infrastructure at a massive scale and is able to provide cloud services to a global audience. Figure 1.1 shows the worldwide market share of leading cloud infrastructure service providers as of Q3 2024 [60].

1.1.1 Cloud Regions and Availability Zones

With the term **cloud region**, cloud providers refer to a geographical area where they have one or more data centers. Cloud providers usually further divide regions into availability zones (AZs) which main purpose is to provide high availability and fault tolerance. For the purpose of this work, we will consider the concept of **cloud region** as the primary unit of deployment of cloud resources. Usually, by cloud providers design choice, each cloud region supports only a subset of the available cloud services and instance types (virtual machines). However, we could safely assume that our targeted workload specifications are quite standard and therefore can be scheduled on any cloud region. As of late 2024, the three major cloud service providers have established extensive global infrastructures. Table 1.1 shows the number of regions and availability zones (AZs) for each provider as of Q4 2024 - Q1 2025 [37] [3]. We must note that the number of regions and availability zones is constantly

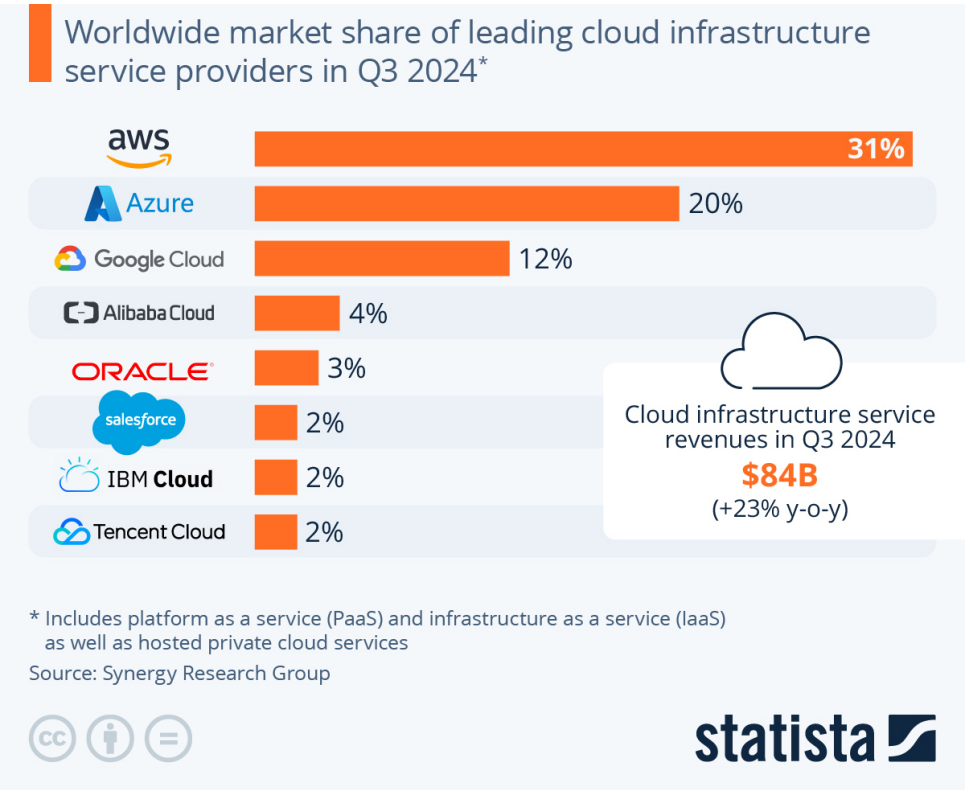

Figure 1.1: Worldwide market share of leading cloud infrastructure service providers as of Q3 2024 [60]

Cloud Provider	Regions	Availability Zones
Amazon Web Services (AWS)	36	114
Microsoft Azure	33	93
Google Cloud Platform (GCP)	40	121

Table 1.1: Number of Cloud Regions and Availability Zones by Provider as of 2024 [37]

changing as cloud providers are expanding their global infrastructure.

We must also note that each public cloud provider has a different number of regions and zones and also different naming conventions for them. There is no standardization nor a convention in the industry for this and cloud providers have chosen their own naming schemes. For instance, a region located in London is called "*eu-west-2*" in AWS, "*uk-west*" in Azure and "*europe-west2*" in GCP. This detail must be taken into account when designing and implementing a multi-cloud system. For what concerns the geo-distribution of cloud regions, as an example, Figure 1.2 shows the location of **Azure cloud data centers** around the world and the country Grid Carbon Intensity as reported by Electricity Maps (year 2024) [13]. The Grid Carbon Intensity is a measure of the carbon intensity of the electricity grid of a country and is expressed in grams of CO2 equivalent per kilowatt-hour (gCO2eq/kWh) and it is included in the figure to provide an insight on the carbon footprint of the cloud regions as it will be a key factor in the scheduling policy of our system. Data center coordinates are retrieved by an Azure CLI command dump [5] and plotted on a map using the Natural Earth dataset [40].

1.1.2 Multi-cloud paradigm

The multi-cloud paradigm refers to the strategic utilization of cloud services from multiple public cloud providers within a single, heterogeneous architecture. This approach allows organizations to distribute workloads, applications, and data across multiple public cloud providers [16]. On the other hand, the hybrid cloud paradigm refers to utilization of both public cloud services and on-premises infrastructure (e.g., a private cloud). The multi-cloud strategy is often adopted by organizations to

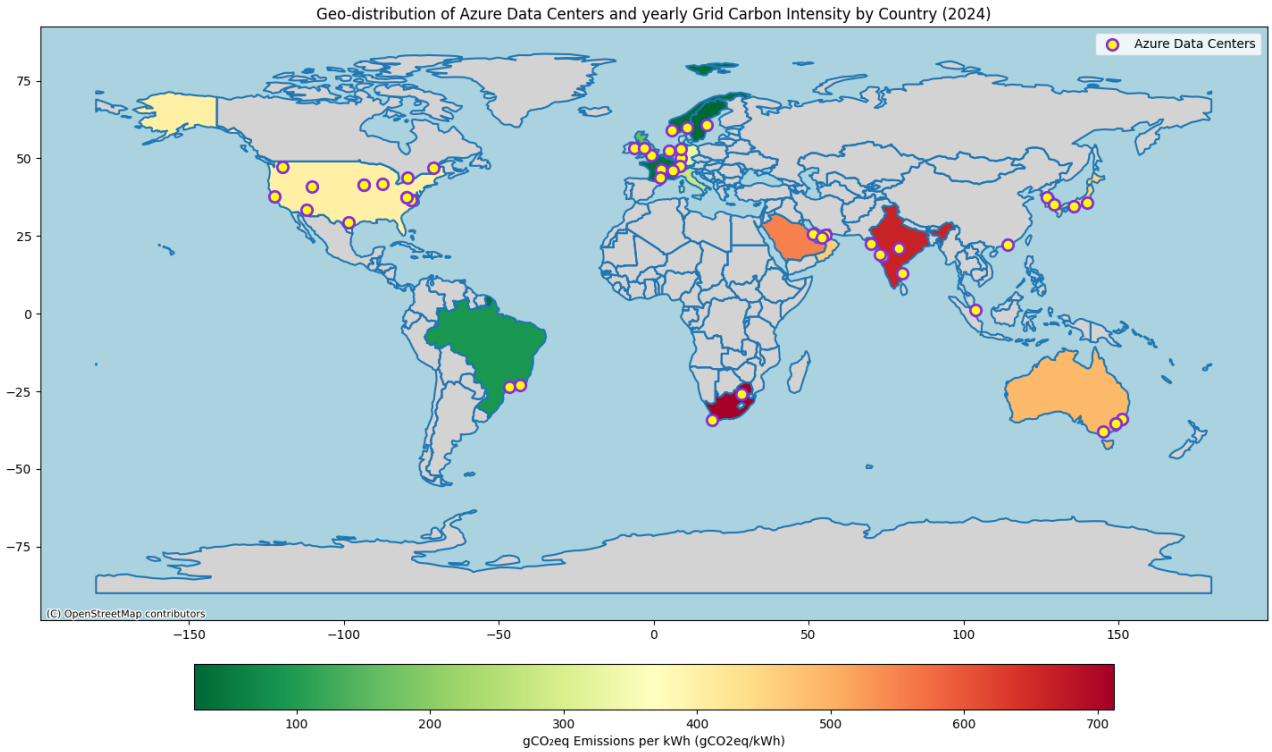

Figure 1.2: Geo-distribution of Azure data centers with country Grid Carbon Intensity

avoid vendor lock-in, **increase redundancy**, **increase flexibility** and **optimize costs** [16]. For what concerns flexibility and optimization, a multi-cloud strategy allows an organization to select the best service on a case-by-case basis, leveraging the strengths of each provider. Moreover, single point of failure weaknesses are mitigated by distributing workloads or critical services across multiple providers. It could even be the case that, for the end user, the choice of the cloud provider for a service is transparent meaning that the organization itself (or the system) is the one that decides which cloud provider to use for a specific service based on some criteria.

To avoid vendor lock-in, applications must be designed to be **cloud-agnostic** and leverage open-source technologies (e.g., Kubernetes) and standards [16]. If an application is designed to be cloud-agnostic, it can be relatively easily migrated from one cloud provider to another. Multi-cloud is a strategy that might be more difficult to implement compared to a single cloud strategy but the long-term benefits are significant.

Challenges and **disadvantages** are also present in a multi-cloud strategy [16]. As a matter of fact, it can be safely said that **complexity in the overall infrastructure management is increased**. Another challenge is the **integration of services** across different cloud providers. An additional consideration could be done in terms of security: a multi-cloud strategy can **increase the attack surface** of an organization, therefore increasing the requirement for security measures.
For the purpose of this work, adopting a multi-cloud strategy is beneficial for several reasons. First and foremost, it achieves **user-centric flexibility** bringing the benefits described above. If a user or organization has a preference for a specific cloud provider, the system can be configured to use only that provider or a specific subset of providers. Secondly, the system can be designed to be **cloud-agnostic**, meaning that the choice of the cloud provider is transparent to the end user. With some specific configurations, the subset of providers to be used can be decided by the system itself based on some criteria. Currently, as described in section 2.6.8, the user can specify a subset of cloud providers (among the 3 major ones) to be used for the deployment of resources. Finally, since different cloud

providers have data centers in various locations around the world, some low-carbon regions might be available only on a specific cloud provider and this could be leveraged for the deployment of resources.

1.2 GreenOps landscape

GreenOps is the abbreviation for "**Green Operations**" and is the term used to describe an operational model that aims to integrate sustainable practices into an organization's digital operations, with a particular focus on cloud computing and data centers. The interest in GreenOps has been growing in the last years due to the increasing awareness of the environmental impact of data centers and cloud infrastructures. The rising interest in the GreenOps ecosystem is also influenced by political and regulatory factors as the European Union's ambitious goals for the reduction of greenhouse gas emissions. In particular, the *Climate Neutral Data Centre Pact in Europe*, is an EU initiative which aims for making climate-neutral data centers by 2030 [9]. Being the work of this thesis focused on the cloud ecosystem, we deem useful to cite the Technical Advisory Group (TAG) on Environmental Sustainability which is focused on the cloud-native sustainability landscape. The TAG is part of the Cloud Native Computing Foundation (CNCF) and aims to provide guidance, standards, and best practices for the industry [58].

1.2.1 Green Software Foundation

Another pivotal actor in the GreenOps landscape is the Green Software Foundation. The Green Software Foundation is a non-profit organization, part of the Linux Foundation, that aims to promote the development of green software and to provide a set of standards, tooling and best practices for the industry [17]. It is considered useful to provide a quick summary of the foundation's major projects that are relevant to the context of this work.

Software Carbon Intensity Specification

Software Carbon Intensity Specification (SCI) is a specification that aims to provide a standard way to measure the carbon intensity of software systems. SCI is defined as a rate: the amount of carbon emissions per one unit of R where R is a functional unit for the software system (e.g., API call, new user, DB query, etc.) [57].

The SCI can be defined as follows:

$$SCI = C \times R = (O + M) \times R = ((E \times I) + M) \times R$$

where:

- SCI is the Software Carbon Intensity
- C is the carbon emissions
- R is the functional unit
- O is the operational emissions
- M is the embodied emissions
- E is the energy consumption
- I is the carbon intensity

Real Time Cloud

The Real Time Cloud project aims to provide a standard for real-time carbon emissions data reporting for cloud providers. The goal is to provide real-time access to information about cloud regions, power usage effectiveness (PUE), water usage effectiveness (WUE), carbon-free energy from the grid and from Cloud provider individual initiatives. The concept is similar to FOCUS specification for FinOps [14] but is at a very early stage and is not yet adopted by cloud providers as of 2025 [54].

Impact Framework

The Impact Framework is a flexible framework for measuring and reporting the environmental impact of software systems [18]. The core of the framework is represented by a **Manifest file** which is a YAML file that is used both for describing **calculation pipelines** and for storing the results of the calculations. Pre-built plugins are available for the most common calculations (e.g., carbon intensity, energy consumption, etc.) but custom plugins can be developed as well [18]. A potential integration of this framework with our system is described in section 2.8.

1.2.2 Computational Sustainability by Public Cloud Providers

For the purpose of ths work, we assume that a cloud data center will likely rely on the same energy sources that characterize a specific geographical region. Indeed, cloud data centers typically draw power from their local electrical grids, meaning their carbon emissions are closely tied to the energy mix of their specific geographical regions just like any other industrial facility. For instance, if Finland's energy production is characterized by low carbon emissions, data centers located there are likely powered by clean energy sources. However, some public cloud providers may implement individual initiatives to enhance their access to renewable energy or enhance their energy efficiency and by consequence to reduce their carbon footprints. For instance, according to an AWS internal report, thanks to their works towards sustainability, they have been able to obtain an infrastructure that is up to 4.1 times more energy efficient than an equivalent on-premise infrastructure [4]. As another example of the efforts made by public cloud providers in the field of carbon-aware computing, we can describe how Google has been working on carbon-aware computing for internal use. In particular, they exploit workload flexibility to shift when, where and how computing is done to reduce carbon emissions [8]. Some of the types of workloads they identified as flexible are: **video processing**, **training large-scale machine learning models**, **simulation pipelines**. The main components they recognized to be necessary for carbon-aware computing are: accurate carbon intensity data, scalable infrastructures and migrations mechanisms [8]. Another important point is to adopt a **data-driven methodology**. Indeed, at Google the total amount of work (computing) that needs to get done per day is quite predictable thanks to the large amount of data they can analyze to make predictions [8].

In section 1.8.3 we will briefly describe how public cloud providers are already providing carbon footprint data reports to their customers.

1.3 Kubernetes

Kubernetes (K8s) is an open-source platform for automating the orchestration of containerized applications. It is widely used in the industry and became the **de-facto standard for container orchestration**. Figure 1.3 shows the Kubernetes architecture as described and depicted by the Cloud Native Computing Foundation (CNCF) [21].

The main components of Kubernetes are:

- **API server**: the central component that manages the Kubernetes cluster. It exposes the Kubernetes API and is responsible for validating and mutating data related to Kubernetes objects before persisting them in the cluster state.
- **etcd**: a key-value store used to store the cluster state.
- **Controller manager**: a daemon that embeds the core control loops shipped with Kubernetes.
- **Scheduler**: a component that assigns Pods to nodes.
- **Kubelet**: an agent that runs on each node in the cluster. It makes sure that containers are correctly running in Pods.
- **Proxy**: a network proxy that runs on each node in the cluster. It maintains network rules and it is responsible for routing network traffic.
- **Container runtime**: the software that is responsible for running containers on the Node (in this case represented by Docker but other runtimes like containerd are supported).

The first four components of the above list are the main control plane components of Kubernetes while the last three are the main components of each worker node in the cluster. It must be noted that

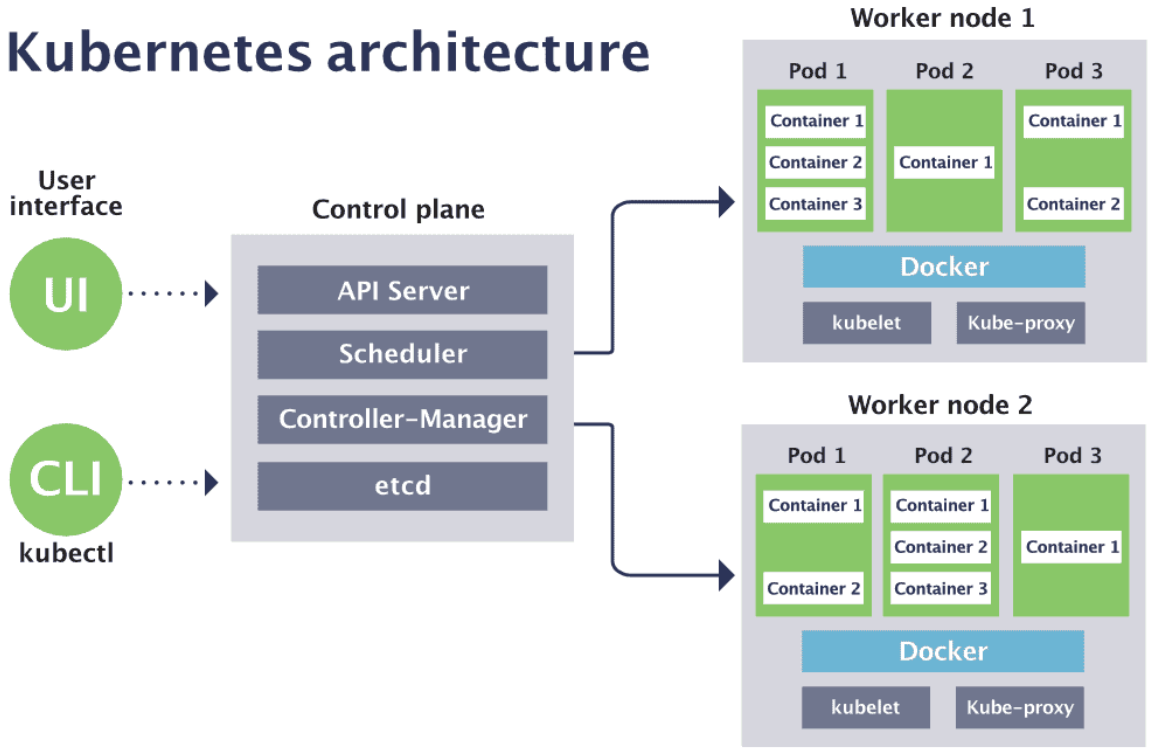

Figure 1.3: Kubernetes architecture by CNCF [21]

in our system we are not extending the Kubernetes scheduler as described in sections 1.10.4 and 1.10.5, since we are not dealing with the scheduling of in-cluster resources (e.g., Kubernetes Pods) nor with the provisioning of entire Kubernetes clusters. We are instead focusing on the **management of external resources on cloud providers**, leveraging Kubernetes as a control plane for the management of these resources. As described in the following section, our focus will be on the Kubernetes API server and in particular on Kubernetes admission control.

1.3.1 Kubernetes extensibility

Kubernetes allows for the extension of its functionalities through the use of **Custom Resource Definitions (CRDs)** and **Kubernetes Operators**, effectively adopting the so-called **operator paradigm**. Simply put, CRDs are a way to instruct Kubernetes to manage new resource types and can be added and removed from a Kubernetes cluster at runtime, through dynamic registration. They are a **schema** that defines the structure of the resource and the Kubernetes API server will validate the resource against the schema. **Custom Resources (CRs)** are actually instances of the resources defined by CRDs and are managed by an operator. An operator is a special type of controller tasked with managing the lifecycle of the CRs. Indeed they implement **control loops** to compare the **current state** of the resources they manage with the **desired state** and take actions to make the current state converge to the desired state. A high-level overview of the operator paradigm is depicted in Figure 1.4. Effectively, the code of an operator is usually deployed on the Kubernetes cluster in the form of a Kubernetes deployment. This concept is similar to the standard built-in Kubernetes resources which are however manged by built-in controllers (e.g., Deployment controller, ReplicaSet controller, part of Kubernetes controller manager).

1.3.2 Kubernetes as a platform

Recently, the paradigm of leveraging Kubernetes as a platform to manage external resources has become more and more popular. Therefore, Kubernetes use cases have expanded beyond the manage-

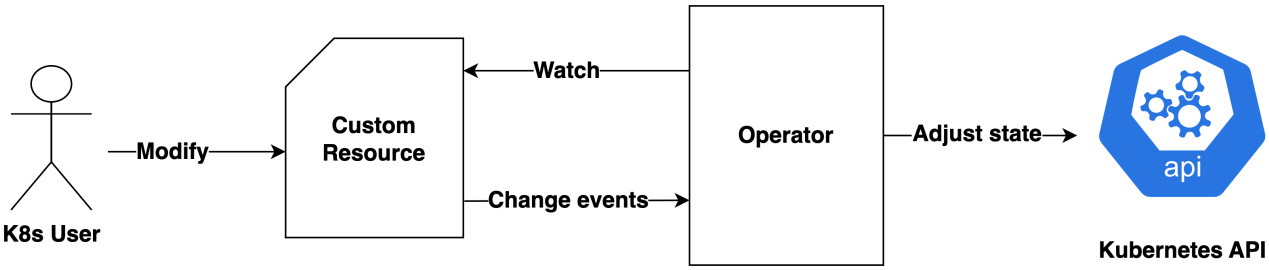

Figure 1.4: Operator paradigm

ment of containerized applications to the management of any desired resource on any infrastructure. The idea is to **represent external resources as Kubernetes objects** and leverage Kubernetes' declarative paradigm and robust API to achieve unified and efficient infrastructure management. This approach can be exemplified by the public cloud providers' operators which extend Kubernetes to manage their cloud resources as described in section 2.4.

For instance, Config Connector is a Kubernetes operator developed by Google Cloud that allows to configure numerous Google Cloud services and resources using Kubernetes tooling and APIs [15]. The general idea is that organizations might have to deal with a **heterogeneous infrastructure** composed of different cloud providers and on-premises resources. By **unifying infrastructure management under Kubernetes**, complexity is reduced and velocity is increased because the same tools and APIs can be used to manage all the resources in a consistent way [15]. In particular, Config Connector provides a set of CRDs that represent Google Cloud resources and services (e.g., Google Compute Instances, Google Kubernetes Engine clusters, BigQuery datasets, etc.) and the Config Connector operator is responsible for managing the lifecycle of these resources.

1.3.3 Helm

Helm is the de-facto standard **package manager** for Kubernetes and it allows to define, install and manage applications in a simpler way compared to standard resource manifests [19]. The key concept is the **Helm chart**, which is a collection of files that describe a related set of Kubernetes resources. These files are mainly of two types: templates and values. The **templates** are Kubernetes manifest files that are rendered by Helm's **powerful templating engine**. The **values** are the set of variables that are used to render the templates. Upon a chart installation, Helm will render the templates "injecting" the values and deploying the resources in the Kubernetes cluster. One major advantage that Helm provides is the complete management of the lifecycle of the resources. As a matter of fact, Helm allows to easily **upgrade**, **rollback** and **uninstall** the Kubernetes resources deployed with a Helm chart reducing time and human errors in such operations [19]. Without Helm, the user would have to deal with each single Kubernetes resource manifest file and manually apply changes to them. Finally, users can benefit of Helm charts already developed by the community and leverage chart distribution within their organization using Helm repositories (either public or private).

1.4 Krateo PlatformOps

Krateo PlatformOps (Krateo) is an **open-source Kubernetes-based platform** that aims to provide a unified interface for managing any desired resource on any infrastructure [28]. Krateo runs as a Kubernetes deployment inside a Kubernetes cluster but **acts as a control plane** even for resource external to the Kubernetes cluster. The only requirement for this management is that the resources need to be logically "encoded" using a YAML file which represents the desired state of the resources [28].

Krateo has three main components:

- Krateo Composable Operations: a set of core modules needed for resource managment
- Krateo Composable Portal: a web-based declarative user interface
- Krateo Composable FinOps: a set of modules for advanced cost optimization

For the purpose of this work, we will focus on the **Krateo Composable Operations** part, which is the core of the Krateo platform and is responsible for managing the lifecycle of resources in a Kubernetes cluster [28]. Krateo Composable Operations is composed in turn by several components. Due to their core importance in our system, we will briefly describe the functionalities of the **Krateo core-provider** and the **Krateo composition-dynamic-controller** as described in Krateo's official documentation [26], [25].

1.4.1 Krateo core-provider

The Krateo core-provider, as its name suggests, is the core component of the Krateo platform. It is a **Kubernetes operator** that has the duty of downloading and managing Helm charts.

It first checks for the existence of a file named *values.schema.json* in the chart folder and uses it to generate a Kubernetes CRD, accurately representing the possible values that can be expressed for the installation of the chart [26]. The file *values.schema.json* is a JSON schema that describes the structure of the *values.yaml* file for the related Helm chart and it is considered a standard best practice for Helm charts. It basically provides a way to validate the *values.yaml* file before the Helm chart is installed (i.e., to check if the values are in the correct format and if all the required values are present) [26]. Therefore, generally speaking, the file *values.schema.json* could be useful in the context of DevOps practices and CI/CD pipelines but in the case of Krateo it is pivotal for the generation of the CRD. In other words, the Krateo core-provider operator is responsible for deploying the Helm chart as a **native Kubernetes resource**, which allows for the management of the whole Helm chart lifecycle through Kubernetes APIs [26]. As a matter of fact, out of the box, Kubernetes does not provide a way to manage Helm charts natively and the Krateo core-provider is one tool that allows to do so.

Krateo core-provider introduces a Kubernetes CRD introduced called ***CompositionDefinition*** to represents the Helm chart and its values (a Helm Chart archive with a JSON Schema for the *values.yaml* file) [26]. Upon a CompositionDefinition manifest application to the Kubernetes cluster, the Krateo core-provider generates the CRD from the schema defined in the *values.schema.json* file included in the chart. It then deploys an instance of the Krateo composition-dynamic-controller, configuring it up to manage resources of the type defined by the CRD [26]. This is another example of the operator paradigm described in section 1.3.1 where the Krateo core-provider is the operator and the CompositionDefinition is the CRD.

1.4.2 Krateo composition-dynamic-controller

The Krateo composition-dynamic-controller (CDC) is the Kubernetes operator that is instantiated by the Krateo core-provider to manage the specific CRs whose CRDs are generated by the core-provider. In practice, when a CR is created, the specific instance of Krateo composition-dynamic-controller checks if a Helm release associated with the CR already exists in the cluster [25]. If this is not the case, it performs an *helm install* operation using the values specified in the CR to create a new Helm release. This will practically deploy all the resources defined in the Helm chart (in the folder "/templates") using **Helm's templating engine**. This feature is particularly important in the context of the system described in this thesis and in particular for multi-cloud resource management as explained in section 2.3. However, if the Helm release does already exist, it instead executes an *helm upgrade* operation, updating the release's values with those specified in the CR, effectively updating the resources in the cluster. This mechanism will be leveraged as well for a particular feature of our system (i.e., "waiting logic") described in section 2.3.2. Finally, if and when the CR is deleted from the cluster, the instance of the Krateo composition-dynamic-controller performs an *helm uninstall* on the Helm release, effectively removing all the resources defined in the Helm chart from the cluster [25].

It must be said that Krateo composition-dynamic-controller has a set of additional features that are not described in this work such as multi-version support. Figure 1.5 shows the architecture of the Krateo core-provider and Krateo composition-dynamic-controller, as described and depicted in [26].

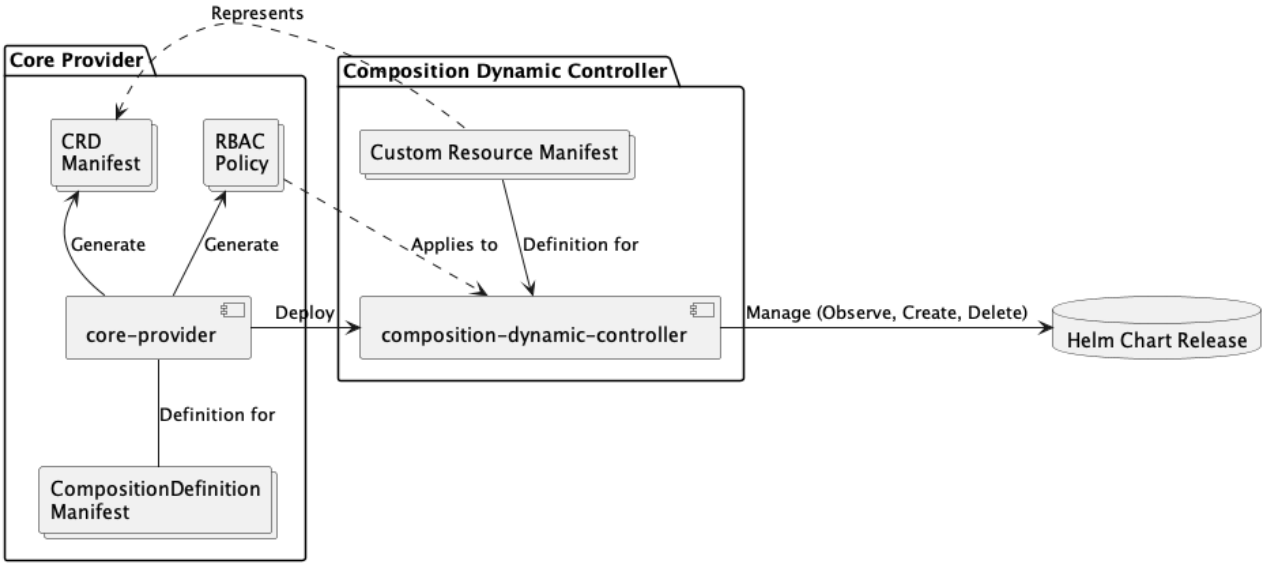

Figure 1.5: Krateo core-provider and composition-dynamic-controller architecture [26]

1.5 Open Policy Agent

Open Policy Agent (OPA) is an open-source general-purpose **policy engine** that enables unified policy decision-making across several types of environments [43]. OPA provides a declarative language called **Rego** enabling a paradigm known as "**Policy as Code**" [43]. 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: **Kubernetes admission control**, CI/CD pipelines, container images, microservices, etc.[43].

1.5.1 Policy as Code paradigm

According to AWS, Policy as Code (PaC) is a software automation approach which is similar to Infrastructure as Code (IaC) [53]. PaC helps assess company system configurations and validate compliance requirements through software automation [53]. 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, as what happened with the IaC approach [53].

OPA's generic definition of policy is: "*A policy is a set of rules that governs the behavior of a software service*" [47]. 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 specific 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 on an API server.
- Enforcing network security policies, such as blocking external access to sensitive internal services.
- Ensuring cloud 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.

Another interesting example is the use of OPA for **compliance automation for AWS infrastructure** [69]. In this case, OPA is used by the authors as a step in a CI/CD pipeline to enforce compliance policies on AWS infrastructure represented with the IaC tool CloudFormation [69]. 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 [43]. 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 [43]:

- **Centralized policy management**: policies are defined in a single location, ensuring uniform enforcement across all services of an organization.
- **Improved maintainability**: updating policies does not require modifying, recompiling or redeploying application code, reducing complexity, errors 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 requiring policy decisions are not tightly coupled.

1.5.2 OPA Gatekeeper

OPA Gatekeeper is a Kubernetes-native tool that extends OPA with **Custom Resources** and controllers to enforce policies within a Kubernetes cluster [45]. As a matter of fact, it provides a declarative approach to defining and enforcing policies using Kubernetes Custom Resources. This makes it a convenient choice for simple and standard policy enforcement scenarios in Kubernetes, such as RBAC (Role-Based Access Control), basic security compliance, and resource constraints [45]. However, while OPA Gatekeeper is **well-suited for simple use cases**, it presents heavy **limitations** when addressing complex policy requirements, particularly when policies involve **mutations** or require access to **external data sources** [46]. Indeed, 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 a 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 custom 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.

```
1 package kubernetes.admission
2
3 deny[msg] {
4   input.request.kind.kind == "Pod"
5   input.request.object.metadata.namespace == "restricted"
6   msg := "Pods cannot be created in the 'restricted' namespace."
7 }
```

Listing 1.1: Simple OPA Rego Policy

The second example, illustrated in listing 1.2 utilizes OPA Gatekeeper and therefore its CRs. By using a ConstraintTemplate CR, policies can be enforced with Kubernetes tooling, making them easier to distribute and manage in this context. 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.

```yaml
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 1.2: OPA Gatekeeper ConstraintTemplate

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

Listing 1.3: 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 [45]. 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 [46]. 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.

1.6 MLflow

MLflow is an open-source platform designed to facilitate and enhance the overall machine learning lifecycle. In particular, it offers tooling for **experiment tracking**, **model management**, and **model storage** [39]. It is compatible with various ML frameworks, including scikit-learn, PyTorch, TensorFlow, and XGBoost. In particular, in the case of our system, PyTorch is the ML framework used for the forecasting models making MLflow a suitable choice for model management. For what concerns model deployment, MLflow does not provide a built-in solution, but it is compatible with other tools like KServe which will be described in section 1.7.

1.6.1 MLflow Tracking Server

MLflow Tracking Server is the central component of the MLflow platform which is responsible for logging and storing model training data, parameters, and metrics. It enables machine learning practitioners (e.g., data scientists, machine learning engineers) to track experiments, compare results, and reproduce models easily in a collaborative environment. MLflow Tracking Server revolves around the concepts of **experiments**, which are collections of **runs**. Each run represents a single model training session, and it contains information such as parameters, metrics, and artifacts (e.g., model files). Selected models can then be registered in the MLflow Model Registry, which is essentially an optional subset of the selected models that are ready for deployment. Model selection can done with the aid of a user interface that allows ML practitioners to visualize and compare the results of different experiments, making it easier to select the best model for deployment. MLflow Tracking Server, as the name suggest, is a server that is constantly waiting for new data to be logged. Said data comes from the various training scripts that are executed by the data scientists or machine learning engineers which are run in training environments. In particular the training scripts must be instrumented with the **MLflow API calls** to log the data in the remote MLflow Tracking Server. This setting is very flexible since the training scripts can be run in any environment (e.g., local environment, Universities' HPC clusters), as long as they have access to the MLflow Tracking Server. It is deemed important to mention some of the MLflow API calls that are used in the training scripts: the *infer_signature* and *autolog* functions [39]. The *infer_signature* function captures the **input and output schema of the model**, which is important for the deployment of the model. The *autolog* function is used to automatically log the parameters and metrics of the model training session, without the need to manually log them. In particular, each supported ML framework has its own autolog function which is used to automatically log the parameters and metrics of the model training session. The final phase of a model training session is the automatic creation of a **self-contained directory**, named after the experiment and run IDs, that contains all the necessary files to deploy the model. In particular, the folder contains: the serialized model with model weights, the configuration files (*conda.yaml*, *requirements.txt*), and the *MLmodel* file that contains additional configuration, among which, the model signature. The self-contained folder is then automatically uploaded to the MLflow Tracking Server as an artifact. The following is an example of the structure of the self-contained directory created by MLflow after a PyTorch model training session:

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

1.7 KServe

KServe is an open-source model inference platform that extends Kubernetes with a set of CRDs to **deploy** and scale machine learning models in production environments [29]. KServe can be deployed in several ways, one of which is the "*Serverless mode*" which is built on top of Istio and Knative, leveraging their powerful capabilities such as automatic scaling. There are two main CRs that can be used to set up a model serving environment: ***ServingRuntimes*** (or ***ClusterServingRuntimes***) and ***InferenceServices*** [28]. The former are abstractions that define model serving environments, specifying the templates for Kubernetes Pods capable of serving particular model formats. The latter are actually leveraging the available ServingRuntimes to deploy the models in the system. KServe provides several out-of-the-box ClusterServingRuntimes for common model formats, such as TensorFlow, PyTorch, and XGBoost, which can be used to deploy models without the need to define and configure the runtimes themselves.

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

Table 1.2: Open Inference Protocol API endpoints specification [32]

1.7.1 Open Inference Protocol

Interoperability is key in a fast-moving environment as the one of machine learning and AI. Therefore KServe has introduced the **Open Inference Protocol specification** to standardize the communication between inference servers and clients. The Open Inference Protocol has been adopted by several inference servers, including NVIDIA's Triton and Seldon MLserver [32]. As mentioned in the previous section, the InferenceService CRs used in the system are leveraging the "kserve-mlserver" ClusterServingRuntime, which under the hood uses MLserver that is compliant with the Open Inference Protocol. The list of API endpoints specified by the Open Inference Protocol is shown in Table 1.2.

1.8 Cloud resource metrics

Metrics are data points that provide information about a system. This information could be related to performance, behavior, resource utilization or any other aspect of the system that could be relevant. In the context of cloud resource management, metrics are essential for monitoring the **performance and health of cloud resources**, enabling organizations to make informed decisions about resource allocation, scaling, and optimization. Focusing on the first use case of the proposed system (i.e., VM scheduling for carbon footprint optimization), carbon metrics could be used to determine the carbon footprint of a scheduled VM. How to measure the carbon footprint of a resource managed in the cloud is a complex and challenging task. The carbon footprint of a cloud resource is influenced by several factors, such as the power usage effectiveness of the data center where the resource is hosted, the carbon intensity of the electricity grid in the region where the data center is located, whether the data center uses additional energy sources off the grid, and finally the energy consumption of the resource itself. The last factor is probably the most difficult to measure, as a cloud resource is an **entity that is abstracted from the physical infrastructure**, and therefore it is not straightforward to measure its power consumption. In this first iteration of the system we are dealing for instance with virtual machines (VMs). On the other hand, an on-premises server, for instance, could be potentially equipped with physical sensors that measure the direct power consumption of the server, but this is not the case for any cloud resource, especially since this work is focused on the consumer side of the cloud, where the consumer does not have access to any physical infrastructure. Moreover, public cloud providers do not provide fine grained and real-time data about the carbon footprint of a resource, as they do not adhere yet to a common standard for carbon footprint calculation (e.g., Real Time Cloud proposed by the Green Software Foundation), like they instead do for the FOCUS standard for FinOps [14]. In this section we want to give an brief overview of several metrics and methodologies that can be used **estimate the carbon footprint of a cloud resource**. In addition, we will also discuss the **challenges and limitations** that can be encountered when dealing with this kind of task.

1.8.1 System performance metrics

System performance metrics are a category of metrics that provide information about the health and performance of a system. Usually these metrics are related to bare-metal servers or virtual machines. Other cloud resources that have a higher level of abstraction (e.g., containers, Kubernetes Pods, serverless functions) may expose a slighlty different set of metrics that fall into this category as well. This category of metrics are also useful for the "**day 2 operations**", which are operations that are performed after the resource has been deployed and monitored for a certain period of time, briefly described in section 2.6.10.

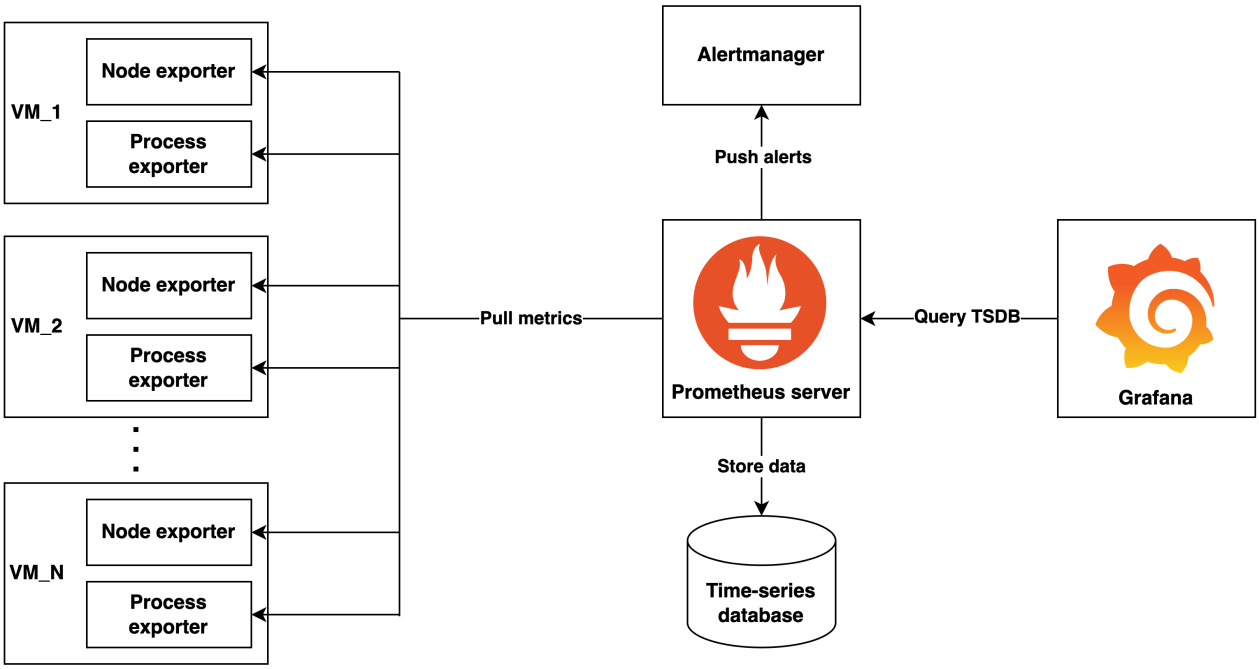

Figure 1.6: Standard Prometheus architecture for system performance metrics collection

System performance metrics comprise, for instance:

- CPU usage
- Memory usage
- Disk usage
- Network usage
- Processes related metrics

A standard strategy to collect these metrics is to use **agents** that are directly installed on the VMs and operate at the Operating System level. Some examples of this kind of agents are the **Prometheus Node exporter** and the **Prometheus Process exporter**. Figure 1.6 shows an example of a standard Prometheus architecture used to collect system performance metrics using the Node exporter and the Process exporter on a set of Linux VMs. In particular, the Node exporter collects metrics such as CPU usage, memory usage, disk usage, and network usage, while the Process exporter collects metrics about the various processes running on the VM (e.g., CPU and memory usage per process group). Metrics are pulled by the Prometheus server, which stores them in a time-series database and makes them available for querying and visualization (e.g., using Grafana). In addition, the Prometheus server can be configured to send alerts based on the collected metrics leveraging the Alertmanager component.

An example of how these metrics can be used for "day 2 operations" is the one put in place by the **Krateo Composable FinOps** component [28]. In this case, the system uses a set of Prometheus exporters and scrapers, configured through Kubernetes Custom Resources, to collect metrics about the VMs managed by the system. The optimizations are encoded in a set of Kubernetes Custom Resources that are used by the specific operators (e.g., *finops-operator-vm-manager*) to performs operations on the VMs, such as scaling up or down the VMs, stopping them during off-peak hours, etc [28].

As Microsoft Azure documentation describes, there are actually two different categories of metrics that can be collected from a VM in the cloud: the **guest-level metrics** and the **hypervisor-level or host-level metrics** [7]. The first category of metrics is collected by agents directly installed on the VMs, as described above. The second category of metrics is collected by **cloud provider APIs**,

and they provide information about the VMs from the hypervisor or host level. For instance, in the case of Azure Virtual Machines, this date is related to the Hyper-V host that runs the VM [7]. The difference between these two categories of metrics is that the guest-level metrics are more fine-grained and provide information about the actual resource usage of the VM (being collected at the OS level), while the hypervisor-level metrics are more high-level.
To support the collection of metrics, several cloud providers offer agents that can be installed on the provider-specific VMs they manage. One of the goal of these agents comparared to standard cloud-agnostic agents is to provide a more convenient experience for the user managing the VMs in the context of the cloud provider. Therefore these agents are usually more integrated with other cloud provider services and APIs. In particular, we can cite the following agents: **Azure Monitor Agent** for Azure VMs, the **Google Ops Agent** for Google Compute Engine instances, and the **Amazon CloudWatch Agent** for AWS EC2 instances.

1.8.2 Power consumption metrics

In the context of our use case, power consumption metrcis could be theoretically used to estimate the carbon footprint of a VM instance. However, Public Cloud Providers do not provide real-time data about power consumption of a VM instance.

Scaphandre is an open-source monitoring agent designed to collect energy consumption metrics of a system [55]. Scaphandre represents a promising solution only for on-premises servers, as it relies on the Intel **Running Average Power Limit (RAPL)** sensors to collect power consumption data. As a matter of fact, due to security and privacy concerns, public cloud providers do not expose to tenants the underlying RAPL sensors that Scaphandre relies on to track energy consumption [56]. Therefore, Scaphandre is unsuitable for our current use case.

Another interesting and quite mature project is **Kepler** [24], which is a tool designed to monitor the energy consumption of Kubernetes resources (e.g., Nodes, Pods). Being Kubernetes-centric, Kepler is not suitable for our first use case, as it does not provide real-time data about the power consumption of a generic VM instance. However, Kepler could be potentially used in a future iteration of the system.

1.8.3 Carbon metrics

As briefly mentioned in the beginning of this section, there is no adopted standard adopted by public cloud providers to calculate and expose the carbon footprint of a cloud resource. In section 1.2.1, we briefly described that the Green Software Foundation is working on a standard called **Real Time Cloud** that aims to provide a common standard for carbon footprint calculation. The goal would be to have a **real-time carbon metric** to be used for optimization and that this metric would be added to the set of metrics that are already provided by the cloud providers (e.g., CPU usage, memory usage). However, this standard is not yet adopted by any major public cloud provider which in turn provide only high-level monthly reports for carbon emission data. A brief summary, as reported by one of the proposers of Real Time Cloud [62], is shown in Table 1.3.

We deem interesting to mention some of the existing projects that can be used to estimate the carbon footprint of a cloud resource. **Cloud Carbon Footprint** [10] is a tool that uses cloud provider billing (i.e., AWS Cost and Usage Reports with Amazon Athena, GCP Billing Export Table using BigQuery, Azure Consumption Management API) and resource usage metrics to provide an estimation of the carbon footprint of a cloud resource. For live grid carbon intensity an integration with Electricity Maps API is supported [10]. Another tool within the GreenOps ecosystem is the **Aether** calculation engine [1] Currently only AWS and GCP are supported as it uses AWS CloudWatch and Google monitoring. There is not the possibility to use a real-time Grid Carbon Intensity Coefficient since carbon data is extrapolated from "governative" data reports [1]. A different approach to estimate the carbon footprint of resources, virtual machines in particular, is the one proposed by the **carbond** agent [71]. This agent must be installed on the VM and aims to provide a real-time estimation of the carbon footprint of the VM via a file-system API [71]. It should be investigated further whether this agent could be used in cloud environments, since those settings, as described above, do not provide access to the underlying hardware sensors that are usually used by agents to estimate the power consumption of the VM.

Cloud Provider	Project name	Scope	Geographical resolution	Carbon resolution	Time resolution
Amazon Web Services (AWS)	AWS Customer Carbon Footprint Tool	Account, EC2, S3, Other	Continental	0.001 Metric Tons of CO2e	Monthly
Microsoft Azure	Azure Carbon Footprint API Schema	Account, Service	Country and region	0.001 Metric Tons of CO2e	Monthly
Google Cloud Platform (GCP)	Google Carbon Footprint BigQuery Export Schema	Account, Project, Service	Country, region and zone	0.1 Kilograms of CO2e	Monthly

Table 1.3: Cloud provider sustainability reports

1.9 Multi-cloud resource management

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

1.9.1 Dynamic Virtual Machine placement

The work of Simarro et al. [72] back at the dawn of cloud computing (2011) proposed a multi-cloud architecture for the dynamic placement of Virtual Machines (VMs). The main objective of the system was cost optimization but this paradigm provides reliability and flexibility as well. The scheduling part is comprised of a "**cloud broker**" that is responsible for VM placement **transparent to users** providing a single uniform interface to the cloud resources. Users can provide to the system a "**service description template**" to specify the number of VMs to provision and some constraints. The cloud broker architecture is composed of two major components: the **scheduler** and the **cloud manager**. The former is responsible for placement decision across multiple cloud providers, while the latter is responsible for the actual management of the VMs in the cloud providers. More precisely, the cloud manager is represented by the OpenNebula (ONE) virtual infrastructure manager. OpenNebula is an open-source platform that aims to provide a unified management interface for multiple virtualization technologies and cloud providers [52].

1.9.2 Cloud service brokers

Cloud service brokers (CSBs) were described and categorized by Wadhwa et al. [76] in their work of 2013. A CSB is a system that acts as an **intermediary** between cloud service providers and consumers, providing a **unified interface** to manage cloud resources across multiple providers. The emerging market of cloud computing led to the proliferation of cloud services and providers, and by consequence the need for mechanisms to manage costs, capacity and resources.

An interesting CSB example in the literature is the **STRATOS** system by Pawluk et al. [70] proposed in 2012. The work can be considered a pioneer in the field of multi-cloud resource management since it can be framed in the first years of cloud computing but the proposed paradigms and concepts are relevant today. STRATOS tried to **avoid the assumption of resource homogeneity** and represented an initial attempt to provide a "**cross-cloud resource provisioning**" system. The proposed architecture enables the specification of high-level objectives that can be assessed in a standardized manner across different providers. The decision-making process is fully automated, shifting the decision point from deployment to runtime. Users first submit a topology document, triggering the Cloud Manager to communicate with the Broker for topology instantiation. The Broker then conducts the initial resource acquisition decision, optimizing the allocation of resources across multiple

providers (configured beforehand). Experiments indicate that distributing workloads across different cloud providers can reduce the overall cost of the topology. The approach taken by the authors primarily focuses on two objectives: **cost efficiency** and **avoiding vendor lock-in**. The application environment was deployed on public cloud platforms, specifically AWS and Rackspace.

A more modern example of a similar strategy is the one developed at IBM and presented by Lublinsky et. al [67], denominated as "**Kubernetes Bridge operator**" in which the authors propose a system that allows to submit various types of workloads from Kubernetes to a **arbitrary external resource manager** . In particular, they tackle the problem that not every workload can be deployed on Kubernetes and that especially scientific workloads might require specific and dedicated computing resources. They refer, for instance, to **HPC clusters**, dedicated AI clusters, quantum computing cluster, etc. Indeed they rely on the fact that specific resource managers (e.g., Slurm, IBM Spectrum LSF, IBM Quantum service, Ray, etc.) usually expose a **HTTP API**. The proposed operator deploys Kubernetes Pods which in turn interact with external resource managers through HTTP APIs for job submission and monitoring. Each Custom Resource, which represent the configuration of an external job, is associated with a specific Kubernetes Pod named "Controller Pod" (one per remote job) that is responsible for the interaction with the external resource manager (effectively acting as a proxy). Credentials for external systems are stored in Kubernetes Secrets and are mounted in the Controller Pod. In other words, the Bridge Operator is **generic and agnostic** and has the duty of managing Controller Pods which are are specifically tailored for a particular external resource manager. The authors also provide an exemplificative set of implementations of controllers in the form of container images to be used in the Controller Pods.

1.9.3 AI-based resource management in cloud computing

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

The work of 2022 by Khan et al. [66] provides a comprehensive review of the state of the art in the field of machine learning (ML)-centric resource management in cloud computing. Although this works focuses on the cloud provider side (**i.e., resource management in data centers**), it provides some interesting insights that can be leveraged also at different levels of the cloud computing stack. The authors highlight the fact that traditionally, only static policies were used for resource management in cloud computing, but the advent of ML techniques has enabled the development of more dynamic and adaptive resource management systems [66].

Tuli et al. in their work of 2021 [75] propose a system named **HUNTER** which stands for "Holistic resoUrce maNagemenT technique for Energy-efficient cloud computing using aRtificial intelligence". In this work, tasks are modeled as **containers instances**. From a infrastructural point of view, the system is composed of four main components:

- **Cloud Workload Management Portal**: a web-based portal that allows users to submit their workloads and specify the requirements (e.g., SLA constraints, QoS constraints, etc.).
- **Workload Manager**: responsible for the processing of incoming workloads.
- **Cloud Broker**: responsible for the allocation of resources to cloud worker nodes
- **Cloud Hosts**: set of cloud worker nodes, in the form of both private and public cloud.

The Cloud Broker is the core of the system and is further divided into three sub-managers: Service Manager, Cloud Data Centers Manager (CDC Manager), and Resource Manager. The Service Manager is responsible for SLAs and QoS constraints, the CDC Manager is responsible for the actual allocation (provisioning) and migration of resources to cloud worker nodes and resource monitoring and the Resource Manager is responsible for the scheduling of tasks and contains the actual sustainability models (e.g., energy, thermal, cooling.).

1.9.4 Policy-driven resource management systems

García García et al. (2014) propose **Cloudcompaas** [63], a Service Level Agreement (SLA)-driven platform for dynamic cloud resource management, focusing on the automation of resource provision-

ing, scheduling, and monitoring. The aim of the work is to provide a SLA-aware Platform-as-a-Service platform for the entire management of the cloud resource lifecycle [63]. The targeted resource to be managed spans from IaaS to PaaS and SaaS. Their approach leverage the WebService-Agreement specification, a standard for SLA negotiation in web services, to define the SLA between the cloud provider and the cloud consumer. Leveraging this representation, they propose a SLA-driven architecture with three main components: the **SLA Manager**, the **Orchestrator**, and the **Infrastructure Connector**. The SLA Manager is the entry point for users and essentially builds and register agreements starting from the user requirements. The Orchestrator, having a complete view of all the available cloud backends, performs the critical task of scheduling and resource allocation, considering the SLA constraints. The Infrastructure Connector is the component that actually interacts with the cloud providers, performing the actual resource allocation and deallocation (provisioning and deprovisioning). An interesting feature of the Infrastructure Connector is the "configuration step" in which arbitrary actions can be performed such as the **injection of a monitoring agent on a virtual machine**.

In the context of serverless computing environments, the work of 2021 by Mampage et al. [68] proposes a deadline-aware dynamic resource management system. The focus of the research is on both the provider and the consumer perspective, proposing a placement policy and "dynamic resource management policy" that aims to minimize the cost of the provider while meeting the requirement of the consumer (i.e., the **deadline**).

1.10 Carbon-aware systems for resource management

Having provided an overview of the existing works in the field of multi-cloud resource management and having introduced the GreenOps landscape, we now focus on the state of the art in the field of carbon-aware resource management. In particular, implementation choices and design patterns that can be leveraged for the development of a carbon-aware resource management system will be discussed.

The work by Sukprasert et al. [74] is a comprehensive analysis on the limits and benefits of the employment of geographical shifting and time shifting for cloud workloads. The authors highlight the fact that different workloads have different characteristics and therefore different degrees of flexibility. Those include, for instance: **execution deadlines**, **data protection laws**, and **latency requirements**. Therefore, carbon savings are constrained by a complex set of factors that need to be taken into account when designing a carbon-aware system.

The following sections will present systems that have been developed by various research groups to tackle the problem of carbon-aware resource management in cloud computing environments.

1.10.1 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 [73]. The system is defined as a multi-objective optimization problem that considers two factors: the **variable carbon intensity** and the **latency constraints** of the network. By evaluating the framework in real-world scenarios, the authors demonstrate that CASPER achieves significant reductions in carbon emissions (up to 70%) while meeting application **Service Level Objectives (SLOs)**, highlighting its potential for practical implementation in large-scale distributed systems. The authors of CASPER highlight the importance of considering the workload characteristics such as memory state, **latency** and **regulatory constraints such as GDPR**. The system is not adopting time-shifting since it is dealing with web services that are by their nature non-stopping workloads. The architecture is tied to scheduling Kubernetes resources inside Kubernetes clusters and does not consider external resource management.

1.10.2 CASPIAN

A research on carbon-aware scheduling in Kubernetes environments was conducted by the authors of CASPIAN (A Carbon-aware Workload Scheduler in Multi-Cluster Kubernetes Environments) [61].

The proposed system leverages Multi Cluster App Dispatcher (MCAD), a multi-cluster management platform, to provision workloads over distributed Kubernetes clusters. Caspian is a scheduling and placement controller which lives in a master cluster and interacts with the MCAD to provision workloads across multiple geographical distributed clusters. In particular, two main components are described: the **carbon tracker** and the **scheduler**. The carbon tracker is responsible for the periodic collection of carbon intensity data along with worker cluster locations. The scheduler, taking into account cluster information (i.e., carbon intensity, power efficiency, etc.), and workload requirements (e.g, **run time**, **deadline**, etc.), is responsible for the scheduling time and geographical placement of the workloads.

1.10.3 CarbonScaler

The work by Hanafy et al. [64] proposes a system that leverages the elasticity of batch cloud workloads to optimize carbon efficiency . The targeted workloads are for instance **MPI jobs** and **Machine Learning training jobs**. The system is entirely Kubernetes-based and it is composed of three main components: the **Carbon Profiler**, the **Carbon AutoScaler**, and the **Carbon Advisor**. The Carbon Profiler main duty is to estimate energy usage of jobs. The Carbon AutoScaler component is the core of the system and is a Kubernetes controller that leverage the **Kubeflow training operator**. Kubeflow is effectively used for the resource management of batch jobs. Finally, the Carbon Advisor simulates jobs execution in different configuration and allow the carbon reduction estimation.

1.10.4 A Low Carbon Kubernetes Scheduler

The work is focused on the extension of the Kubernetes scheduler ("kube-scheduler") for the ranking and filtering of the "greenest" region for the deployment of entire Kubernetes clusters [65]. The system is tailored and tested on Azure but can be extended to other cloud providers. Provisioning operations of the clusters are done by the system leveraging Kubernetes API or IaaS management APIs. Therefore Kubernetes is leveraged as a control plane to provision other Kubernetes clusters. An interesting feature proposed is the use of local air temperature and solar irradiance as tiebreaker for two datacenters with a similar carbon intense grid. The claim is that the solar irradiance has a bigger spread than the carbon intensity across global regions and that the local air temperature surrounding a datacentre affects the amount of energy needed for cooling.

1.10.5 Microsoft's Carbon-Aware Kubernetes strategy

Microsoft proposes a simple carbon-aware strategy for Kubernetes [38], integrating carbon intensity data into the scheduling process of Kubernetes Pods. The Kubernetes scheduler, which allows custom rules for assigning nodes to pods, can incorporate carbon metrics like the Marginal Operating Emissions Rate (MOER) as factors in placement decisions. A weighted distribution can be created by normalizing the MOER values across the available Nodes. These weightings are encoded in a YAML file and applied as a priority for the Scheduler, which out-of-the-box supports these kind of custom rules to extend the scheduling process. The claim is that by combining three elements (i.e., Kubernetes scheduler, carbon intensity data, and a weighting algorithm) any Kubernetes instance can be made carbon-aware, at least in a simplified way.

Generic component	Candidate solution
Representation of a generic workload	Krateo CompositionDefinition
Representations of constraint and objectives (e.g., SLAs, QoS)	Open Policy Agent policies
Cloud service broker / Orchestrator	Krateo composition-dynamic-controller with Helm templating engine
Modules to directly interact with the cloud providers	Kubernetes operators of cloud providers

Table 1.4: Mapping of main generic components for multi-cloud resource management to the components of the system proposed in this thesis

1.11 Multi-cloud resource management - major outcomes

Many of the concepts described in this chapter are leveraged in the design and implementation of our system. We can briefly list some of the major recurrent elements that are generally needed in order to build a system for multi-cloud resource management:

1. Representation of a generic workload
2. Representations of constraint and objectives (e.g., SLAs, QoS)
3. Cloud service broker / Orchestrator
4. Modules to directly interact with the cloud providers

1.11.1 Differences with respect to the state of the art

In the system proposed by this thesis, the elements needed for multi-cloud resource management (listed in the above section) are represented by **cloud-native components** in the Kubernetes ecosystem as described in the following chapters and as shown in Table 1.4.

From a general standpoint, the system proposed in this thesis is different from the state of the art analyzed for a number of reasons. First of all, the system was designed to be easily integrated in a production environment rather than be used in a "sandboxed" or simulation environment. The system is designed to be **flexible** and **agnostic** with respect to the type of resources managed and the cloud providers used. On the other hand, the works described in sections 1.9 and 1.10 are usually tied to one type of resource (e.g., VMs, K8s pods) or when they are not, like in the case of the "Kubernetes Bridge operator" [67], a specific controller must be implemented each time for each external resource manager. In our case, we leverage Kubernetes as a control plane to manage the resources of the cloud providers. We do not need to implement a specific controller for each cloud provider but we leverage the Kubernetes ecosystem of operators to manage cloud resources. We then leverage Open Policy Agent and the "Policy as Code" paradigm to encode the constraints and objectives of the system, which we believe is a more flexible and powerful approach than the one used in the works described in the previous sections.

2 System 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. A general description of some of the key components is provided to better explain their role in the system and the reasons for their inclusion. The system is designed to be modular, scalable, and extensible, enabling the integration of additional components as needed. We present a Proof of Concept (PoC) implementation of the proposed architecture.

2.1 Assumptions

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

2.1.1 Workload definition

In this work, workloads have been modeled as **Virtual Machines (VMs)**, representing the **primary use case** considered during the system's initial design phase. It is possible to define as "interrupible workloads" those workloads that can be stopped and restarted without losing the work done. For this work, only "**non-interruptible workloads**" are considered. This choice was driven by the fact that VMs are both a common and widely used cloud resource and one of the simplest to provision on multiple cloud providers and therefore the most important aspect to optimize. For the purpose of this work, we propose the following formalization where a VM is defined as a tuple:

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

with:

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

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

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

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

where:

- $MinCPU = 4$ vCPUs
- $MinRAM = 4$ GB
- $D = 1$ hour
- $DL =$ `"2025-03-20T23:59:59Z"` (i.e., the processing task P inside the VM must complete before this timestamp)
- $ML = 100$ ms (i.e., the maximum allowed latency)

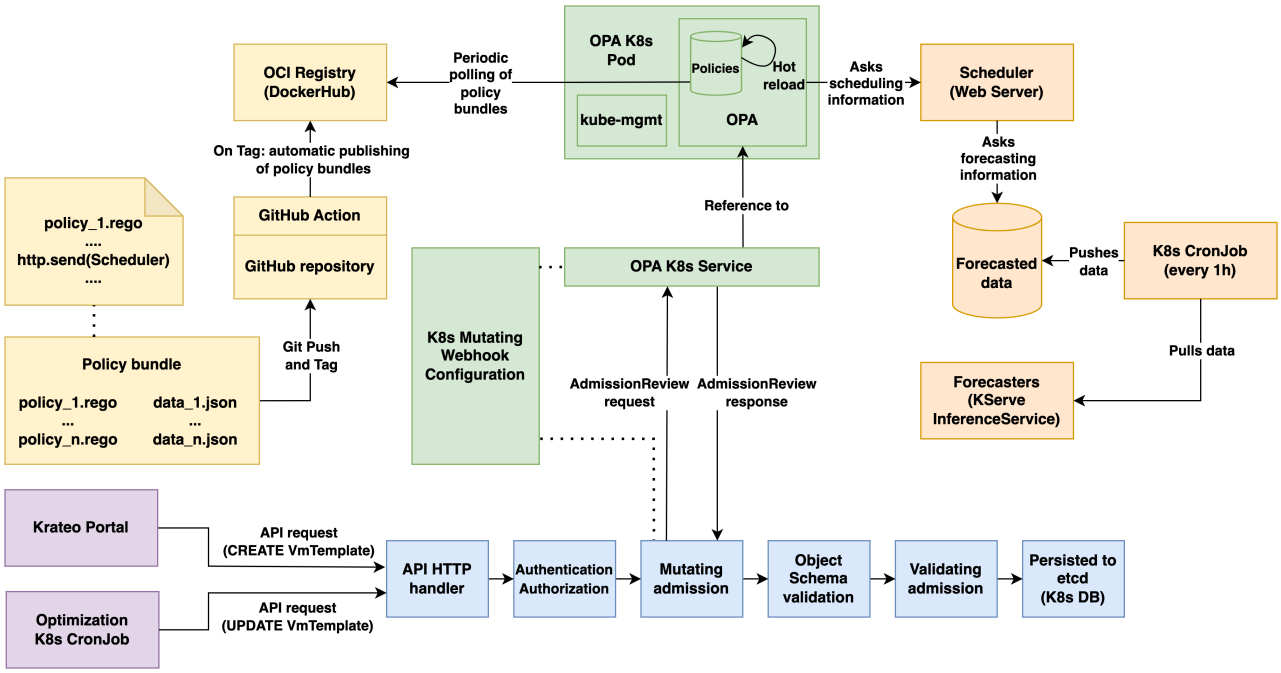

Figure 2.1: General architecture of the system

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**, and **Amazon Web Services**.

2.1.2 System limitations

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

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

Despite these constraints, the system architecture remains highly adaptable, and future enhancements could incorporate additional or alternative provisioning mechanisms. An example of an alternative implementation could be the **direct API interactions** with cloud providers to bypass operator limitations. As a matter of fact, usually cloud provider operators are leveraging these APIs under the hood to interact with the cloud provider's services. Another approach could involve the development of custom operators or controllers to manage specific resource types not supported by existing operators. For instance, Krateo PlatformOps provides the so-called "oasgen-provider" that aims to fill the gaps of missing or incomplete Kubernetes operators. This module is a Kubernetes controller that is able to generate Kubernetes CRDs and the related controllers from OpenAPI specifications [27].

2.2 System Architecture

Table 2.1 provides an overview of the main components of the system and their respective functions and Figure 2.1 illustrates the general architecture of the system.

All the components listed in table 2.1 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 2.6.5.

Component	Function
Krateo PlatformOps	Provides an abstraction layer for infrastructure orchestration, enabling declarative resource management and allowing integration with cloud providers with operators.
Cloud Providers Kubernetes Operators	Manages the provisioning and reconciliation of cloud resources within Kubernetes, ensuring the actual state matches the desired state.
Kubernetes Mutating Webhook Configuration	Intercepts and modifies API requests before they are persisted, allowing dynamic resource modifications with policy enforcement.
OPA Server	Evaluates policy decisions based on defined constraints and input data from Kubernetes API requests through the webhook configuration.
OPA Policies and Data	Define the rules and contextual information used by OPA to make policy decisions, namely scheduling information (for this use case)
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 2.1: Main components of the system and their respective functions.

2.3 Krateo PlatformOps integration

Krateo PlatformOps is leveraged in this system as a core component for **multi-cloud resource management**, as it allows declarative orchestration of cloud resources across different cloud providers leveraging Kubernetes as a control plane. This section highlights the differences between two different approaches that were considered for multi-cloud resource management:

- The **custom Kubernetes "Synchronization Operator"** approach (initially considered)
- The **Krateo PlatformOps** approach (adopted in the final system design)

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

2.3.1 Resource management: the custom Kubernetes "Synchronization Operator" approach

When dealing with multi-cloud resource management with Kubernetes as a control plane, a **synchronization and mapping mechanism** (i.e., broker) is required to bridge the gap between:

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

This requirement was also highlighted in the literature review described in section 1.11.

In this first approach, a custom Kubernetes "Synchronization Operator" would be responsible for the mapping and synchronization of the above resource types. Figure 2.2 illustrates the high-level architecture of the system with the custom operator approach. 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 implementation of a Kubernetes 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 operator.

Responsibilities of the "Synchronization 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, in particular:

- **scheduling region**: Defines where the workload should be placed.
- **scheduling time**: Specifies when the workload should be deployed.

These fields, if set, indicate a geographical placement and timing for the workload that 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 provider-specific 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. It must be said that this approach conflicts with the "single source of truth" paradigm, which is represented by the generic CR in this case.

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, including:

- 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.

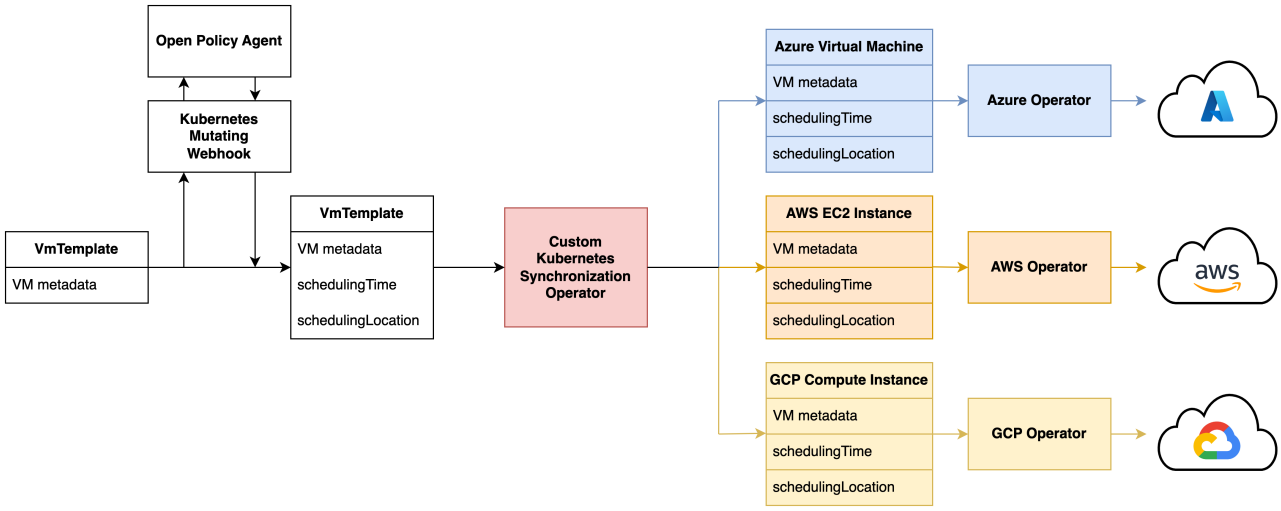

Figure 2.2: Multi-cloud resource management with Custom Kubernetes "Synchronization Operator" approach

- If the provider-specific resource is deleted directly on the cloud provider (e.g., on the provider console), 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, triggering 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 operator must take into account and 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 provider operator among the ones we used for the system (AWS, Azure, GCP) currently provides a dedicated field for the scheduling time. This means that the custom Kubernetes operator itself must handle a time scheduling logic, delaying CR creation until the scheduled time. If the operator, upon the creation of a generic CR, immediately creates the cloud-provider specific CR (without a "waiting logic"), the cloud provider operator will trigger and provision the VM immediately, ignoring scheduling constraints. Due to these limitations and complexities, we explored and leveraged an **alternative template-driven approach** using Krateo PlatformOps, described in the next section.

2.3.2 Resource management: the Krateo PlatformOps approach

In our final approach, we opted to logically 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, a custom Kubernetes 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 re-

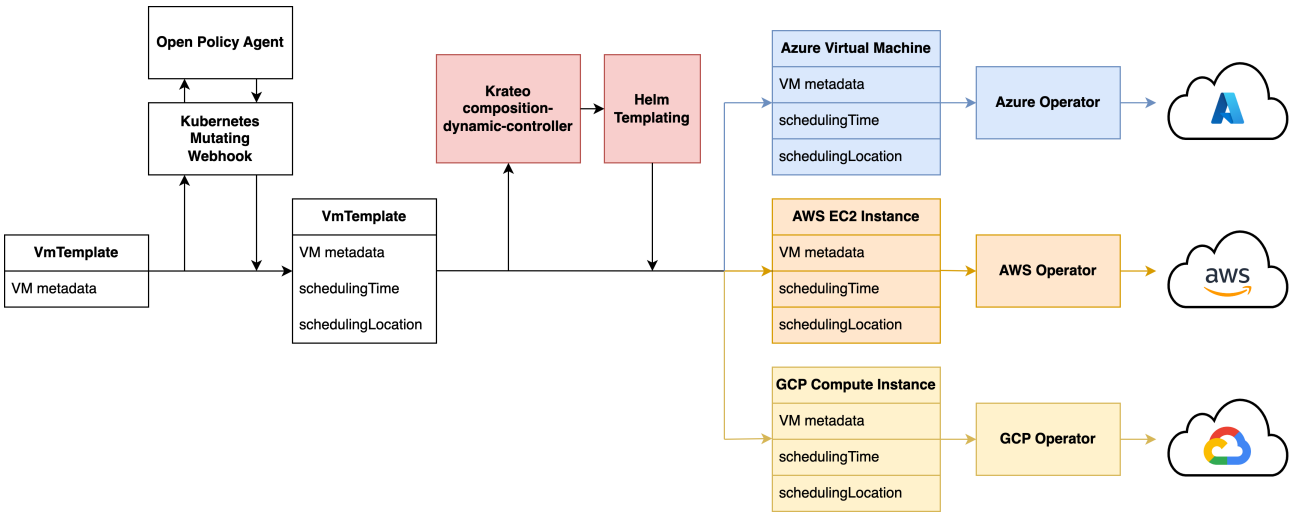

Figure 2.3: Multi-cloud resource management with Krateo PlatformOps approach

sources. More precisely, another Krateo component, the **Krateo composition-dynamic-controller** is leveraging **Helm Template Engine** under the hood to generate Kubernetes resources starting from Helm templates. By adopting an Helm-based resource generation, we can reduce the maintenance overhead and simplify the system architecture. Figure 2.3 illustrates, in a high-level manner, the revised system architecture for resource management with Krateo PlatformOps.

This approach, further described in this section, offers several advantages:

- Simplified resource management: Helm enables a standardized way to define resources without developing complex operator logic.
- Greater extensibility: By externalizing the logic a the custom operator (effectively removing it), future modifications and integrations on the system with additional cloud providers become easier.
- Reduced maintenance overhead: Custom operators typically require constant updates and refinements, especially if they are responsible for complex business logic.

Generic VM resource definition

In order to define a generic VM resource, we leverage Krateo Core Provider to define a **CompositionDefinition** resource that specifies the structure of the VM resource. Effectively a CompositionDefinition is a Kubernetes Custom Resource that defines the structure of a Composition Custom Resource (instance) which is an Helm chart comprised of Helm templates and values. As a matter of fact, a CompositionDefinition is referencing a versioned Helm chart that is stored in a Helm repository (e.g. on a Helm repository server).

```yaml
apiVersion: core.krateo.io/v1alpha1
kind: CompositionDefinition
metadata:
  name: vmtemplate
spec:
  chart:
    repo: vm-template
    url: https://leonardovicentini.com/helm-charts/charts
    version: 1.2.0
```

Listing 2.1: CompositionDefinition used in the system

The Helm chart referenced in the CompositionDefinition contains the Helm templates and values needed to define the resource named *VmTemplate*. Listing 2.2 shows an example of the VmTemplate values.yaml file, which defines the fields of the VM resource while listing 2.3 shows the corresponding JSON schema for the *values.yaml* file. The fields in the *values.yaml* file are used to define the VM

resource, including the VM name, CPU, memory, scheduling time, scheduling location, duration, deadline, and maximum latency and include the ones used in the formal definition of a VM resource in section 2.1.1.

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

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

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

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

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

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

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

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

Listing 2.2: values.yaml file used in the system

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

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

Listing 2.3: values.schema.json file used in the system

Generic VM to Cloud Provider Specific VM mapping

The Krateo composition-dynamic-controller (CDC) is responsible for creating the Composition CR by templating the Helm chart referenced in the CompositionDefinition. We leveraged Helm templating to dynamically generate cloud-provider-specific resources from a generic VM resource. As a matter of fact, the Helm chart contains all the **three cloud provider specific sets of templates** necessary for a VM provisioning but **only one set of templates is used at a time**. The CDC will generate the cloud-provider-specific VM resources based on the cloud provider specified in the generic VM resource. Therefore, for each generic VM resource, the CDC will use **only one set of templates** to generate the cloud provider specific VM resource. This "brokering mechanism" (routing through Helm templates) is implemented in the Helm templates using "**guards**" as reported in listing 2.4. The manifest will be created only if the cloud provider specified in the generic VM resource is the same as the cloud provider specified in the generic VM resource. This flexibility allows the system to be cloud-agnostic and to support multiple cloud providers as soon as the cloud provider specific templates are available in the Helm chart. The directory structure of the Helm chart is the following:

```
chart/
├── [several utilities files for mappings (omitted)]
├── templates/
│   ├── aws/
│   │   ├── instance.yaml
│   │   ├── subnet.yaml
│   │   └── vpc.yaml
│   ├── azure/
│   │   ├── networkinterface.yaml
│   │   ├── virtualmachine.yaml
│   │   ├── virtualnetwork.yaml
│   │   └── virtualnetworksubnet.yaml
│   ├── gcp/
│   │   ├── computeinstance.yaml
│   │   ├── computenetwork.yaml
│   │   └── computesubnetwork.yaml
│   └── _helpers.tpl
├── Chart.yaml
├── values.schema.json
└── values.yaml
```

```
{{ if hasKey .Values "provider" }}
{{ $provider := .Values.provider }}
{{ if eq $provider "azure" }}
...
apiVersion: compute.azure.com/v1api20220301
kind: VirtualMachine
...
```

Listing 2.4: Helm Template guards example

VM size selection is a crucial step in the VM provisioning process. Helm allows the definition of **helper functions** which resides in the *_helpers.tpl* file. In our case, we defined a helper function called ***findBestVmSize()*** that takes as input the CPU and RAM requirements of the VM and returns the best provider-specific VM size available. For instance: a generic requested VM specification could be (4 vCPU, 8 GiB of RAM) and this would be mapped to the "*Azure Standard_A4_v2*" VM size. A prerequisite for that is to have a mapping between the tuple (CPU, RAM) and the VM sizes (a string) for each cloud provider. This mappings are built using the cloud provider documentation and are stored in the Helm chart as a set of utilities files. Potentially, an automatic way to fetch the available VM sizes for each cloud provider could be implemented in the future. For our first iteration of the system we used a small subset of all the available instance types (i.e., 5 instance types for each cloud provider).

Scheduling time waiting logic

Helm template engine is also leveraged to handle the **scheduling time waiting logic**. As previously described in section 2.3.1, this logic is crucial for the system due to the fact that cloud provider operators do not support scheduling time metadata in their CRs and therefore as soon as a CR is created the cloud provider operator will provision the resource immediately. It must be remembered that is not a trivial task to implement this logic in a Kubernetes operator. In this case, we leverage a set of "**guards**" to effectively block manifest creation until the scheduling time is reached. As a result, only when the scheduling time is reached the manifest is created and applied by Helm and finally the resource is provisioned by the cloud provider operator. As a matter of fact, CDC will periodically perform an *helm upgrade* and if the scheduling time is reached the cloud provider specific resources will be created. As a consequence, the specific cloud provider operator will be triggered by the creation of the cloud provider specific resources and will provision the VM. Listing 2.5 shows an example of how guards are used to handle the scheduling time waiting logic and other scheduling constraints. We also make sure, for instance, that the generic CR contains a "provider" field that specifies the cloud provider where the resource should be provisioned.

```
{{ if hasKey .Values "provider" }}
{{ $provider := .Values.provider }}
{{ if eq $provider "azure" }}

{{ if hasKey .Values "schedulingTime" }}
{{ $schedulingTime := .Values.schedulingTime | toDate "2006-01-02T15:04:05Z" }}
{{ $now := now }}
{{ if $now.After $schedulingTime }}
...
apiVersion: compute.azure.com/v1api20220301
kind: VirtualMachine
metadata:
  name: {{ .Values.vmName }}
...
```

Listing 2.5: Scheduling time guard

In order to take into account the time needed for the cloud provider operator to provision the resource, additional logic could be added in the template to anticipate the scheduling time (e.g, using a *provisioningTimeOffset*).

2.4 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. More precisely, the system leverages Kubernetes operators from **Microsoft Azure**, **Google Cloud Platform**, and **Amazon Web Services**. Each operator, when installed on a Kubernetes cluster, installs a **set of CRDs** that represent cloud resources specific to the cloud provider. These CRDs can be used to define cloud resources in a declarative manner, in the form of Kubernetes CRs, allowing users to specify the desired state of the cloud resources they wish to manage. The other component installed by the operator is the **controller**, a software module that watches for changes to the CRs in the cluster and takes all the necessary actions to reconcile the actual state with the desired state. Under the hood the controller interacts with the cloud provider's API to provision, update, and delete cloud resources. 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 (*watch*), adjust, and manage external cloud resources within a Kubernetes environment. Inside the Kubernetes cluster lie the **real-time representations of the provisioned cloud resources**, which are managed by the operators. It must be noted that different cloud providers adopt **different design choices** for their Kubernetes operators and more in general for their overall cloud 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, VM templates)
- Networking components (e.g., virtual networks, subnets, security groups)
- Storage allocations (e.g., persistent volumes, cloud disks)
- Access management (e.g., resource groups, roles, authentication credentials)

For the purpose of this work we defined a **baseline infrastructure** for each cloud provider taken into account in order to have a common ground for the system to work. This baseline infrastructure is composed by the minimum set of resources needed for VM provisioning. Each public cloud provider has its complexities and nuances when it comes to managing cloud resources. In the following sections, we provide an overview of the minimum set of resources needed for VM provisioning on each cloud provider, as well as some of the specific configurations required for each resource. We deem useful to provide a table summarizing some of the key fields that are needed for VM provisioning. These fields are labeled with an ID which will be attached to the resources' listings in the following sections.

ID	VM field
1	Scheduling region
2	VM size
3	Operating System

Table 2.2: Key fields for VM provisioning

2.4.1 Azure Kubernetes Operator

Microsoft Azure provides a Kubernetes operator called **Azure Service Operator v2** (ASO). Currently, ASO supports more than 150 different Azure resources one of which is the **Azure VM**. An example of a Azure VM CR is the one illustrated in listing 2.6. The minimum set of resources needed for VM provisioning on Azure through Azure Service Operator is:

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

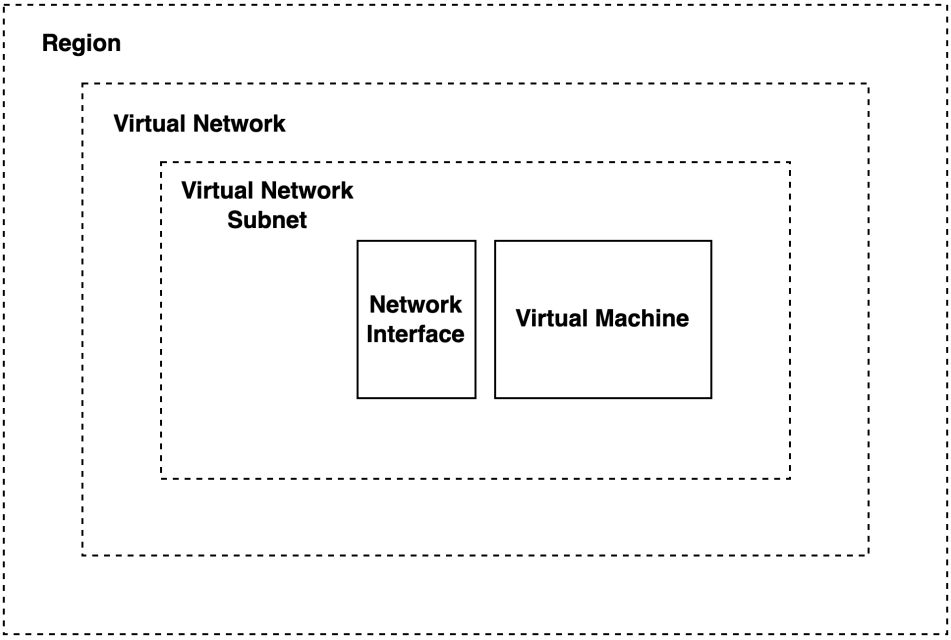

Figure 2.4: Minimum set of Azure resources for VM provisioning

```
apiVersion: compute.azure.com/v1api20220301
kind: VirtualMachine
metadata:
  name: {{ .Values.vmName }}
  namespace: {{ .Values.namespace | default "greenops" }}
spec:
  hardwareProfile:
    vmSize: {{ $vmSize }} #[2]
    location: {{ .Values.schedulingLocation }} #[1]
  networkProfile:
    networkInterfaces:
    - reference:
        group: network.azure.com
        kind: NetworkInterface
        name: {{ .Values.vmName }}-{{ $av.networkInterface.suffix }}
  osProfile: {{ $av.virtualMachine.osProfile | toYaml | nindent 4 }}
  owner:
    armId: {{ $av.resourceGroup.armId }}
  storageProfile:
    imageReference: #[3]
      publisher: Canonical
      offer: 0001-com-ubuntu-server-jammy
      sku: 22_04-lts
      version: latest
```

Listing 2.6: Azure VM Custom Resource

2.4.2 GCP Operator

Google Cloud Platform provides a Kubernetes operator called **GCP Config Connector**. The name of the virtual machine resource in GCP is **ComputeInstance**. An example of a GCP ComputeInstance CR is the one illustrated in listing 2.7. For what concerns the GCP Operator, the minimum set of resources needed for VM provisioning is:

- ComputeNetwork
- ComputeSubnetwork
- ComputeInstance

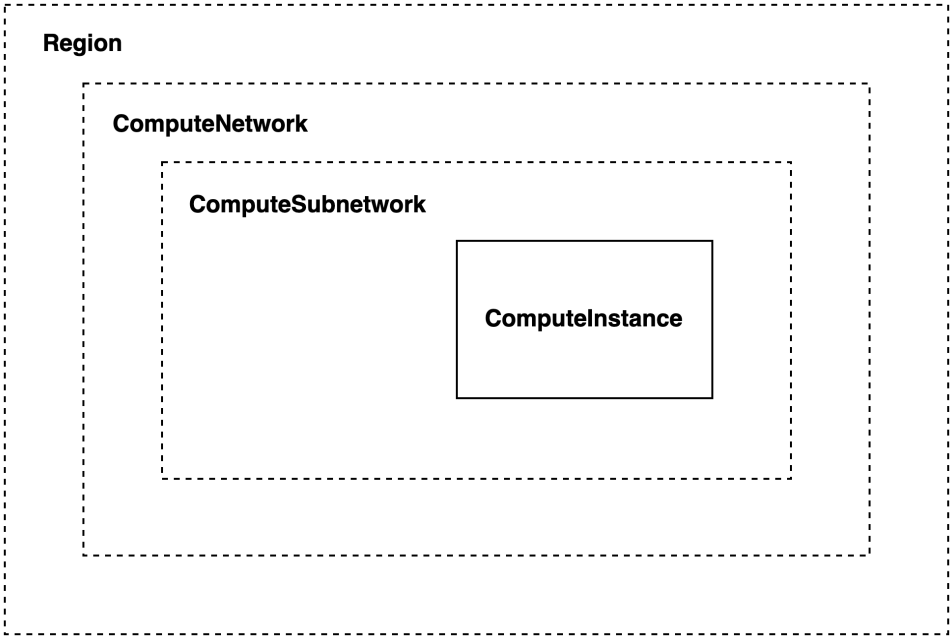

Figure 2.5: Minimum set of GCP resources for VM provisioning

We can mention the fact that some fields on GCP resource manifests are based on regions and some fields are based on avavilability zones. In order to select a zone, a helper function is used to select a zone based on the region.

```
apiVersion: compute.cnrm.cloud.google.com/v1beta1
kind: ComputeInstance
metadata:
  name: {{ .Values.vmName }}
  namespace: {{ .Values.namespace | default "greenops" }}
spec:
  machineType: {{ $vmSize }} #[2]
  zone: {{ $zone }} #[1]
  bootDisk:
    initializeParams:
      size: 24
      type: pd-ssd
      sourceImageRef:
        external: debian-cloud/debian-11 #[3]
  networkInterface:
    - subnetworkRef:
        name: {{ .Values.vmName }}-subnetwork
      aliasIpRange:
        - ipCidrRange: /24
          subnetworkRangeName: cloudrange
```

Listing 2.7: GCP ComputeInstance Custom Resource

2.4.3 AWS Operator

AWS provides an entire collection of operators that are part of the AWS controllers for Kubernetes (ACK) project. Each controller is responsible for managing a specific AWS service, such as EC2, EBS, RDS, S3, and more. In the context of this research, the EC2 controller is used to manage virtual machines (called EC2 Instances) provisioning on AWS. An example of a AWS EC2 Instance Custom Resource is the one illustrated in listing 2.10.

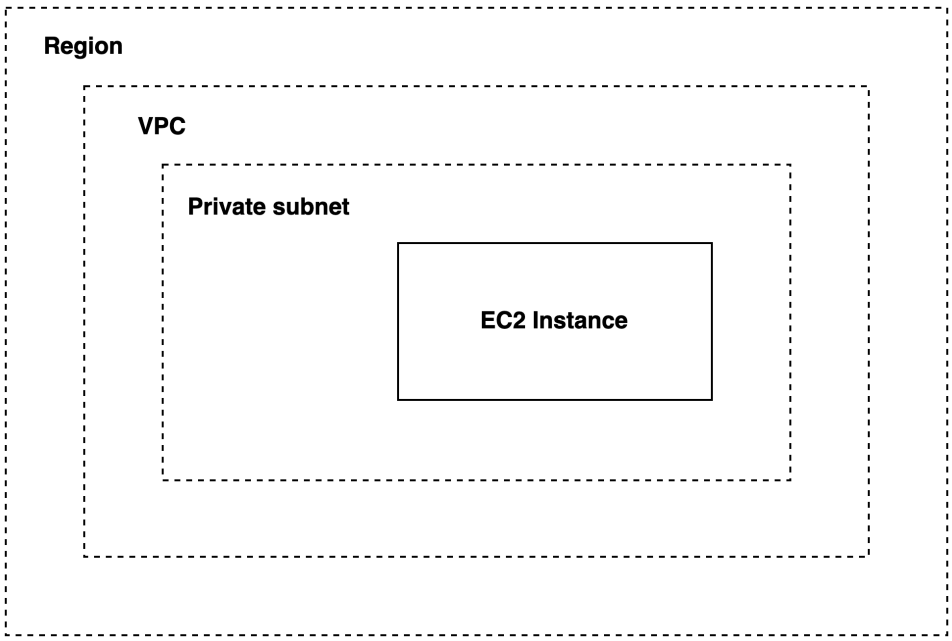

Figure 2.6: Minimum set of AWS resources for VM provisioning

The minimum set of resources needed for VM provisioning is:

- VPC
- Subnet
- EC2 Instance

Provider specific configurations

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 CR manifest. This limitation means that developers cannot directly link a resource (e.g., a VM) 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.

The Helm lookup function can be used to look up resources in a running cluster and its synopsis is: "lookup apiVersion, kind, namespace, name" [20]. In listing 2.8, 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 2.9.

In the case of Listing 2.9, the Subnet CR manifest directly references in a convenient way the VPC CR using its name and namespace since the operator is designed to support this type of relationship. As explained before, this is determined by operator design choices but our system is able to handle both scenarios.
Another important aspect to consider is that, when launching an Amazon EC2 instance, specifying

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

Listing 2.8: Helm Lookup example: dynamically resolving SubnetIDs

```
...
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 2.9: AWS Operator direct reference example

an AMI is **mandatory**. 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 [2]. In other words, AMIs serve as a blueprint for launching VMs in AWS. 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 installed on the AMI (e.g., Ubuntu, RHEL).
- 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.

The most important attribute for our system is the **Operating System** since it determines the software environment for the VM. 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. Nonetheless, this does not limit the applicability of this architecture since VMs with other operating systems could be also used. In order to select the most suitable AMI for a given VM, the system leverages a custom helper function to dynamically select the appropriate AMI ID based on the region and other parameters specified in the VmTemplate Kubernetes Custom Resource.

```
apiVersion: ec2.services.k8s.aws/v1alpha1
kind: Instance
metadata:
  name: {{ .Values.vmName }}
  namespace: {{ .Values.namespace | default "greenops" }}
  annotations:
    services.k8s.aws/region: {{ .Values.schedulingLocation }} #[1]
spec:
  imageID: {{ $imageID}} #[3]
  instanceType: {{ $vmSize }} #[2]
  subnetID: {{ (lookup "ec2.services.k8s.aws/v1alpha1" "Subnet" (.Values.namespace |
      default "greenops") (printf "%s-subnet" .Values.vmName)).status.subnetID }}
```

Listing 2.10: AWS EC2 Instance Custom Resource

2.5 Kubernetes Mutating Webhook Configuration

Kubernetes admission control is a set of mechanisms that control how Kubernetes API requests are processed by the API server before they are persisted in the cluster [34]. In this context, in addition to standard, compiled-in admission plugins, Kubernetes supports the use of additional admission plugins that are effectively extensions of the system and run as **webhooks** configured at runtime [34]. This means that the admission control logic can be extended dynamically without the need to recompile the Kubernetes API server or other Kubernetes components. Changes are applied at runtime to the running Kubernetes cluster, making the system more flexible and adaptable. Said plugins can be used to enforce custom policies and perform additional validation and mutation of Kubernetes objects before they are persisted in the cluster. We can classify webhooks in two categories: validating and mutating webhooks. Validating webhooks are used to validate the object and reject it if it does not meet the validation criteria. Mutating webhooks are used to modify the object before it is persisted in the cluster. We must highlight the difference of two entities: the webhook configuration and the actual webhook server which performs mutation or validation. The latter could be a custom server to apply custom mutation or validation logic or it could be a service ready to use out of the box like Open Policy Agent [43].

Figure 2.7 shows the Kubernetes admission control flow with the addition of mutating and validating webhooks. The flow can be summarized as follows:

1. The Kubernetes API server receives an API request.
2. Authentication and authorization phase take place.
3. Mutating webhooks are called first. If any of them returns an error, the request is rejected.
4. The Object Schema validation phase is executed.
5. Validating webhooks are called. If any of them returns an error, the request is rejected.
6. The object is persisted in the cluster.

A webhook service can be either a Kubernetes deployment with the related service inside the cluster like seen in the image 2.8 or can be a service deployed outside the cluster as long as it is reachable by the Kubernetes API server.

Figure 2.8 shows an example of a Kubernetes mutating webhook. The webhook server is responsible for receiving the AdmissionReview request, applying custom logic, and returning an AdmissionReview response to the Kubernetes API server describing the mutation to be applied to the resource. The custom logic in this simple example is to add a label “mutated: true” to the resource.

In our system, we configured a ***MutatingWebhookConfiguration*** to intercept CREATE and UPDATE Kubernetes API requests for VmTemplate CRs. In this case, the Kubernetes Mutating Webhook server that is the target of the MutatingWebhookConfiguration is OPA. OPA is used to assign scheduling decisions to VmTemplates based on policies. OPA then sends back to the API server the mutation (patches) to be applied to the VmTemplate CR.

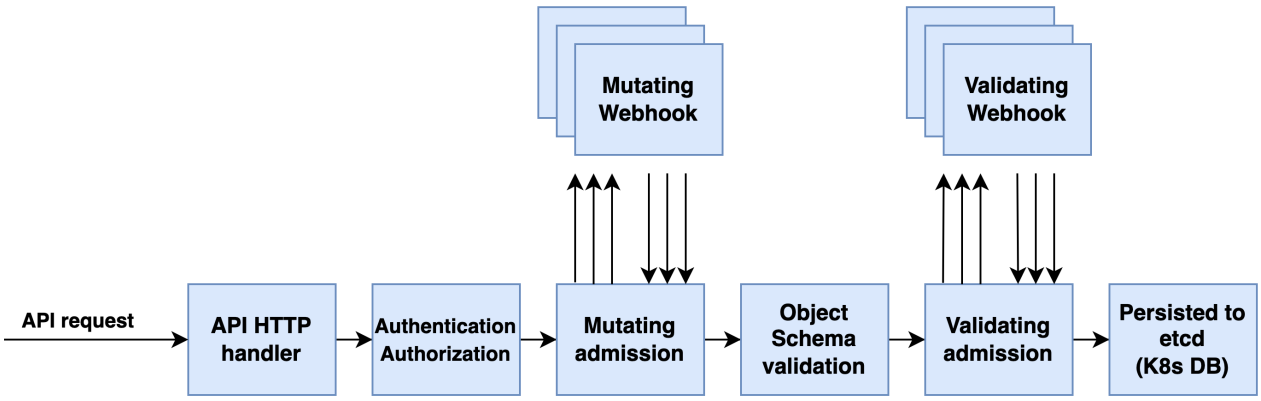

Figure 2.7: Kubernetes Admission Control

2.6 Open Policy Agent integration

In the context of our system, OPA and the Policy as Code paradigm are mainly leveraged to define **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 (QoS, data residency, etc.). In particular, OPA assumes the role of a **Kubernetes Mutating Webhook server** to apply scheduling decisions to VmTemplate Custom Resources based on policies.

2.6.1 OPA architecture overview

As mentioned in the background chapter, 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, represented in figure 2.9 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:

1. **OPA Server**: The core service that evaluates policy queries and returns decisions based on defined rules, contextual data and input data.

2. **OPA Policies**: Rules written in Rego language that define the logic to be enforced.

3. **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 2.6.4. These bundles enable version-controlled, centralized policy distribution, ensuring consistency and maintainability across distributed environments.

2.6.2 OPA and external data sources

In order to get data from external sources, OPA provides several options that can be chosen based on size and frequency of update [44]. For the use case of this work, external data to be pulled is the **scheduling decision** from the GreenOps Scheduler in a synchronous way. The most suitable option is to **pull data during policy evaluation** since scheduling decisions must be made in real-time. The data is pulled from the GreenOps Scheduler through OPA built-in functions and libraries [44]. Listing 2.11 shows an example of how the external data pull is implemented in Rego.

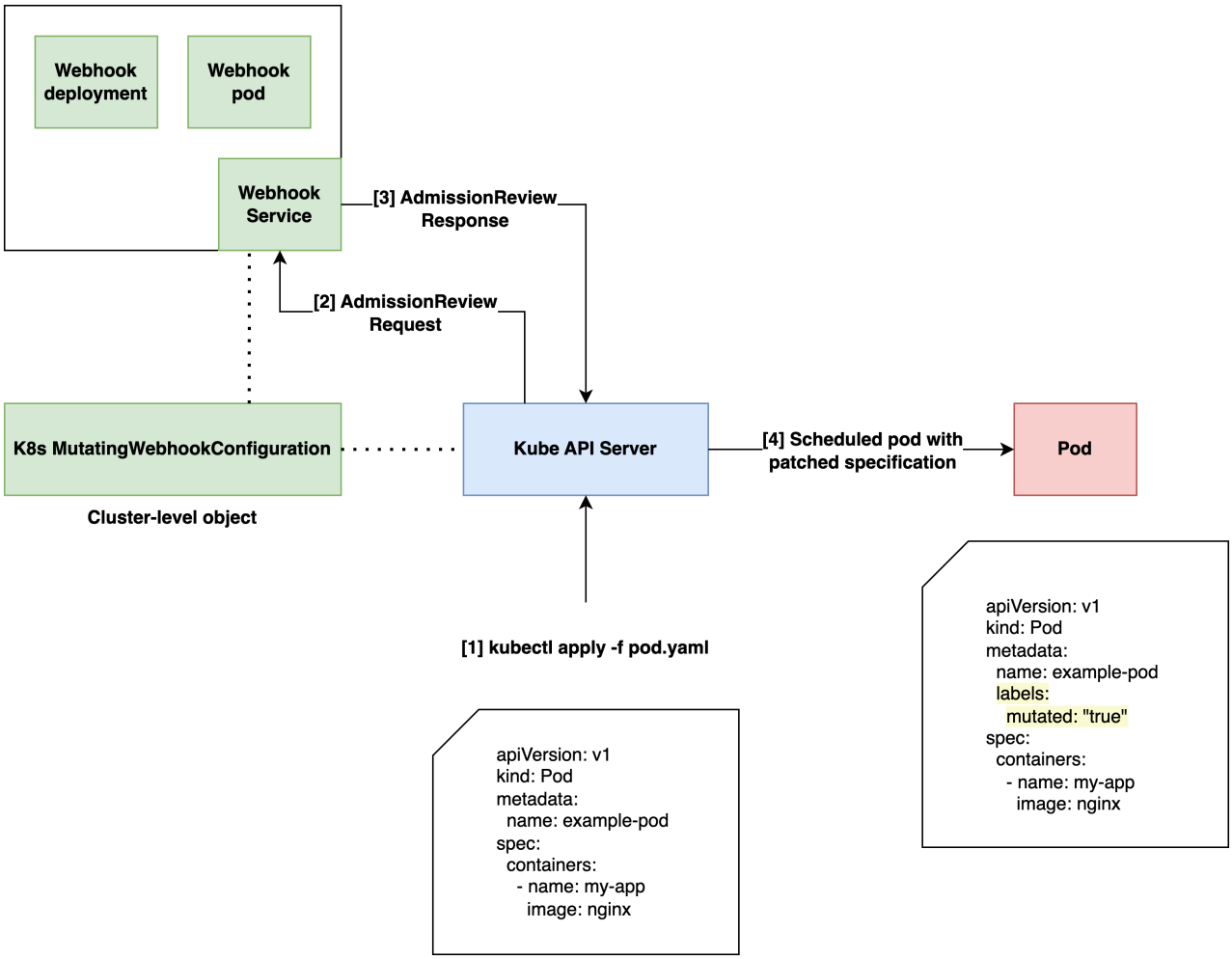

Figure 2.8: Kubernetes Mutating Webhook example [35]

```
scheduler_url := opa.runtime().env.SCHEDULER_URL # GreenOps Scheduler URL

scheduling_details := http.send({
  "method": "POST",
  "url": scheduler_url,
  "body": {
    "number_of_jobs": 1, # currently only one job (workload (VM)) can be scheduled
        at a time
    "eligible_regions": eligible_electricity_maps_regions,
    "deadline": deadline,
    "duration": duration,
    "cpu": cpu,
    "memory": memory,
    "req_timeout": 10 # seconds, scheduler wants to know the timeout to tune the
        execution time of the optimization
  },
  "timeout": "10s",
  "headers": {
    "Content-Type": "application/json"
  },
  "max_retry_attempts": 3, # retry 3 times in case of failure
})
```

Listing 2.11: OPA external data pull

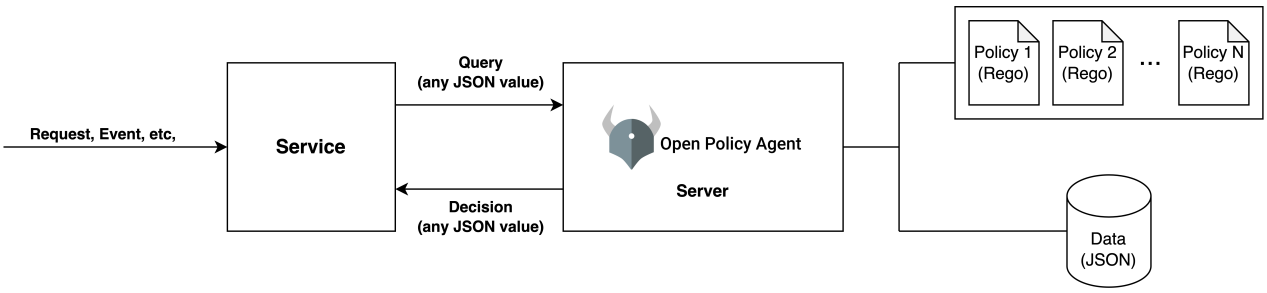

Figure 2.9: OPA architecture

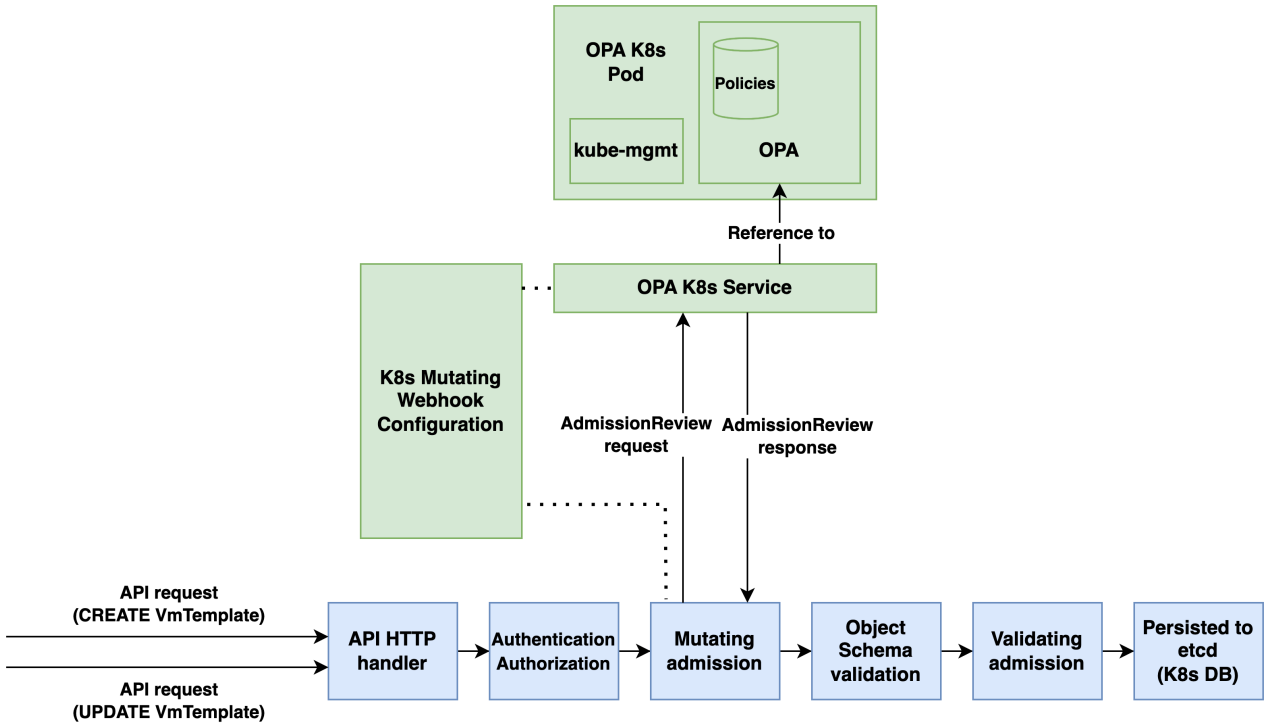

Figure 2.10: Kubernetes mutating webhook and OPA integration

2.6.3 OPA integration with Kubernetes

In the context of the Kubernetes admission control mechanism, 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, denying, modifying requests) is executed by Kubernetes' built-in admission control mechanisms. This is effectively one of the core design principles of OPA: to provide policy decisions to external systems, which then enforce the decisions based on their own logic [51]. This workflow is represented in figure 2.10 where the **AdmissionReview 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 [51]. The Kubernetes API server will then use the received AdmissionReview response for its decision.

In a Kubernetes deployment, an **OPA server 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 [36]. 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 the cluster for the decision [33]. 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 [33]. 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.

2.6.4 OPA policies

As OPA official documentation describes, when the Kubernetes AdmissionReview request from the webhook arrives, it is bound to the **OPA input document** and generates the default, "root", decision: *system.main* [43]. The root policy, in the case of Kubernetes admission control, is responsible for generating the AdmissionReview response in accordance with the Kubernetes API specifications. Indeed, 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 [43]. OPA policies are compiled before being evaluated, and any errors that occur during compilation are reported back to the caller. An example of these errors is when there are merge errors on contextual data [43]. For the purpose of this thesis, 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, in this example *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 while in a standard scenario, data is used to provide additional context to the policy evaluation process.

Listing 2.12 shows the Rego policy that enforces the rule described above while listing 2.13 shows the root policy that generates the AdmissionReview response. Listings 2.14 and 2.15 show respectively an example of an AdmissionReview request and response.

```
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 2.12: Rego policy for Pods registry

```
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 2.13: 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 2.14: AdmissionReview request

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

Listing 2.15: 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 comply with the defined policies,

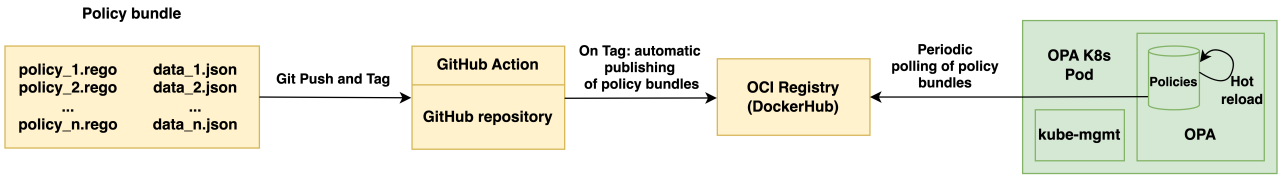

Figure 2.11: OPA policy bundles

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.

2.6.5 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 [48]. 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) [48]. One of the most convenient approaches is packaging policy bundles as **OCI (Open Container Initiative) images** [50] 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 and configurable 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 Common Vulnerabilities and Exposures (CVEs). 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") [48]. This is particularly useful in production environments where service availability is critical and downtime must be minimized. The overall process of policy distribution adopted and implemented in our system is illustrated in figure 2.11.

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

2.6.6 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.

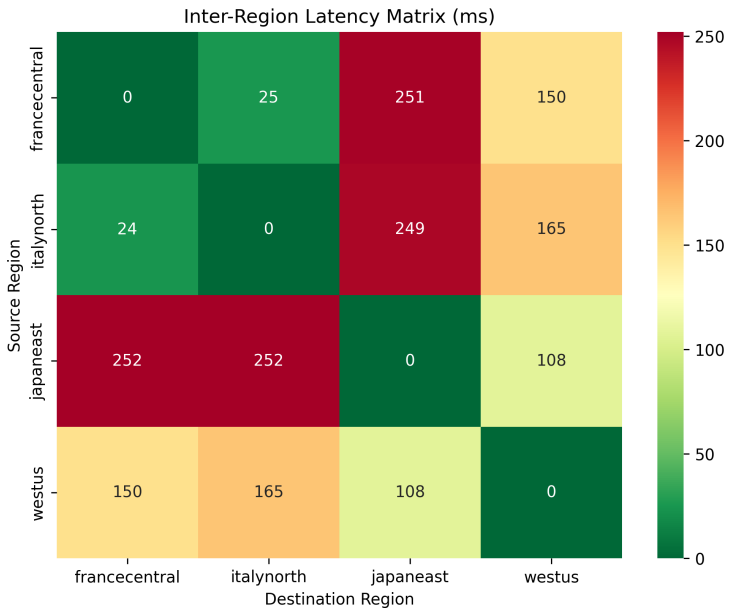

Figure 2.12: Latency matrix example (Azure regions subset)

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. As a matter of fact it would be possible to simply schedule all VMs in the region with the lowest carbon intensity, but this would not be a viable solution in a real-world scenario where performance is a critical factor. Therefore, by adding a latency constraint, the greenest region could not be selected if it does not meet the 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 2.12 illustrates a 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 2.16, 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.

For what concern data sources, cloud-specific inter-regional latency data were obtained in different ways for each cloud provider. Microsoft Azure provides monthly "Percentile P50 round trip times" between Azure regions in its documentation [6]. This data was automatically scraped, merged and used to build the latency matrix for Azure regions. For AWS a similar approach was used, but the data was obtained from a third-party website that provides latency data between AWS regions calculated using AWS Lambda functions [11]. For Google Cloud Platform, no official data was found, so the latency matrix was built using synthetic data that simulates realistic latencies between regions, similar to other cloud providers.

2.6.7 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 (EU). 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 countries. The intersection of these two sets defines the

```
{
  "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 2.16: Latancy matrix example encoded in JSON format

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.

2.6.8 Scheduling outcome policy

In the context of this work, the "Scheduling outcome policy" is a policy that determines the scheduling decision for a given workload based on the output of the GreenOps Scheduler which is queried in real-time, as described in section 2.6.2. The inputs are defined as follows:

- {CPU, RAM, duration, deadline, max latency}: set at request time
- eligible cloud providers: set at request time or in policies (configurable)
- origin region: set in policies
- GDPR compliancy: set in policies
- inter-region latency matrix: stored in policy data

While the outputs are:

- provider
- schedulingLocation
- schedulingTime

It is therefore a policy that has the duty to mutate (patch) the VmTemplate Kubernetes custom resource (Generic VM), adding the scheduling information (provider, schedulingLocation, scheduling-Time) to the resource. According to Kubernetes documentation, this can be done using "*patch*" and

"*patchType*" fields in the AdmissionReview response [34]. The "*patchType*" field must be "*JSONPatch*" and the "*patch*" field must contain a base64-encoded array of JSON patch operations to be applied to the resource. JSON Patch is a format for describing changes to a JSON document which avoids the need to send the entire document when only a part of it has changed. Effectively, only deltas are sent back to the requester which are themselves JSON documents. The format is defined in RFC 6902 from the IETF [23]. As an example, a single patch operation is the one shown in listing 2.17, where a new field "schedulingTime" is added to the resource. It must be said that whether the provider is randomly chosen from a subset, chosen due to additional logic or chosen by the user at request time is not a concern of the policy itself, but this choice is made by the system designer and implemented in the policy.

```
# schedulingTime is data coming from the GreenOps Scheduler
patchCode = {
  "op": "add",
  "path": "/spec/schedulingTime",
  "value": schedulingTime,
}
```

Listing 2.17: JSON Patch example

2.6.9 OPA Data mapping

OPA is flexible enough to handle **data mapping operations** 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 needed to be done and we deemed that inside the OPA policies was the best place to do it. In particular, these mappings are needed since the GreenOps scheduler knows only the notion of ElectricityMaps regions, and does 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.

The entire data mapping process can be broken down into the following steps:

1. Cloud provider selection (e.g., Azure, AWS, GCP): this determines the set of cloud provider regions.
2. Latency filtering: this step can only be done with cloud provider-specific latencies and determines the eligible regions.
3. GDPR compliance filtering (if enabled): this step ensures that only regions in the EU are selected.
4. ElectricityMaps regions mapping: this step maps the eligible cloud provider regions to ElectricityMaps regions.
5. Scheduling outcome: this step determines the scheduling region (ElectricityMaps region) based on the output of the GreenOps Scheduler.
6. The ElectricityMaps region is then mapped back to the cloud provider region as final step.

This process is illustrated in figure 2.13 and the Rego code in listing 2.18 illustrates the Rego functions used for data mapping between cloud provider regions and ElectricityMaps regions.

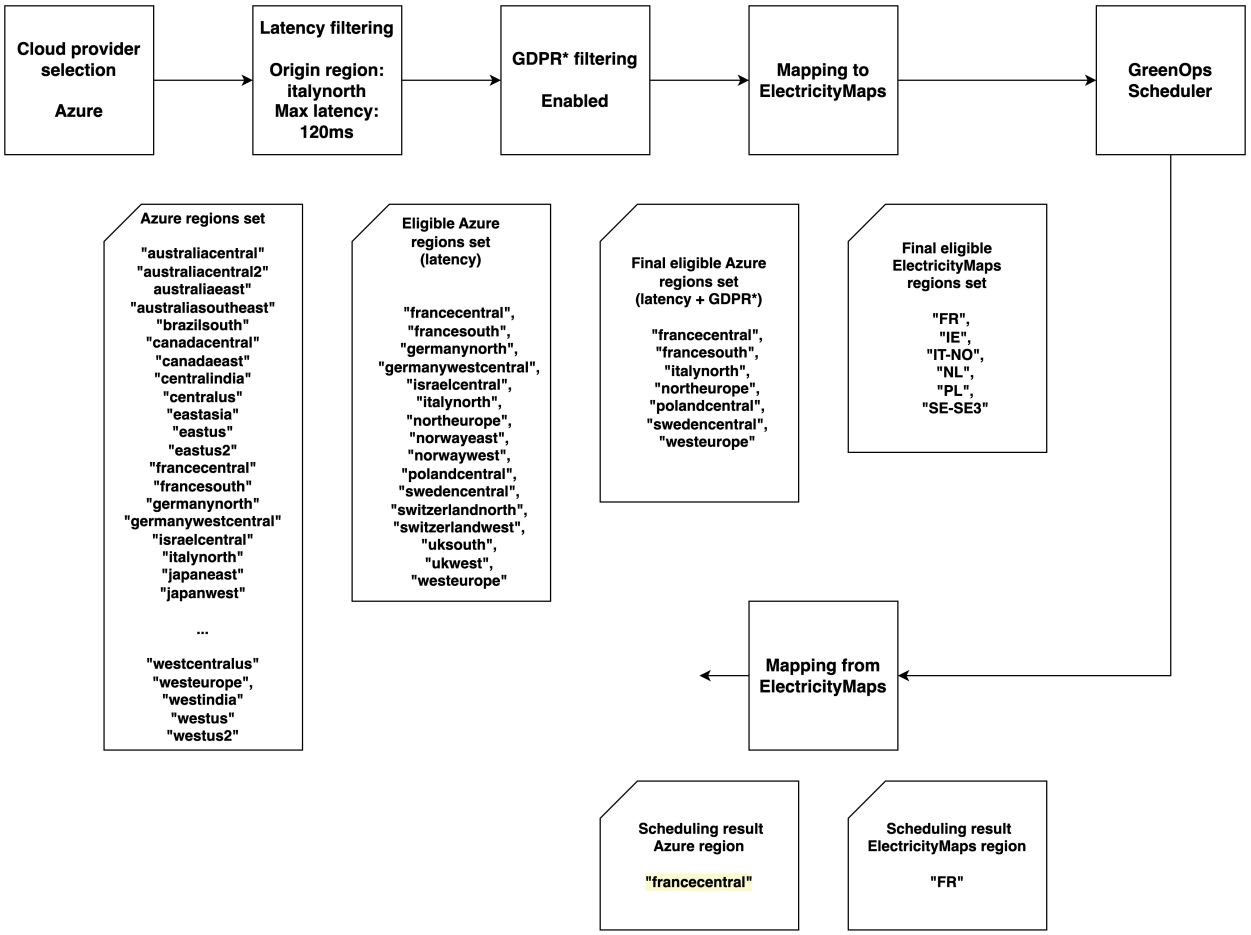

Figure 2.13: OPA Data mapping

```
# 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 2.18: Rego data mapping

2.6.10 OPA role within the system

In this section we describe the integration of OPA in the system. The entire architecture of the system and the integration of OPA is illustrated in figure 2.1. OPA has the role of **mutating webhook**

server which is consulted by the Kubernetes API server when a CREATE or UPDATE operation is performed on a VmTemplate Custom Resource.

The OPA server is responsible for evaluating the policies and returning the AdmissionReview response to the API server. The AdmissionReview response contains the decision of the policy evaluation (i.e. scheduling outcome) and the JSON patch operations to be applied to the VmTemplate resource by the Kubernetes API server. OPA is periodically polling the policy bundles from an external container registry (e.g., DockerHub) to ensure that the policies are up to date. The main policy, namely the "scheduling outcome policy", is responsible for determining the scheduling decision based on the output of the GreenOps Scheduler called in real-time at request time, as described in section 2.6.8. OPA is also responsible for data mapping operations between ElectricityMaps regions and cloud provider regions, as described in section 2.6.9. The system is designed to be highly flexible and extensible, allowing for the addition of new policies and data mappings as needed.

Day 2 operations

What was analyzed so far can be defined as "Day 1" optimization, i.e., the optimization of resources at deployment time. We can define as ***"Day 2 operations"*** all the management operations that are performed after the deployment of a resource. These operations comprise tasks such as scaling up or down a VM based on the load, stopping a VM during off-peak hours, or migrating a VM to a different region to optimize costs. As we previously described, the mutating webhook configuration is set on both the CREATE and **UPDATE** operations. A possible UPDATE operation trigger could be a Kubernetes Cronjob that attaches a label *"greenops-optimization": "919166400"* to the VmTemplate resource at a specific time of the day. This could be useful to trigger specific policies that are only applied during the day 2 operations. In addition to that, the UPDATE operation could be useful for VMs that have already been scheduled in a distant future (due to their deadline being very far in the future) to obtain a new scheduling decision based on new conditions (e.g. more recent carbon intensity forecast).

2.6.11 OPA advanced features

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

To ensure data integrity of policies (i.e., to prevent unauthorized modifications or MITM (Man in the Middle) attacks) and to provide a mechanism for verifying the authenticity of policies, OPA provides a **policy signing** feature [49]. This feature allow policy developer to digitally sign their policies bundles by adding a file named "*.signature.json*".

OPA also provides a more efficient way to distribute policies using **Delta Bundles** [42]. Normally, when a policy bundle is downloaded, OPA will download the entire bundle, erase and overwrite the current policies and data with the new ones. Delta Bundles are composed of a single "patch.json" file that contains a set of JSON Patch operations that can be applied to the current data to update it to the new version. Currently only data updates are supported ("data.json") and not policy updates ("policy.rego") [42]. This feature could be used in the context of the system described in this thesis to update in a more efficient way the contextual data used by the policies. For instance, in the event of a region being added or removed by a public cloud provider, a delta bundle could be used to add only the new region and update the existing latency matrix.

2.7 MLOps infrastructure

MLOps is the abbreviation of "**Machine Learning Operations**", and it broadly refers to a set of methods designed to improve workflow procedures and automate machine learning deployments. It enables the reliable and efficient management, maintenance and deployment of models at scale [59]. A MLOps infrastructure is not necessarily required for multi-cloud resource management, but it is believed that AI models will be utilized in the future more and more to get **scheduling and management decisions**, as evidenced by recent studies discussed in section 1.9.3. It is therefore deemed important to describe the MLOps infrastructure deployed in a Kubernetes environment and

leveraged by the system described in this thesis.

2.7.1 MLOps purpose

In a way, MLOps implements DevOps principles, tools and practices into typical Machine Learning workflows. Its main purpose is to effectively industrialize the machine learning models lifecycle, enabling faster model development, selection, and deployment to production compared to traditional manual approaches. Some principles of MLOps can be summarized as follows [59]:

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

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

2.7.2 MLOps general architecture

Figure 2.14 illustrates the general architecture of the MLOps infrastructure deployed in the Kubernetes environment. The architecture consists of two main components, described in the previous background sections: MLflow and KServe.

2.7.3 MLflow deployment configuration

In the context of our system, MLflow is deployed on a Kubernetes cluster with a loosely coupled architecture as can be seen in Figure 2.15, where the MLflow Tracking Server is decoupled from its storage: the metadata store (backend store) and the artifact store. This configuration is the most common in production environments, as it allows for better scalability and environment flexibility. The metadata store chosen for the system is **CrateDB**, while the artifact store is the **SeaweedFS** object storage service.

During the design phase, **alternative configurations** were considered. One of the alternative configurations was to use a single CrateDB instance as both the metadata store and the artifact store. This configuration was not implemented due to the lack of native support of object storage in CrateDB. A second theorized approach was to use a sidecar container with the duty of watching the MLflow Tracking Server local directory in the container filesystem (e.g., using *watchdog* Python package), packaging the model artifacts as an OCI image, and uploading them to an image registry. This approach could be implemented to offer an alternative to the MLflow Tracking Server artifact store in environments where adding a new storage service is not feasible.

2.7.4 Model deployment

Our specific use case requires the deployment of multiple models, each corresponding to a different region. This is true if we want to achieve better model performance compared to a single model that tries to predict the carbon intensity of all the regions. The current strategy adopted for the system is the following: **one model per region** is deployed, and a **generic model is used as a fallback** if the specific model is not available. This translates into the deployment of multiple InferenceServices, each corresponding to a specific region, and one InferenceService that acts as a fallback. This setting introduces a quite large amount of overhead in terms of resources, since each InferenceService sets up a whole new set of resources (e.g., large Kubernetes pods with underlying model serving environments) for each model. Figure 2.16 illustrates the configuration of the InferenceServices used to deploy the forecasting models in the system.

In particular, in our specific use case, since the forecasting models are PyTorch models, packaged

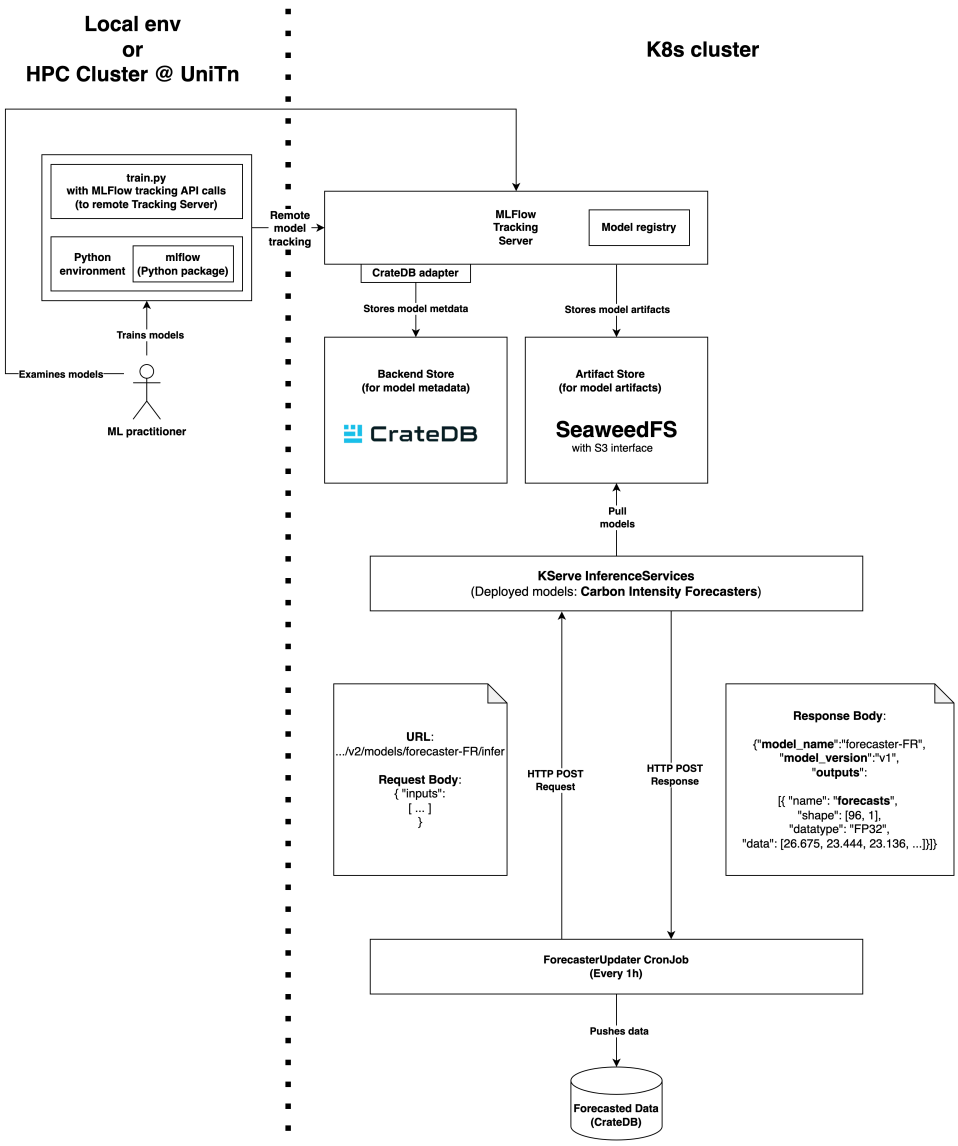

Figure 2.14: MLOps Architecture

by MLflow as described in the previous section, InferenceServices with "kserve-mlserver" ClusterServingRuntime are used. As a matter of fact, this runtime is the one that supports models packaged with MLflow [30]. Listing 2.19 shows an example of the InferenceService Custom Resource used to deploy a model in the system, namely the forectaser related to the "*IT-NO*" ElectricityMaps region (northern Italy)

```
apiVersion: "serving.kserve.io/v1beta1"
kind: "InferenceService"
metadata:
  name: "forecaster-IT-NO"
spec:
  predictor:
    serviceAccountName: sa-s3creds
    model:
      modelFormat:
        name: mlflow
      protocolVersion: v2
      storageUri: s3://mlartifacts/forecaster-IT-NO
```

Listing 2.19: InferenceService Custom Resource example

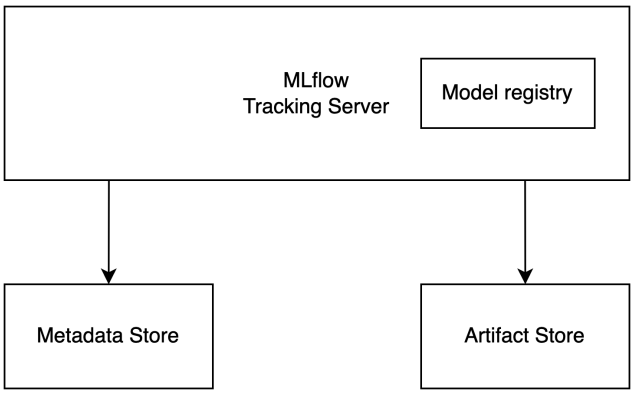

Figure 2.15: MLflow deployment configuration

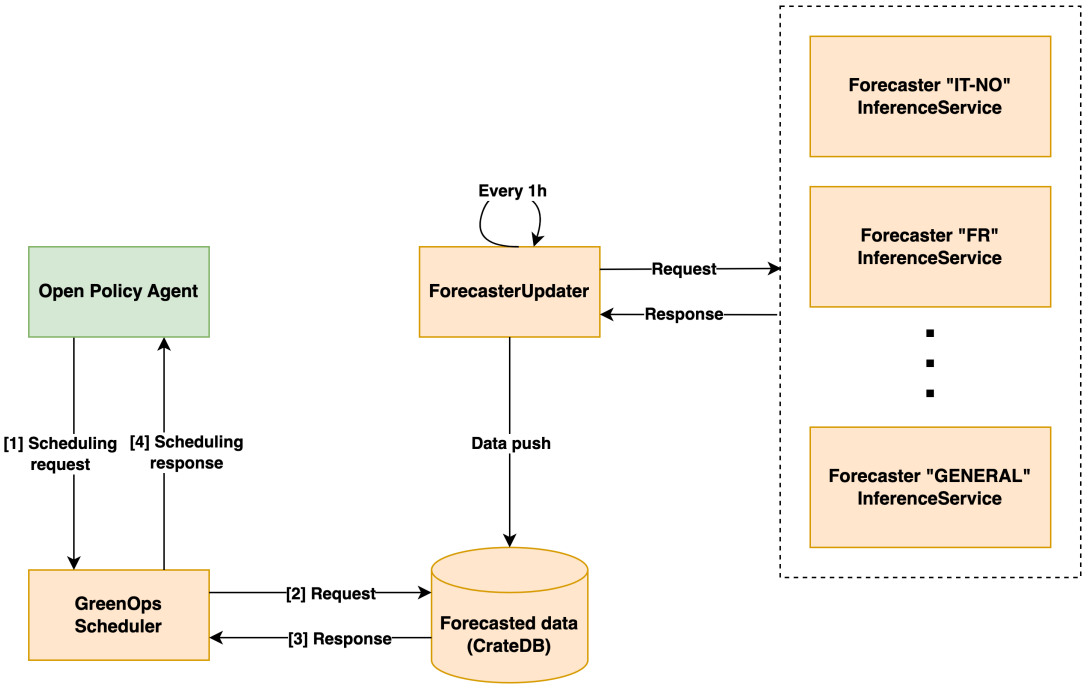

Figure 2.16: Model deployment configuration and GreenOps system integration

We can see that the InferenceService CR showed in 2.19 is quite simple and self-explanatory. The most important fields are the *model* field, which specifies the model format, the protocol version, and the storage URI. The storage URI is the location of the model artifacts in the artifact store, which in the case of the system is the SeaweedFS object storage service with S3 compatibility. At the location specified by the storage URI, the model artifacts are stored in the self-contained directory created by MLflow after the model training session as described in section 1.6.1.

Model compatibility

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

2.8 Impact framework potential integration

A potential integration with the Green Software Foundation's Impact Framework is envisioned for the system. Currently, said integration is not implemented, but it is deemed a valuable addition to the system in future iterations. As briefly described in section 1.2.1, the Impact Framework is a tool that allows the creation of pipelines for data extraction, transformation, and calculations. In addition, Impact Framework provides a database of cloud instances (a CSV file) with their respective specifications (e.g., CPU model, CPU TDP, GPU model, GPU TDP, memory, etc), which can be used to estimate the power consumption of a cloud instance. Listing 2.20 shows an example of a simple Impact Framework pipeline that extracts the metadata of a cloud instance. As a matter of fact the pipeline is composed of just one step that extract the metadata of the Azure *Standard_A1_v2* instance.

```
name: cloud-instance-metadata-extraction
description: null
tags: null
initialize:
  plugins:
    cloud-instance-metadata:
      path: builtin
      method: CSVLookup
      config:
        filepath: >-
          https://raw.githubusercontent.com/Green-Software-Foundation/if-data/main/
              cloud-metdata-azure-instances.csv
        query:
          instance-class: cloud/instance-type
        output: '*'
...
tree:
children:
  child:
    pipeline:
      compute:
        - cloud-instance-metadata
    inputs:
      - timestamp: 2023-08-06T00:00
        cloud/provider: azure
        cloud/instance-type: Standard_A1_v2
    outputs:
      - timestamp: 2023-08-06T00:00
        cloud/provider: azure
        cloud/instance-type: Standard_A1_v2
        cpu-cores-available: 52
        cpu-cores-utilized: 1
        cpu-manufacturer: Intel
        cpu-model-name: >-
          Intel Xeon Platinum 8272CL,Intel Xeon 8171M 2.1 GHz,Intel Xeon
          E5-2673 v4 2.3 GHz,Intel Xeon E5-2673 v3 2.4 GHz
        cpu-tdp: 205
        gpu-count: nan
        gpu-model-name: nan
        gpu-tdp: nan
        memory-available: 2
```

Listing 2.20: Cloud Metadata extraction example

As we can see from the example, the pipeline is composed of two main sections: the *initialize* section, where the plugins (pipelines steps) are defined, and the *tree* section, where the actual pipeline is defined. Among the output fields of the pipeline, we can see the *cpu-tdp* field, which represents the ***Thermal Design Power*** of the CPU of the Azure Standard_A1_v2 instance. This field could be used in a more complex pipeline to estimate the power consumption of the instance, and therefore its

carbon footprint. An example of such pipeline is described in the Impact Framework documentation [22] and is shown in Figure 2.17. A series of steps are performed to estimate the carbon footprint of a cloud instance, starting from the metadata extraction, to the power consumption estimation (with conversions), to the final carbon footprint calculation.

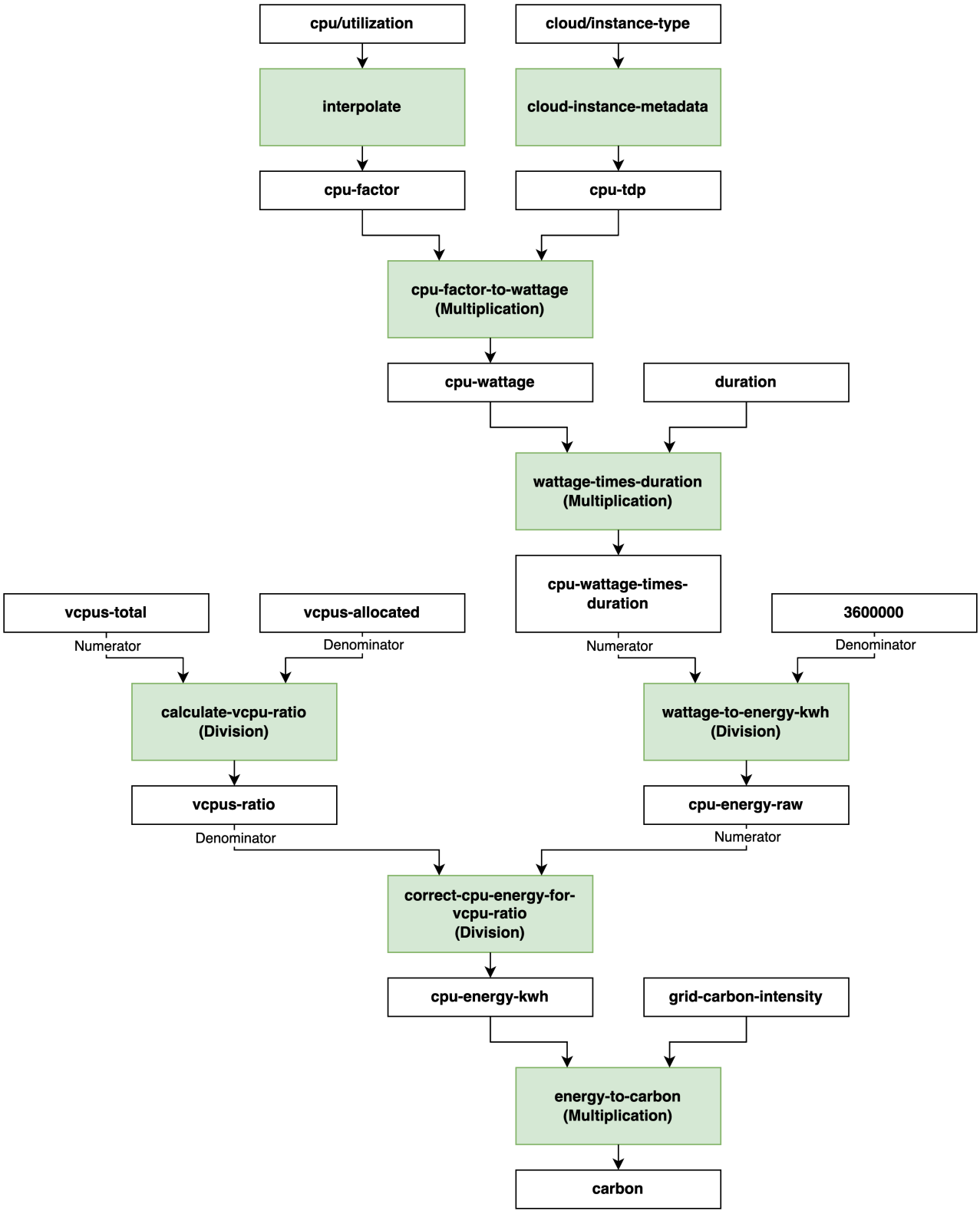

Figure 2.17: Impact Framework pipeline to estimate carbon footprint of a cloud instance

2.9 End-to-End workflow

In this section we will provide a complete end-to-end workflow of the system, from the user request to the final provision of the workload. First, we will describe the workflow in a high-level manner, and then we will provide a detailed sequence diagram (Figure 2.18) that illustrates the interactions between the various components of the system. We must note that the workflow described in this section is a simplified version of the actual system, as it does not include some low level details.

1. User (Developer, Data Scientist, etc) selects the VmTemplate Composition card on the Krateo Composable Portal (web-based user interface)
2. User fills in the VmTemplate Composition form with the required fields (e.g., MinCPU, MinRAM, Deadline, Duration, MaxLatency)
3. This triggers and Helm install of the VmTemplate Composition on the cluster
4. A VmTemplate Composition CREATE API request is sent to the Kubernetes API server
5. Authentication and Authorization checks are performed by the Kubernetes API server
6. The VmTemplate Composition CREATE API request is intercepted by a Kubernetes mutating webhook configured with OPA as webhook server
7. AdmissionReview request is sent to the OPA server
8. OPA evaluates the AdmissionReview request against the policies
   (a) Cloud provider is selected among the available providers
   (b) Eligible regions are calculated based on the selected cloud provider and the MaxLatency parameter
   (c) GDPR policy (if enabled) is evaluated and a subset of eligible regions is returned
   (d) A scheduling request is sent to the GreenOps Scheduler along with the eligible regions and a set of parameters
   (e) The GreenOps Scheduler returns a decision with the selected region and scheduling time
   (f) OPA maps the return ElectricityMaps region name to the Cloud Provider region name
   (g) OPA creates the JSON patch with the provider, schedulingRegion, and schedulingTime fields
   (h) OPA crafts and sends the AdmissionReview response to the Kubernetes API server
9. The Kubernetes mutating webhook receives the AdmissionReview response and mutates the VmTemplate Composition specification
10. Kubernetes API server perform resource validation
11. The VmTemplate Composition is persisted in the etcd database
12. A periodic Helm upgrade operation is done by Krateo composition-dynamic-controller
13. After schedulingTime is reached, an Helm upgrade operation will install the provider-specific manifests for VM provisioning on the Kubernetes Cluster
14. Cloud provider operator (e.g., Azure Service Operator) that is constantly watching for new provider-specific resources is triggered
15. The cloud provider operator provisions the VM (and required resources) on the cloud provider
16. The VM is up and running on the cloud

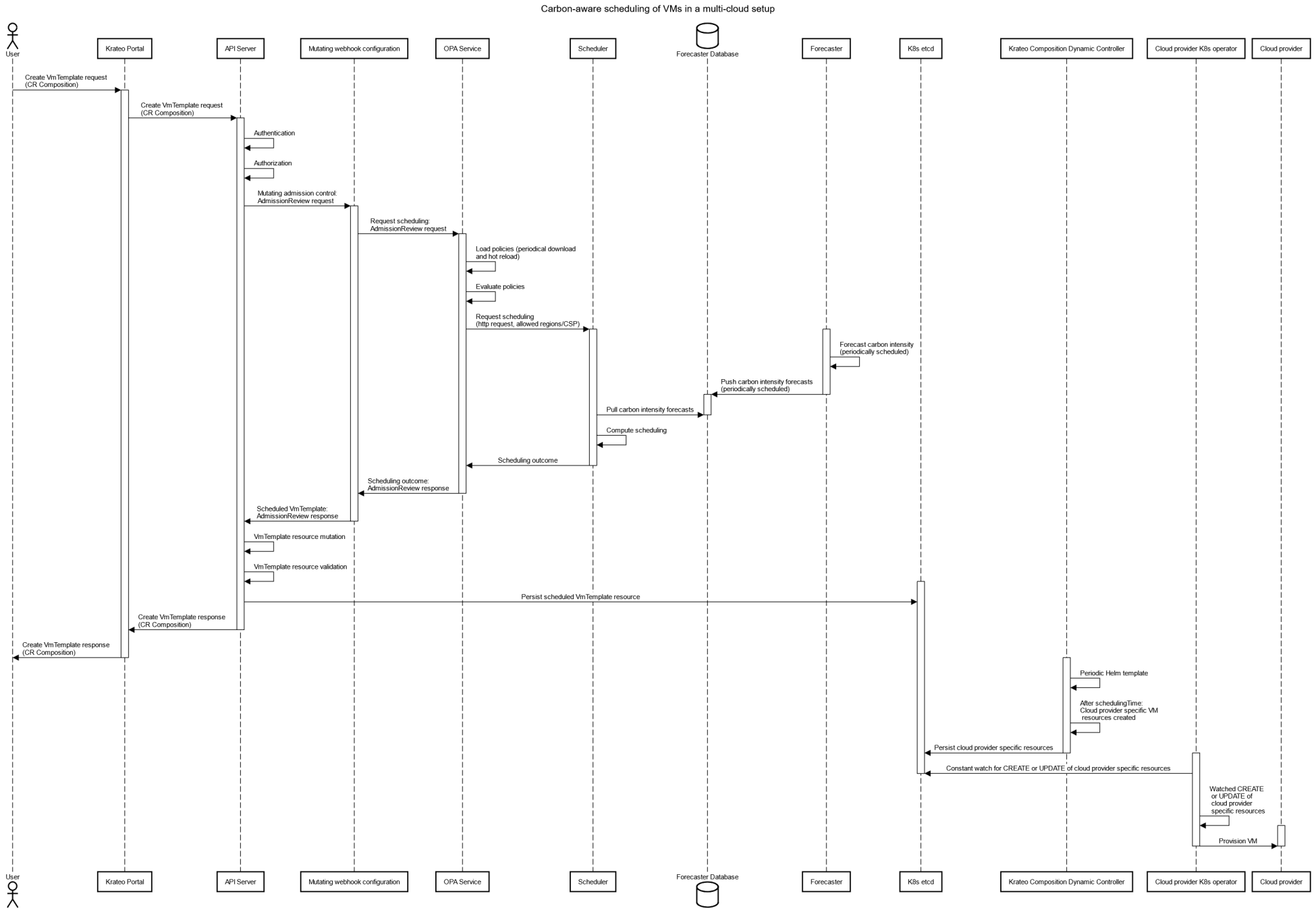

Figure 2.18: Sequence diagram of the end-to-end workflow

3 Conclusions and future work

This thesis has introduced a **system for resource management in multi-cloud environments, leveraging a policy-driven, Kubernetes-based architecture**. In particular, the guiding principle was to employ and integrate existing and well-known technologies to provide a flexible solution for cloud resource management. As a matter of fact, Kubernetes, Krateo PlatformOps, Open Policy Agent are cloud-native technologies that have flexibility and extensibility as their core principles. The focus of the work proposed in the thesis was on **optimizing the scheduling of virtual machines to minimize carbon footprint**, effectively demonstrating how GreenOps principles can be integrated into a cloud resource management system. Thanks to a comprehensive end-to-end integrated test, the system was successfully deployed and validated on a Kubernetes cluster, proving its feasibility in a real-world multi-cloud setting. The integration of Open Policy Agent for dynamic decision-making (i.e. real-time scheduling outcome), the use of machine learning models for carbon intensity forecasting deployed within the same Kubernetes environment adopting MLOps principles, the enforcement of policies encoding Quality-of-Service requirements (e.g., latency constraints) and compliance with exemplificative regulatory policies (e.g., GDPR-like policy) highlight the system's adaptability to a wide array of requirements and constraints of organizations that operate in multi-cloud environments. However, we believe that there is a potential for several improvement and extensions to the system, which are discussed in a following section of this chapter. First and foremost, the current iteration of the system focuses only on VM scheduling, but expanding the system to support additional cloud resources, such as serverless computing, containerized workloads, and databases is a natural next step as we described in section 3.2.2.

3.1 End-to-end integrated test

A comprehensive end-to-end integrated test has been carried out on a Kubernetes cluster. In particular, the test has been performed on a Azure AKS cluster, which is a managed Kubernetes service provided by Microsoft Azure. The cluster had a maximum of 4 nodes (autoscaling enabled) with 4 vCPUs and 16 GiB of RAM each. During the test, a **comprehensive deployment guide** has been prepared to guide future deployments of the system. The guide is divided into the following sections:

- **Part 1 - Cloud Providers' Operators**
- **Part 2 - MLOps stack**
- **Part 3 - VmTemplate CompositionDefinition, OPA and Kubernetes Mutating Webhook**
- **Part 4 - GreenOps Scheduler**

It must be noted that the order of the parts is important, as the system has dependencies between the components.

The test has been successful, as we were able to perform the following operations:

- Deployment of the operators for the three cloud providers (AWS, Azure, GCP) on the Kubernetes cluster.
    - Testing of the operators by creating provider-specific resources (e.g., AWS EC2 instances, Azure Virtual Machines, GCP Compute Instances) as test resources.
    - Verification of the provisioned resources on the cloud providers' consoles.
- Deployment of the MLOps stack
    - Deployment of CrateDB using the CrateDB operator (model metadata store).

    - Deployment of SeaweedFS using a Helm chart (model artifact store).
    - Deployment of MLflow tracking and serving with a custom Helm chart.
    - Deployment of KServe using a Helm chart.
    - Deployment of machine learning forecasting models on KServe using InferenceServices.
- Installation of a VmTemplate CompositionDefinition
    - Verification of the related card on the Krateo Composable Portal (Web User Interface).
- Deployment and configuration of Open Policy Agent.
- Set up of a policy bundles pipeline with GitHub Actions.
- Configuration of the Kubernetes mutating webhook.
- Deployment of the GreenOps scheduler.
- Simulation of user requests for generic resources (VmTemplates) on the Krateo Composable Portal.
- Verification of the creation of VmTemplate resources on the Kubernetes cluster.
- Verification of the mutation of the VmTemplate resource by the Kubernetes Mutating Webhook, with OPA integration, to obtain scheduling outcomes from the GreenOps scheduler.
- Verification that, when the scheduling time is reached, the provider-specific resources are provisioned on the cloud providers.

3.2 Future improvements

Since the system is designed with flexibility in mind, there are several possible improvements that could be made to the system. In addition to that, it must be remembered that the work proposed in this thesis is a first iteration of the system. This section describes some of the possible improvements that could be made to the system.

3.2.1 Day 2 operations

In the previous chapters we have focused on a use case: VM scheduling (placing) that can be considered as a "Day 1 operation". In section 2.6.10 we have briefly described the concept of "Day 2 operations" and the fact that the proposed system is designed to support them as well. Indeed, this is possible thanks to the flexibility of Kubernetes admission control and OPA. In particular, currently the Kubernetes mutating webhook is configured to intercept both the creation and the update of a VmTemplate generic resource. This means that the system can be potentially used to perform operations such as **scaling up or down a VM** as long as this logic is implemented in OPA policies and the specific operator is capable of performing the operation.

3.2.2 Support for other resources

As described in the previous chapters, the system currently supports only VMs but it is designed with **flexibility as a guiding principle**. In future iterations, the system could be extended to support other resources, such as serverless functions, databases, Kubernetes clusters, and other cloud resources. The requirements for enabling support for other resources are:

1. the definition of generic resources (VmTemplate-like) with Krateo core-provider
2. the configuration of a set of Helm templates to define provider-specific resources
3. the deployment of specific operators for the management of resources

For example, the system could be extended to support **serverless functions** by defining a ***FunctionTemplate*** resource and deploying a set of operators that manage the lifecycle of the functions. For instance, AWS Lambda functions could be supported by deploying the ACK service controller for AWS Lambda. Listing 3.1 shows an example of a manifest of a AWS Lambda function that could be rendered by the system [12].

```yaml
apiVersion: lambda.services.k8s.aws/v1alpha1
kind: Function
metadata:
    name: $FUNCTION_NAME
    annotations:
        services.k8s.aws/region: $AWS_REGION
spec:
    name: $FUNCTION_NAME
    code:
        s3Bucket: $BUCKET_NAME
        s3Key: my-deployment-package.zip
    role: $LAMBDA_ROLE
    runtime: python3.9
    handler: lambda_function.lambda_handler
    description: test lambda function
```

Listing 3.1: AWS Lambda manifest example [12]

3.2.3 Multi-model serving optimization

It is deemed plausible that in the future resource management will rely on an increasing number of machine learning models for decision-making. However, the current design of KServe standard model deployment may not be the most efficient one. As a matter of fact, the "**one model, one server**" paradigm is the one that is currently implemented in KServe. Indeed, even in the context of our system, where **each ElectricityMaps region could be represented by a model**, the current design of KServe would require the deployment of a large number of InferenceServices. Currently, as described in section 2.7.4, our strategy is to deploy specific models for a set of regions and to use a generic model for the remaining regions.

The "one model, one server" paradigm is not the most efficient way to deploy models, as it introduces a lot of overhead and could represent a bottleneck in the system. The Kubernetes cluster could be overloaded with a large number of InferenceServices, with in turn are composed of several Kubernetes Pods with additional sidecar containers. Resource limitations have been calculated by KServe team and described in the official documentation [31]. In particular, since Kubernetes Kubelet (the Kubernetes agent that runs on each node) has a default limit of 110 pods per node, the number of InferenceServices that can be deployed on a single node is limited [31]. In addition to that, Kubernetes has an IP address limit per cluster which could represent an additional bottleneck in some cases.

In recent versions, KServe introduced as alpha feature the so-called **ModelMesh** which aims to solve the limitations briefly described above [31]. In particular, ModelMesh is a component that allows to serve multiple models in a single InferenceService. The study of the ModelMesh feature and its integration with our system is a possible future improvement.

Bibliography

[1] Aether calculation engine. https://aether.green/docs/intro. Last access: 20/12/2024.

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

[3] Aws global infrastructure. https://aws.amazon.com/it/about-aws/global-infrastructure/. Last access: 25/02/2025.

[4] Aws sustainability. https://sustainability.aboutamazon.com/products-services/aws-cloud. Last access: 17/01/2025.

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

[6] Azure network latency. https://raw.githubusercontent.com/MicrosoftDocs/azure-docs/refs/heads/main/articles/networking/azure-network-latency.md. Last access: 10/12/2024.

[7] Azure virtual machines monitoring. https://learn.microsoft.com/en-us/azure/virtual-machines/monitor-vm. Last access: 07/01/2025.

[8] Carbon aware computing at google. https://www.performance2021.deib.polimi.it/www.performance2021.deib.polimi.it/wp-content/uploads/2021/11/Carbon-Aware-Computing-%40-Google-and-Beyond.pdf. Last access: 10/01/2025.

[9] Climate neutral data centre pact. https://www.climateneutraldatacentre.net/. Last access: 17/01/2025.

[10] Cloud carbon footprint. https://www.cloudcarbonfootprint.org/docs/. Last access: 20/12/2024.

[11] Cloudping. https://www.cloudping.co/grid. Last access: 10/01/2025.

[12] Deploying aws lambda functions using aws controllers for kubernetes (ack). https://aws.amazon.com/it/blogs/compute/deploying-aws-lambda-functions-using-aws-controllers-for-kubernetes-ack/. Last access: 27/02/2025.

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

[14] Finops focus specification. https://focus.finops.org/focus-specification/. Last access:15/01/2025.

[15] Gcp config connector. https://cloud.google.com/config-connector/docs/overview. Last access: 12/01/2025.

[16] Google: What is multicloud? https://cloud.google.com/learn/what-is-multicloud?hl=en. Last access: 20/02/2025.

[17] Green software foundation. https://greensoftware.foundation/. Last access: 20/02/2025.

[18] Green software foundation impact framework. https://if.greensoftware.foundation/intro/. Last access: 20/02/2025.

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

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

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

[22] Impact framework cpu to carbon pipeline. https://if.greensoftware.foundation/pipelines/cpu-to-carbon. Last access: 20/02/2025.

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

[24] Kepler documentation. https://sustainable-computing.io/design/architecture/. Last access: 07/01/2025.

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

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

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

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

[29] Kserve documentation. https://kserve.github.io/website/master/. Last access: 27/02/2025.

[30] Kserve mlflow. https://kserve.github.io/website/master/modelserving/v1beta1/mlflow/v2/. Last access: 27/02/2025.

[31] Kserve multi model serving. https://kserve.github.io/website/master/modelserving/mms/multi-model-serving/. Last access: 27/02/2025.

[32] Kserve open inference protocol documentation. https://github.com/kserve/open-inference-protocol. Last access: 27/02/2025.

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

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

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

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

[37] Microsoft azure regions worldwide as of 2024. https://www.statista.com/statistics/1491357/microsoft-azure-availability-zones-global-by-region/. Last access: 25/02/2025.

[38] Microsoft carbon-aware kubernetes. https://devblogs.microsoft.com/sustainable-software/carbon-aware-kubernetes/. Last access: 18/02/2025.

[39] Mlflow documentation. https://mlflow.org/docs/latest/index.html. Last access: 28/01/2025.

[40] Natural earth data dataset. https://www.naturalearthdata.com/downloads/110m-cultural-vectors/. Last access: 27/02/2025.

[41] Nist special publication 800-145. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf. Last access: 27/12/2024.

[42] Opa delta bundles. https://www.openpolicyagent.org/docs/latest/management-bundles/#delta-bundles. Last access: 20/02/2025.

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

[44] Opa external data. https://www.openpolicyagent.org/docs/latest/external-data/. Last access: 15/02/2025.

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

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

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

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

[49] Opa signing. https://www.openpolicyagent.org/docs/latest/management-bundles/#signing. Last access: 20/02/2025.

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

[51] Open policy agent kubernetes primer. https://www.openpolicyagent.org/docs/latest/kubernetes-primer/. Last access: 20/12/2024.

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

[53] 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.

[54] Real-time cloud. https://github.com/Green-Software-Foundation/real-time-cloud. Last access: 20/02/2025.

[55] Scaphandre documentation. https://hubblo-org.github.io/scaphandre-documentation/index.html. Last access: 07/01/2025.

[56] Scaphandre github issue. https://github.com/hubblo-org/scaphandre/issues/142. Last access: 07/01/2025.

[57] Software carbon intensity specification. https://sci.greensoftware.foundation/. Last access: 20/02/2025.

[58] Tag environmental sustainability. https://tag-env-sustainability.cncf.io/. Last access: 17/01/2025.

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

[60] Worldwide market share of leading cloud infrastructure service providers (2024). https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/. Last access: 25/02/2025.

[61] Tayebeh Bahreini, Asser N. Tantawi, and Olivier Tardieu. Caspian: A carbon-aware workload scheduler in multi-cluster kubernetes environments. In *2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2024.

[62] Adrian Cockcroft. Cloud provider sustainability current status and future directions. Last access: 20/02/2025.

[63] Andrés García García, Ignacio Blanquer Espert, and Vicente Hernández García. Sla-driven dynamic cloud resource management. *Future Generation Computer Systems*, 31:1–11, 2014. Special Section: Advances in Computer Supported Collaboration: Systems and Technologies.

[64] Walid A. Hanafy, Qianlin Liang, Noman Bashir, David Irwin, and Prashant Shenoy. Carbonscaler: Leveraging cloud workload elasticity for optimizing carbon-efficiency. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(3), December 2023.

[65] Aled James and Daniel Schien. A low carbon kubernetes scheduler. 07 2019.

[66] Tahseen Khan, Wenhong Tian, Guangyao Zhou, Shashikant Ilager, Mingming Gong, and Rajkumar Buyya. Machine learning (ml)-centric resource management in cloud computing: A review and future directions. *Journal of Network and Computer Applications*, 204:103405, 2022.

[67] Boris Lublinsky, Elise Jennings, and Viktória Spišaková. A kubernetes 'bridge' operator between cloud and external resources, 2022.

[68] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. Deadline-aware dynamic resource management in serverless computing environments. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 483–492, 2021.

[69] Alen Paul, Rishi Manoj, and Udhayakumar S. Amazon web services cloud compliance automation with open policy agent. In *2024 International Conference on Expert Clouds and Applications (ICOECA)*, pages 313–317, 2024.

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

[71] Andreas Schmidt, Gregory Stock, Robin Ohs, Luis Gerhorst, Benedict Herzog, and Timo Hönig. carbond: An operating-system daemon for carbon awareness. In *2nd Workshop on Sustainable Computer Systems (HotCarbon)*, Boston, MA, USA, 7 2023.

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

[73] 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.

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

[75] Shreshth Tuli, Sukhpal Singh Gill, Minxian Xu, Peter Garraghan, Rami Bahsoon, Schahram Dustdar, Rizos Sakellariou, Omer Rana, Rajkumar Buyya, Giuliano Casale, and Nicholas R. Jennings. Hunter: Ai based holistic resource management for sustainable cloud computing. *Journal of Systems and Software*, 184:111124, 2022.

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