SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Carbon-Aware Scheduling for Serverless Computing

Thandayuthapani Subramanian

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Carbon-Aware Scheduling for Serverless Computing

Kohlenstoffbewusste Planung für serverloses Computing

Author:	Thandayuthapani Subramanian
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Mohak Chadha
Submission Date:	15.02.2023

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.02.2023 Thandayuthapani Subramanian

Acknowledgments

I extend my sincerest thanks to Prof. Dr. Michael Gerndt for affording me the opportunity to engage in a stimulating project within a conducive workspace.

I express my profound appreciation and indebtedness to my advisor, Mohak Chadha, for his unwavering mentorship and forbearance throughout the thesis. His invaluable guidance and assistance were instrumental in enabling me to attain the current milestone.

I would like to express my deep appreciation to my mother, Maheswari, my father, Subramanian and my brother, Saravanan for their unwavering love, support, and encouragement throughout my academic journey. Their steadfast belief in my abilities has been a constant source of strength and motivation, and I acknowledge that my achievements would not have been possible without their guidance and sacrifices.

Thank you Amma, for your unwavering support and sacrifices, and for always being there for me no matter what. Your love, care, and guidance have been instrumental in shaping me as a person and as a scholar.

Thank you Appa, for your guidance, wisdom, and unflinching support. Your unwavering belief in me has given me the courage and determination to pursue my academic goals, and I am forever grateful for your sacrifices and dedication.

Thank you Anna, for being my constant source of motivation and inspiration. Your unwavering belief in me has been a driving force behind my academic pursuits, and I am grateful for your support and encouragement.

I want to take this opportunity to extend my sincere appreciation to you, Anand, for the unwavering support and encouragement you have provided me. Your unwavering belief in my abilities and your willingness to lend an ear and offer guidance whenever I have needed it has been an enormous help in guiding me through the challenges of academia. Moreover, your friendly and positive attitude has made the journey more enjoyable and less stressful, and for that, I am incredibly grateful.

Abstract

The combustion of fossil fuels for the production of electricity is a prominent driver of global greenhouse gas emissions, which have far-reaching implications for climate change on a planetary scale. The proliferation of cloud computing in recent years has led to a significant increase in energy consumption and consequent carbon emissions. In an effort to mitigate this, there has been a growing trend towards the adoption of serverless computing as an alternative to traditional infrastructure-based deployments. However, serverless computing is still reliant on energy consumption and thus, contributes to the overall carbon footprint of cloud computing. Geographical differences in the energy mix, which dictate the proportion of electricity generated from fossil fuels versus renewable energy sources, coupled with local disparities in electricity demand, play an instrumental role in the emission levels across different regions. So, it does make a difference in executing a serverless functions in different regions of the world, depending on how clean the electricity is. In this thesis, we propose GreenCourier, an intelligent carbon-aware scheduling approach that enables the delivery of serverless functions in various geographical regions based on the greenness of electricity available at any given moment, to address the issue of carbon emission during function invocation. To achieve this, Knative was chosen as the open-source platform to deploy serverless functions and Kubernetes as the container orchestration platform and Liqo to establish multi-cluster topology of Kubernetes clusters distributed in different geographical regions. The scheduling decision is taken using real-time data obtained from a external source, such as WattTime or Green Software Foundation's carbon-aware Software Development Kit (SDK), which provide information on the carbon intensity of the electricity at a given location. Our proposed approach takes into account the carbon footprint of electricity at different regions at the time of scheduling, with the aim of minimizing the overall carbon emission during function execution by selecting the most carbon-efficient location to run a given workload. To evaluate the effectiveness of our proposed approach, we conduct experiments using realistic serverless workloads in a geographically distributed Kubernetes cluster. The experimental results demonstrate the efficacy of our proposed approach in reducing the carbon footprint of the system without compromising as much as possible on performance. For instance, our approach reduced the carbon emission by 8.7% and 17.8% per function execution in comparison to the default implementation of Kubernetes scheduler and geo-aware scheduler. Additionally, GreenCourier exhibited superior performance over the default and geo-aware scheduling schemes by 36.6% and 63.7%, respectively, in correctly identifying ecologically viable regions and deploying pods in those regions.

Contents

1 Introduction

The escalating energy demands of data centers are projected to consume a considerable fraction of global electricity around 3% to 13% by 2030 [1], exacerbating the greenhouse gas emissions predicament. A recent report [2] from the Intergovernmental Panel on Climate Change [3] underscores that the world is rapidly approaching a perilous 1.5° Celsius rise in temperature, which would trigger disastrous environmental impacts, as it is predicted to have global temperature raise of about 4° Celsius. Consequently, there is an urgent need to devise innovative approaches to boost data center energy efficiency and mitigate its environmental footprint [4, 5]. Serverless Computing is an emerging cloud computing paradigm that promises to separate software from hardware, representing a potential avenue to enhance cloud systems in terms of energy efficiency. To this end, it is essential to evaluate the energy efficiency of serverless cloud offerings, especially as many predict it to be the next big evolution in cloud computing [6].

Prominent cloud providers such as Google Cloud Platform (GCP) have publicly announced their intention to attain complete carbon neutrality by 2030 [7], and have demonstrated tangible progress towards this goal by consistently matching their electricity use with renewable sources since 2017. Likewise, Microsoft has adopted a carbon-neutral stance since 2012, with an ambitious target to become carbon negative by 2030 [8]. Other cloud service providers such as Oracle Cloud have also made significant strides towards achieving carbon neutrality, with their European data centers already functioning on 100% carbon-neutral electricity [9]. Despite these impressive efforts, several cloud providers continue to generate carbon emissions, which are offsetted by their carbon offset programs. This enables them to claim a net-zero operational emissions status, as verified in Google's sustainability report [6].

Cloud computing has been extensively adopted due to its widely accepted and highly scalable nature. The enterprises and organizations are significantly migrating from on-premise to the cloud infrastructure in recent years, owing to the availability of various cloud service offerings such as Infrastructure-as-a-Service (IaaS) that provide a pay-per-use model to their customers. Although these services have made it easier for the customers to rent the resources, the infrastructure management has become more challenging, and the developers tend to over-provision the resources, leading to under-utilization and resource idling. According to a recent study, data centers are wasting 50% of their energy in idle resources [10]. In this context, the emerging paradigm of cloud computing called *Serverless Computing* has the potential to address the issue of wasted energy consumption by idle computing resources.

Function-as-a-Service (FaaS), a key enabler of the serverless computing paradigm offers an attractive cloud model with several advantages such as no infrastructure management, scaling-to-zero for idle resources, a fine-grained pay-per-use billing policy, and rapid automatic scaling on a burst of service requests [11]. As a result, it has gained significant popularity and widespread adoption in various application domains such as machine learning, federated learning, edge computing, and scientific computing [12, 13, 14]. In FaaS, applications are developed as small pieces of code called functions that are executed in response to event triggers such as HTTP requests. According to recent estimates, the market for serverless computing is going to increase from USD 13.47 Billion in 2021 to USD 86.94 Billion by 2030 [15]. All major cloud service providers such as Google, Amazon, and Microsoft have their own commercial FaaS offerings such as Google Cloud Functions[16],Amazon Web Services (AWS) Lambda [17], and Azure Functions [18]. Typically, these cloud service providers receive around 1.1 billion function invocations each day [11]. However, none of the current commercial FaaS offerings consider the carbon-efficiency of the different geographical regions for executing the functions.

In general the characteristics of serverless functions being *temporal* i.e, short execution time, so moving them to a different region for execution will not impact their performance much. In this work we strive to achieve a novel approach to sustainable serverless computing, in the form of GreenCourier, an intelligent scheduling policy that enables the delivery of serverless functions in various geographical regions based on the renewable resources available at any given moment. GreenCourier leverages Kubernetes[19], the leading container orchestration platform in the cloud, and Knative [20], a corporate-grade platform for constructing serverless applications. And as part of this thesis, we evaluate implementation of GreenCourier against performance of default scheduler implementation in Kubernetes and geo-aware scheduling scheme which prioritizes scheduling of functions to a region which is the closest.

This thesis introduces the problem of escalating energy demands of data centers, which are projected to consume a considerable fraction of global electricity, exacerbating the greenhouse gas emissions predicament. It proposes that Serverless Computing, an emerging cloud computing paradigm, represents a potential avenue to enhance cloud systems in terms of energy efficiency. The thesis then introduces the concept of sustainability in Function-as-a-Service (FaaS) and its benefits, as well as the potential of GreenCourier, an intelligent scheduling policy that enables the delivery of serverless functions in various geographical regions based on the greenness of electricity available at any given moment, to address the issue of carbon emission during function invocation. GreenCourier aims to reduce carbon emissions without compromising on performance. Finally, the thesis outlines the research objective of evaluating the implementation of GreenCourier against the performance of default scheduler implementation in Kubernetes and a geo-aware scheduling scheme.

1.1 Problem Definition

The problem that GreenCourier aims to address is the lack of consideration for carbon efficiency in current commercial serverless computing offerings. Although serverless computing provides a highly scalable and flexible model for executing short-lived tasks, its carbon footprint has not been optimized due to the lack of intelligent scheduling policies that leverage renewable energy sources. As a result, the energy consumption of data centers continues to rise, exacerbating the global greenhouse gas emissions predicament. GreenCourier proposes an intelligent scheduling policy that delivers serverless functions in various geographical regions based on the availability of renewable resources to minimize carbon emissions while ensuring optimal performance.

1.2 Research Objectives

Based on the above mentioned goals, this study aims to answer the following three research questions (RQs) by evaluating and implementing scheduling policies for serverless workloads:

- **Research Question 1:** *When and what kind of serverless workloads should be migrated?*
- **Research Question 2:** *How can Knative be extended to support carbon-aware scheduling of serverless functions?*
- **Research Question 3:** *What kind of scheduling policy should be adopted to achieve the goal?*

By answering these three research question, aim of this work is to deliver key outcomes as such:

- Implementation of a scheduling framework which can identify and schedule serverless functions in a environmentally-sustainable region, to minimise carbon emission associated with function execution.
- Implement metrics server which calculates carbon score for different regions which should have capability of supporting different sources of Marginal Operating Emissions Rate (MOER).
- Evaluate system's performance against default scheduler implementation and geo-aware scheduling scheme and benchmark results in terms of carbon emission and response time.

Our experiments show that there is significant reduction in carbon emission associated with function execution by intelligently scheduling functions to region of "low-carbon" electricity when compared with default and geo-aware scheduling schemes.

1.2.1 RQ 1: Migrating Serverless Workloads

In general, we can safely segregate types of workloads which are commonly deployed as serverless workloads into four [21]: 1) Micro Jobs, 2) Application, 3) Machine Learning (ML) Model - Training, 4) ML Model - Serving. Micro Jobs can be relatively short running workloads in comparison with other workloads. Typically, this kind of workload involve having arithmetic operations, matrix operations and solving equations. It can be safely assumed that micro jobs take either some values or a JavaScript Object Notation (JSON) as input and either JSON or file as an output. In case of application, the workload can vary from processing a transaction and storing the state in some database to image and video processing application. So the execution time of such workloads will vary depending on the use case. Though the names look similar for next two workloads, they both have different characteristics. In ML model - Training workload, we train actual ML model, which is very much a data intensive and long running workload. In ML model - Serving workload, we deploy already trained model and use it for inference. This type of workload can also be long running one, but not as data intensive as training a ML model.

Our main goal in this research is to make carbon-aware scheduling decision. Transferring huge amount of data through network as input for serverless workload or output from serverless workload will consume energy and directly contradicts our main goal, which is to reduce the carbon foot print for our workload execution. So it is sensible to migrate serverless workloads which are not data intensive and transferring input and output through network will not cost much in terms of energy. In conclusion, it is logical to migrate following serverless workload types: Micro jobs, Application which are not data intensive and ML model - Serving.

1.2.2 RQ 2: Extending Knative

In this research, Knative will be used to create serverless workloads. Knative is an open source solution to build and manage serverless workloads in kubernetes clusters. In short, the functionality involved in Knative can be explained as follows: It is required as an user to create Knative service [22] to deploy their application. Under the hood, Knative create a ReplicaSet [23] in Kubernetes. ReplicaSet is a Kubernetes object which ensures there is stable set of running pods for a specific workload. Number of pods should be defined in ReplicaSet configuration. Those pods are scheduled using scheduler process in Kubernetes. As mentioned above, it is possible to extend kubernetes schedulers [24, 25]. Real-time data carbon intensity data is required for making scheduling decisions. It is possible to access those data from services like WattTime [26, 27] and electricityMap [28, 29]. Other alternatives like simulated dataset [30] is also available. Theoretically, it is possible to extend Knative with Kubernetes with the above mentioned resources to make carbon-aware scheduling decisions.

1.2.3 RQ 3: Scheduling Policy

Scheduling policy is set of directives and purpose which directs the scheduler process to make optimal decision. In Kubernetes scheduler, selection of node is done in a 2-step operation [31]: 1) Filtering using Predicate scheduling policies, 2) Scoring using Priorities scheduling policies. Filtering step finds suitable set of nodes which is feasible to schedule the workload. In filtering stage, scheduler runs workload's requirement and list of nodes in cluster through predefined set of predicate policies like CheckNodeConditionPredicate, CheckNodeUnschedulablePredicate, etc., and obtains list of nodes which satisfies workload's requirement. After filtering step, in scoring stage filtered set of nodes and workload's requirement is given as input for definite set of priority scheduling policies like ImageLocality, PodTopologySpread, etc. After scoring stage, we get score assigned to each node and scheduler chooses the node which has the highest score. In this research, a new scheduling policy in scoring stage would help accomplishing the expected behaviour. Aim of this research is to implement a carbon-aware scheduling policy, which prioritizes to place the workload in a node in a region which uses energy with less carbon footprint.

1.3 Thesis Overview

This thesis follows a structured approach and consists of multiple chapters. In Chapter 2, we examine the fundamental concepts of serverless computing, container orchestration, and Kubernetes, with a specific focus on the scheduler's core concepts and its extensibility through the Scheduling framework. We also delve into different open-source serverless platforms' architectures and their adoption by the open-source community. Furthermore, the chapter explores various projects that allow users to establish a multi-cluster topology in Kubernetes, with an in-depth analysis of Liqo and its components. Chapter 3 is dedicated to exploring different scheduling solutions in the realm of serverless computing and sustainable scheduling in data centers, along with other sustainable scheduling approaches on top of Kubernetes. In Chapter 4, we provide an in-depth discussion of the proposed solution's system architecture and implementation, with a focus on GreenCourier. Additionally, we discuss the challenges encountered during development and the solutions implemented to overcome them. Chapter 5 defines the metrics used to evaluate various scheduling schemes and presents an analysis of the results. Furthermore, the chapter delves into experiments that were conducted to determine the response time of functions and analyzed the pod placement characteristics in relation to carbon emission numbers. Finally, the last chapter provides a summary of the work and discusses the system's limitations while suggesting future steps to enhance GreenCourier.

2 Background

2.1 Serverless Computing

Serverless is the fastest-growing and the most preferred cloud service model with annual increase of 75% increase in adoption [32]. Serverless computing offers a platform in cloud where developers simply have to upload their code, and the platform has capabilities to execute on their behalf as needed at any scale. Provisioning resources is responsibility of the platform. Developers only have to pay for the resources only when the code is invoked [33]. Serverless platforms promise new capabilities like event processing, API composition and data flow control that make writing scalable microservices easier and effective. All the major cloud providers have their own solution for serverless computing, like Amazon Web Services (AWS) Lambda [17], Google Cloud Platform (GCP)'s Cloud functions [16] and Azure functions [18]. And many open source solutions like OpenWhisk [34] from Apache Software Foundation [35] and Knative [36] from Cloud Native Computing Foundation [37].

The main benefits of using a proprietary solution for serverless computing are the reliable underlying infrastructure and the ability to easily integrate with the provider's ecosystem (e.g. using an S3 bucket object to trigger a function on AWS Lambda). The significant drawback of utilizing proprietary solutions lies in the dependence on the vendor for both the framework updates and underlying infrastructure, which imposes limitations on developer agility and may result in vendor lock-in. In contrast, the implementation of open-source solutions confers greater control, transparency, and customization potential, leveraging the contributions of a vast developer community and offering support. Open-source platforms do offer more control over the platform's infrastructure and adopt more suitable deployment schemes, either to enhance resource usage, to reduce latency or to enhance data locality. It also enables more efficient testing and benchmarking, which can help identify and address performance issues, leading to overall system efficiency. However, it demands a higher investment in terms of maintenance, deployment, and security, and may entail a more challenging learning curve for neophytes. So in scope of this thesis, we use Knative as the serverless platform for deploying the functions and the architecture of the Knative serving is shown 2.1. Knative [20] as the open-source framework for our research, because major public cloud offers services to deploy Knative application such as in Google Cloud Platform [38], IBM Cloud [39] and also because of it's interoperability with Amazon Web Services [40] using TriggerMesh [41]. By this way, issue of vendor-lock in is avoided and true potential of public cloud is also utilized.

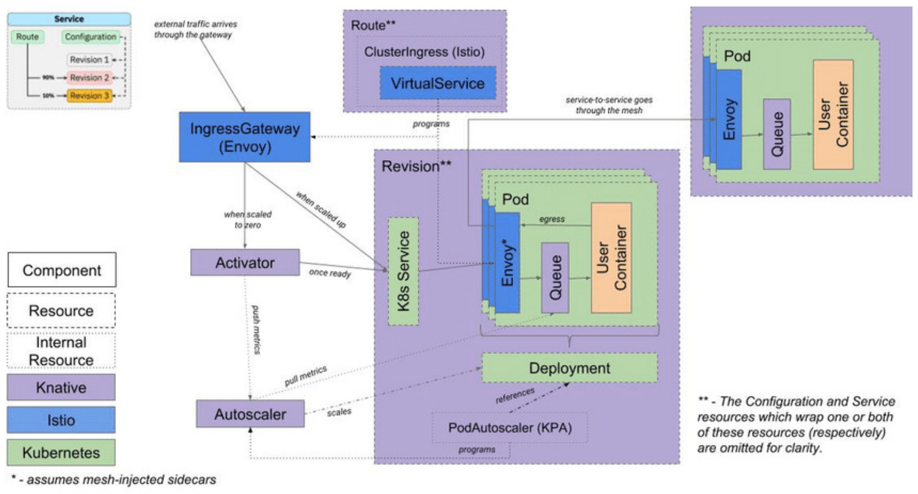

Figure 2.1: Knative Serving Architecture [42]

2.1.1 Characteristics

Applications conforming to the tenets of serverless computing and specifically, FaaS solutions, typically exhibit a set of common attributes, as evidenced by the findings presented in [43]. An exhaustive investigation of the runtime characteristics of serverless applications in this domain revealed that the vast majority of such applications display a runtime that is negligible in magnitude, with a typical order of seconds or milliseconds. In contrast, only a minuscule fraction of serverless applications exhibit a runtime that surpasses a duration of minutes. The study further revealed that serverless applications frequently exhibit a workload pattern characterized by "burstiness," i.e., the utilization of Function-as-a-Service (FaaS)-based applications often exhibits sporadic spikes in resource consumption, as a result of their predominantly HTTP-based or cloud-trigger-initiated invocations, such as those arising from file uploads to cloud-based storage. In FaaS-based serverless applications, small and specific functions are used to perform particular tasks efficiently. The scalability of such functions is generally easier and cost-effective as it necessitates scaling only the specific components of the application, as opposed to a less specialized function that might result in the unnecessary scaling of multiple parts of the application. Using functions with fewer libraries results in faster load times due to the smaller size of the function code archive and quicker library initiation, reducing the cold start problem as explained in 2.1.2. Hence, it is considered a general guideline to minimize the number of libraries utilized in functions within FaaS-based applications.

2.1.2 Advantages and Issues

As previously elucidated, serverless computing enables developers to concentrate solely on the implementation of the application, absolving them of any obligation for the administration of the underlying infrastructure. This advantage, in tandem with the various language runtimes commonly accessible in serverless platforms, facilitates the optimization of the development and deployment process. The serverless architecture allows the use of various programming languages and libraries without needing a deep understanding of the infrastructure.

In comparison to Infrastructure-as-a-Service (IaaS) solutions, serverless computing is generally considered to be cost-efficient due to the pay-per-use model, in which customers are only charged for function execution. This is particularly beneficial for scenarios with dynamic or ephemeral workloads, which involve alternate short durations of computation and extended periods of inactivity. In an IaaS approach, the customer would need to incur expenses for provisioned resources until their explicit decommissioning, whereas in FaaS, only the time of execution incurs a charge. Despite these advantages, various limitations and challenges exist within the domains of serverless computing and FaaS. A comprehensive evaluation of these existing behaviors and constraints is presented by the authors of [44] such as:

- The limitations of FaaS solutions in terms of resource allocation and execution time often hinder their suitability for computationally intensive applications. The FaaS abstraction typically only provides a single function without differentiating between function types, limiting the resources available to the function's runtime. Additionally, if platforms impose maximum execution time restrictions on functions, errors can occur for prolonged tasks. This is particularly relevant in the context of deep learning applications, which often require longer execution times and precise control over the hardware executing the function (e.g. use of Graphics Processor Units (GPUs)).
- Users lack control over the platform's elasticity controller, causing standard provisioning and de-provisioning times of FaaS solutions to be unsuitable for the application's scaling requirements. This may lead to slow response to spikes in traffic or reduced performance due to over-eager de-provisioning. This becomes a bigger issue when combined with the limited elasticity of many services, potentially failing to meet the demands of serverless applications [45].
- While FaaS is generally cost-effective, not accounting for idle time, other factors can still increase the cost of execution. Developers must consider the cost-performance trade-off by specifying the memory size of functions, which might not be sufficient or exceed the actual needs. Additionally, the cost of function composition, such as synchronous calls between functions being billed as execution time, should also be considered.

In their work "Serverless Computing: One Step Forward, Two Steps Back" by Hellerstein et al [46] also highlight similar issues, including the lack of access to specialized hardware and limited function lifetime in the AWS Lambda platform. Vendor lock-in is a concern in commercial serverless solutions. While some serverless applications can be deployed on multiple cloud providers or using open-source frameworks, proprietary solutions are still widely used (80% on AWS according to [45]). Migration to other providers can also be difficult if the application interacts with specific workflows or services, and vendor lock-in with serverless computing may be stronger than with traditional solutions. Deploying an open-source solution on a private cluster also requires cluster management, contradicting the ease of use appeal of the serverless paradigm, leading to reliance on a single cloud provider and vendor lock-in. Handling cold starts, or the time taken to initialize the environment, is a widely discussed issue in FaaS [47, 48]. Cold start latency, especially with functions packaged with many libraries, can lead to delays in code execution and make serverless approaches unsuitable for applications with fast response times. However, applications requiring a stable latency (e.g. human-user interactions) still exist [43], despite cold starts.

2.1.3 Use Cases

According to IBM[49], the main use cases for serverless architecture are microservices, Application Programming Interface (API) backends, data and stream processing, and parallel tasks. Microservices benefit from serverless computing's automatic scaling and quick provisioning/deprovisioning of resources. API backends can be created by combining functions triggered by external events like Hypertext Transfer Protocol (HTTP) requests. Data and stream processing benefits from the simplicity and scalability of serverless solutions and can be more cost-effective compared to alternatives like Amazon Elastic MapReduce (Amazon EMR). In [46], three categories of applications are listed: embarrassingly parallel functions, orchestration functions, and function composition. The first category involves simple, independent tasks and has limited scope and complexity. The second category involves functions that orchestrate calls to other services (e.g. preprocessing data for analytics). The last category involves collections of functions combined to build applications and pass along inputs and outputs (e.g. groups of functions triggered by events on storage services). Serverless solutions are widely applicable and are used for various application domains such as machine learning [50, 51, 52, 53], edge computing [54, 55], high performance computing [56, 57], and heterogeneous computing [58, 59].

2.2 Container Orchestration and Serverless Frameworks

Kubernetes was developed and open sourced by Google in 2015. It was initially used internally in Google for running and maintaining their containers [60]. Kubernetes was then donated to Cloud Native Computing Foundation, which is part of Linux Foundation. Default scheduling algorithm used in Kubernetes can be understood from kubernetes official documentation [61]. Excerpt from the official documentation as follows: There are two steps before a destination node of a Pod is chosen. The first step is filtering all the nodes and the second is ranking the remaining nodes to find a best fit for the Pod. Scheduler first evaluates all the nodes in the cluster based on number of rules called predicates, which filters out unqualified nodes. Next, all the qualified nodes goes through another scheduling rules called priorities, which ranks the nodes according to preferences. In general, a scheduling policy is combination of predicates and priorities. List of all supported scheduling policies of predicates and priorities are explained in official Kubernetes documentation [61].

2.2.1 Kubernetes Architecture

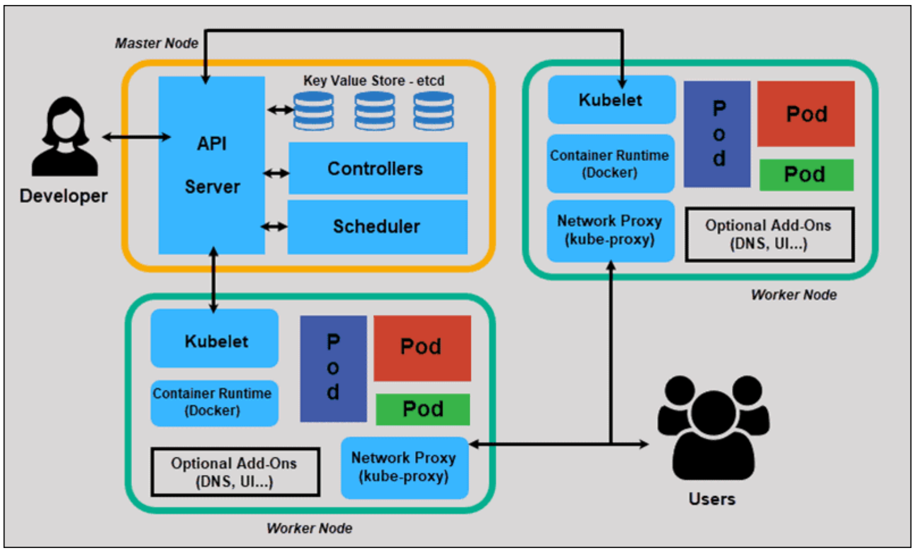
Master Node
Key Value Store - etcd
API
Server
Controllers
Scheduler
Developer
Kubelet
Container Runtime (Docker)
Network Proxy (kube-proxy)
Optional Add-Ons (DNS, UI...)
P o d
Pod
Pod
Worker Node
Kubelet
Container Runtime (Docker)
P o d
Pod
Pod
Optional Add-Ons (DNS, UI...)
Network Proxy (kube-proxy)
Worker Node
Users

Figure 2.2: Kubernetes Architecture [62]

Kubernetes is also no different from other Cloud Native applications, which use the Twelve Factor App methodology [63]. Different components of Kubernetes can be seen in 2.2. And each components are explained in detail below.

2.2.1.1 Control Plane Components

The orchestration layer within the control plane is tasked with executing critical, high-level determinations regarding the overall cluster operations, including but not limited to the allocation of resources for tasks and processes. Additionally, it executes event-driven, real-time monitoring and response functions with respect to changes and fluctuations within the cluster, including scaling events involving the incorporation or detachment of nodes. The control plane is charged with preserving the predefined cluster configuration, and it engages in continuous, bidirectional communication with worker nodes to guarantee compliance with their assigned responsibilities. Control plane components are as follows:

- **API Server:** API server acts as the front-end for other control plane components. Every control plane component should communicate with each other through API server. API server can be scaled horizontally to ensure high-availability in the cluster even when one of the instances fail. API Server is responsible for authentication and authorization of the requests from user to control plane.
- **Controller Manager:** Controller Manager are simply just a control loops which ensure the system's current state to desired state. There are many controllers like Node Controller, Job Controller, EndPoint Controller, ServiceAccount Controller. The implementation of a Replication Controller in a Kubernetes cluster facilitates the management

of pod replicas by monitoring and adjusting the number of running pods to align with the specified configuration. The availability of nodes is monitored via the Node Controller, while the Endpoint Controller performs the function of aggregating pods into services. Furthermore, the Service Account Controller is responsible for generating API access tokens and creating default accounts for newly created namespaces, thereby contributing to the maintenance of the desired state within the cluster.

- **ETCD:** ETCD is the default data-store in Kubernetes. It is the only component which is stateful and other control plane componenets are stateless and store their state in ETCD. ETCD is a key-value store and serves request from API server. ETCD can also be deployed in highly available setup, which uses RAFT consensus protocol to maintain consistency. Initially the development was driven by CoreOS, but currently it is adopted as a open source project.
- **Cloud Controller Manager:** The Cloud Controller Manager, similar to the Controller Manager, implements a control loop to execute cloud-specific operations within the Kubernetes cluster environment. Its introduction in version 1.6 of the platform facilitated the independent lifecycle management of cloud components, enabling leading cloud service providers, such as Amazon, Google, Microsoft, Oracle, etc., to develop their own unique Cloud Controller Manager implementations. This abstraction layer integrates cloud-specific features, such as load balancing and storage, into the Kubernetes API, thereby facilitating the management of cloud resources required by the cluster. The separation of cloud-specific code enables the Kubernetes core to maintain its vendor-agnostic and flexible nature, enabling users to employ a uniform management layer across multiple cloud providers or on-premise infrastructure.
- **Scheduler:** Scheduler picks up the pod objects from ETCD through API server and checks for pods which are not assigned with nodes and assigns a node for the pod to run by evaluating several predicate and priority schemes, which are selected as part of scheduler configuration. Scheduler can preempt or remove pods from the cluster if it jeopardizes the cluster state.
- **Kube-Proxy:** Kube-proxy acts as the network proxy running on every node of the cluster enabling the capability of connecting multiple pods through service objects. It creates and maintains network rules across cluster. It is capable of forwarding every User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) traffic and does simple loadbalancing within available workloads in round-robin fashion.
- **Kubelet:** Kubelet is a daemon agent running on every worker node, responsible for ensuring the desired state mandated by control plane. Also establishes status sync loopback with API server about the node status and status of workloads running that node. Kubelet is the connection between Kubernetes and container runtime in a particular node.

2.2.1.2 Kubernetes Objects

Kubernetes objects serve as persistent entities within the Kubernetes ecosystem, representing the current state of the cluster. These objects, such as Pods, Services, Replication Controllers, and Deployments, each symbolize a particular facet of the cluster, including a running process, network service, desired replication state, or a specified application deployment. The Kubernetes API is utilized for the creation, modification, and deletion of these objects, with the state of the cluster monitored and maintained by the control plane components, including the API server, Controller Manager, and ETCD, which continuously assess the state of the objects and take appropriate actions to ensure adherence to the desired state. The Kubernetes object model plays a crucial role in the definition, deployment, and administration of applications on the cluster, further incorporating configuration information such as resource constraints, network policies, and access controls, as well as the capability for annotation with supplementary metadata to provide additional context and information. The object model's ease and efficiency in managing complex and large-scale applications make it a vital component of the Kubernetes platform. In this section, we pick two of Kubernetes objects - Pods, Services and discuss in detail.

- **Pods:** Pods are the smallest unit in Kubernetes which can be deployed. A pod can have one or more containers. Containers in a single pod share same network and storage namespaces, i.e., uses same IP address and shares a mounted data across each other. Communication between two containers in a pod is done through localhost and inter-pod communication is done through unique IP assigned when a pod is created.

- **Service:** The Service abstraction in Kubernetes represents a persistent network endpoint for accessing a set of Pods in a manner that is insulated from the underlying network infrastructure. It serves as a crucial component in the Kubernetes networking model and provides a stable and consistent mechanism for accessing applications hosted on Pods. By utilizing Services, application developers are relieved from the requirement of modifying their code to adapt to unfamiliar service discovery methodologies. Furthermore, Services are equipped with the ability to assign unique IP addresses and single Domain Name Service (DNS) names to a group of Pods, and can effectively balance network traffic amongst them to achieve high levels of application availability and resiliency. The utilization of Services in a Kubernetes cluster enables decoupled deployment and management of applications, allowing developers to focus solely on the development of the application, rather than worrying about the complexities of the underlying network infrastructure. Additionally, Services provide a unified and streamlined access mechanism for the deployed applications, making it easier to scale and manage applications in a production environment. The Service abstraction layer serves as an integral part of the Kubernetes infrastructure and enables efficient application deployment, management, and scaling in a robust and scalable manner.

- **Event:** Event in general is used for denoting some change in the system usually generated by some components in the cluster. It is usually used for debugging clusters by examining state of the particular component or some objects in the cluster.

2.2.2 Scheduling in Kubernetes

In the context of Kubernetes cluster orchestration, the process of determining the optimal node for running a pod involves two consecutive stages - filtering and scoring. The filtering phase is executed using a set of predicates, which act as stringent limitations that the selected node must satisfy to be deemed eligible for hosting the pod. In contrast, the scoring phase evaluates the node based on a series of soft constraints, referred to as priorities. These priorities may either be specified in the pod definition or could stem from broader constraints associated with the nodes or the cluster. During filtering, the scheduler eliminates nodes that fail to meet the required predicates and the final selection is based on the outcome of the scoring phase. Example of filtering plugins is *NodeResourcesFit*, this plugin checks whether the resource requested by the pod is available in a particular node, i.e., whether a node satisfies resource contraints set by pod or not. Another example is *NodeUnschedulable* plugin, which checks whether a node is marked as unschedulable because of various reasons, such as upcoming maintenance or node failure. Once all the predicate plugin is done evaluating all the nodes in a cluster, filtered set of nodes are sent to priority phase as shown in Figure 2.3. In priority phase, every node is evaluated by enabled priority plugins, which assigns every node with a score. Example of a priority plugin is *ImageLocality*, which scores a node with high score if the container image requested by pod is locally present in that node. Similarly node, runs through all the enabled priority plugins and is ranked depending on node's score. Node with highest score is selected and pod is assigned to that node. Few of the important plugins are listed below [31]:

- **NodeResourcesFit:** Checks for resource available in particular node against pod's resource request and retains or removes the node from scheduling cycle depending on resource availability.
- **NodeUnschedulable:** Removes nodes with *.spec.unschedulable* field set to true from scheduling cycle.
- **ImageLocality:** The scheduling algorithm prioritizes nodes that already have the required container images for the specified Pod, thereby reducing the time and resources required for image transfer and improving the overall efficiency of the scheduling process.
- **TaintToleration:** Implements taints and toleration logic. In Kubernetes, set of nodes can be assigned with a taint, which forces scheduler to avoid scheduling in that node. But when a toleration is added as part of podSpec, then scheduler tries to ignore the taint associated with a node and adds a possibility of deploying pod to the tainted node as well.
- **NodeName**: This is a filter algorithm which checks for assigned node name already, if any. If so, the pod is forced to be deployed in that particular node, taking other nodes out of contest.

- **NodePorts**: In a multi-tenant computing environment, the presence of heterogeneous workloads originating from various sources can raise the probability of port conflicts. To mitigate this issue, the scheduling mechanism enforces the placement of newly created pods with conflicting port requirements on separate nodes, thus avoiding any port collisions and ensuring the seamless operation of all Pods. This approach safeguards the cluster's ability to operate optimally and maintain its high level of resource utilization.
- **NodeAffinity:** The heterogeneous nature of nodes in a cluster may sometimes result in workloads having specific requirements for node specifications. To address this issue, the algorithm implements a mechanism for identifying suitable nodes based on these specifications and assigning the relevant pods to these nodes, thereby ensuring optimal workload placement and resource utilization.
- **PodTopologySpread:** Regulation of Pod distribution across various topological domains including regions, zones, nodes, and user-defined domains, in accordance with failure-domain considerations. This results in a balance between high availability and resource efficiency.

2.2.3 Scheduling Framework in Kubernetes

Scheduling framework[65] is a feature provided by Kubernetes community from kubernetes-v1.19 release. It adds a new set of "plugin APIs" into scheduler code, enabling developers to create custom logic as required for a business use case. Because of this pluggable architecture, the necessity of developing and maintaining core scheduler logic has become effortless, thus decoupling the dependency on maintaining core modules of scheduler. Scheduler framework defines various extension points exposed by scheduler API, where plugins can be plugged and executed during scheduling. Scheduling will happen in two phases - scheduling cycle and binding cycle. Each phase has different number of extension points, each invoked at different point of time during scheduling and providing unique functionality. Different extension points can be seen in Figure 2.4 There are 10 unique extension points, each serve own purpose. One can implement the exposed interface in scheduler code and compile the plugin code into scheduler code and can be used for deployment. Those 10 extension points and it's functionality are as explained:

- **QueueSort:** Queue sort plugins serves to prioritize the sequence of Pod scheduling within a queue. The queue sort plugin operates by implementing a comparison function, "Less", that assesses the relative order of two Pods in the queue. It is important to note that, in order to maintain consistency, only a singular queue sort plugin can be enabled concurrently.
- **PreFilter:** PreFilter plugins serves to provide pre-processing of information related to a Pod or to validate specified conditions that must be satisfied by the cluster or the Pod. If a PreFilter plugin returns an error during its execution, the scheduling cycle will be terminated.

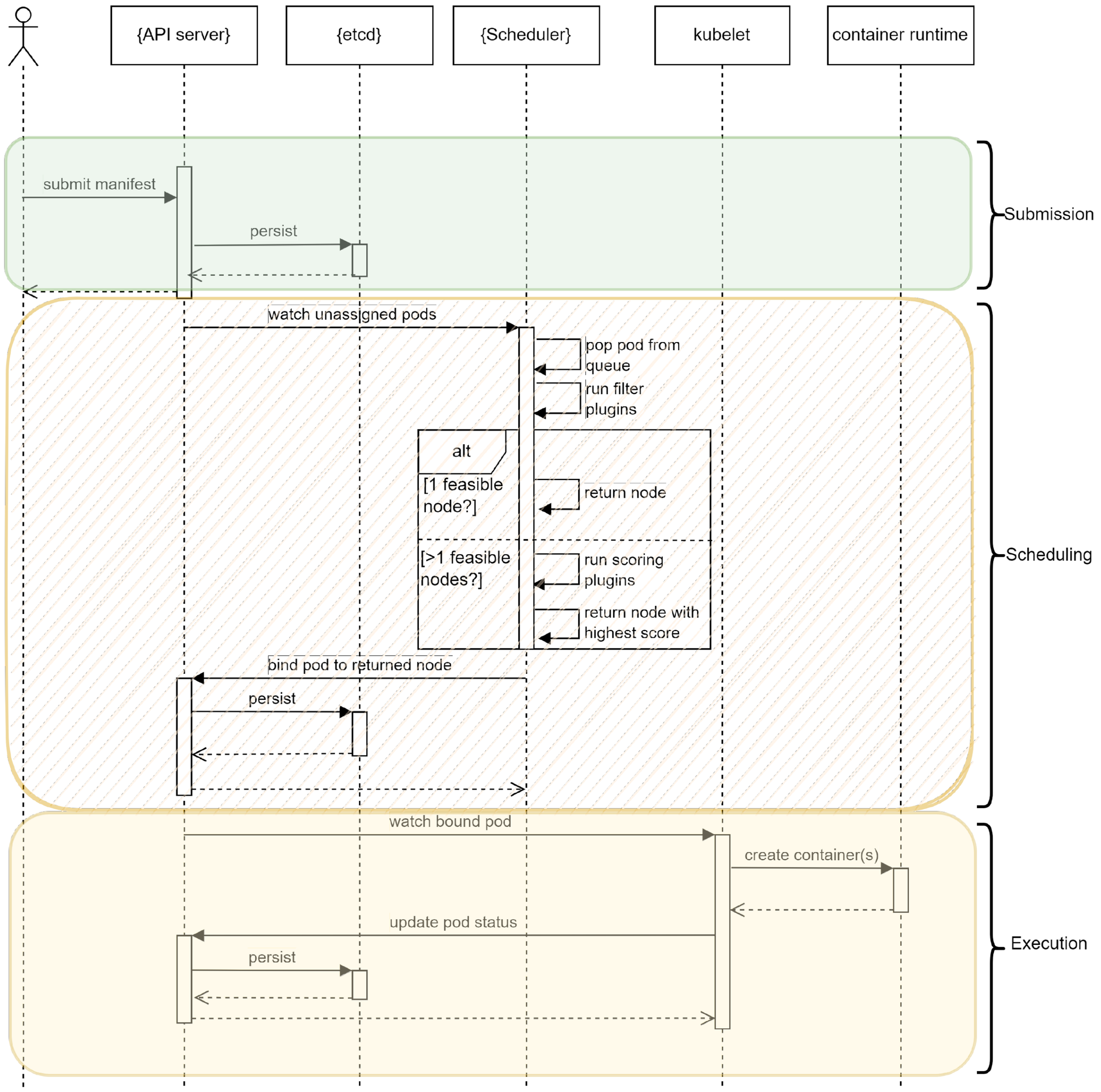

Figure 2.3: Pod lifecycle in Kubernetes [64]

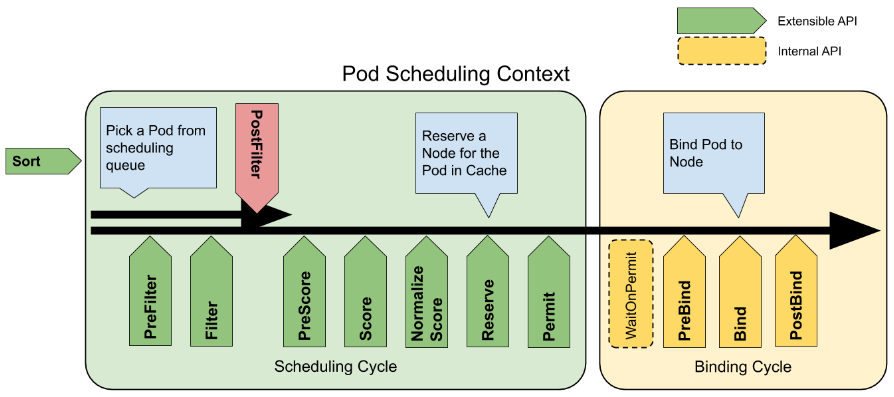

Figure 2.4: Kubernetes Architecture [65]

- **Filter:** Filter plugins are employed to eliminate nodes that are unable to run a particular Pod. The scheduler will sequentially call the filter plugins for each node, in accordance with their configuration order. If any of the filter plugins marks a node as infeasible, subsequent filter plugins will not be invoked for that node. Evaluation of nodes may occur concurrently.

- **PostFilter:** The PostFilter phase is invoked when no feasible nodes have been found for a Pod during the Filter phase. The plugins are called in their configured order and if any PostFilter plugin designates a node as Schedulable, the subsequent plugins will not be executed. An example implementation of PostFilter is preemption, which aims to make a Pod schedulable by preempting other Pods.

- **PreScore:** PreScore plugins are utilized to perform pre-scoring operations, which create a shared state for the Score plugins. If a PreScore plugin returns an error during its execution, the scheduling cycle will be terminated.

- **Score:** Score plugins are leveraged to hierarchically categorize nodes that have been deemed eligible via the filtering phase. The scheduler will iteratively invoke each scoring plugin for each node and the scores generated will be represented by an integer range that denotes the lower and upper bounds. Upon completion of the NormalizeScore phase, the scheduler will amalgamate the node scores generated by all plugins, utilizing the specified weightage configuration of each plugin.

- **NormaliseScore:** After score phase, this phase is invoked. This plugin is used to modify the score, before scheduler makes the decision of assigning a node to a pod. This phase is executed once every scheduling cycle.

- **Reserve:** Reserve phase is executed before scheduler starts its binding cycle. It is implemented through a plugin that encompasses two operative methods, Reserve and Unreserve. These methods are associated with the Reserve and Unreserve informational scheduling phases, respectively. Stateful plugins, which maintain runtime state information, should leverage these phases to be apprised of the scheduling process with regards to the reservation and de-reservation of resources on a node for a designated Pod.

- **Permit:** In this phase, scheduler takes either one of three decisions - approve, deny, Wait. Once all the plugins in permit phase approves the pod, then the pod is taken for binding cycle. If any of the plugins rejects the pod object, the pod is sent back to scheduling queue and in next cycle, scheduler tries to assign node for that pod. If any of the permit plugins return "wait" signal, then the pods are kept in waiting state until timeout or until any other signal to approve or deny is sent.
- **PreBind:** From this phase onwards, the binding cycle of the scheduler starts. These plugins perform any actions which are required before a pod is bound. For example, if the pod needs a network volume, prebind plugin creates one and mounts it on the target node, before pod is deployed. If any of the prebind plugins returns error, then the pod is again added scheduling queue.
- **Bind:** These plugins are the actual ones which work on binding the pod to assigned node. If any of the bind plugins chooses to handle the pod, then the other bind plugins are skipped.
- **PostBind:** This phase does not do anything important in terms of running the pod in assigned node. It is used for any clean up activity after the scheduling is done.

2.2.4 Serverless Frameworks based on Kubernetes

The open-source platforms affords developers the flexibility to seamlessly integrate multiple services into their applications, thereby augmenting the robustness of the application and reducing development time. The open-source nature of such software and frameworks is further reinforced by a large and thriving community of developers who contribute to their ongoing development and maintenance. While open-source platforms offer numerous advantages, but there are certain drawbacks too and that must be taken into consideration. One such limitation is the learning curve associated with utilizing open-source software, which can require a certain level of technical competency and proficiency to effectively utilize and integrate the various components within an application. Despite this challenge, the benefits of open-source platforms often outweigh the disadvantages, making it a popular choice among developers for building robust, scalable and flexible applications.

2.2.4.1 OpenFaaS

OpenFaaS [66] is an open-source Serverless framework for building and deploying cloud-agnostic Functions as a Service (FaaS). The architecture of OpenFaaS consists of three main components: API gateway, function watchdogs, and user-defined functions. How those components interact with each other can be seen in Figure 2.5. The API gateway serves as the entry point for incoming requests and manages the routing of requests to the appropriate function. The API gateway is implemented using a reverse proxy, such as NGINX, and is responsible for handling request authentication, monitoring and scaling of functions, and providing an API endpoint for management operations. The function watchdogs are responsible for managing the lifecycle of user-defined functions. The watchdog is a lightweight process that monitors the function's health, restarts it in case of failure, and communicates with the API gateway to update the function's status. The watchdog is a critical component in ensuring high availability and resilience of the OpenFaaS deployment. The user-defined functions are the main building blocks of the OpenFaaS architecture and are the units of computation that perform the desired business logic. The functions are packaged with a minimal runtime and executed in isolated environments, which enables high scalability and reduces the attack surface of the overall deployment. In conclusion, the OpenFaaS architecture provides a robust, scalable, and secure Serverless platform for building and deploying cloud-agnostic FaaS, leveraging containerization, reverse proxying, and lightweight process management to achieve its goals.

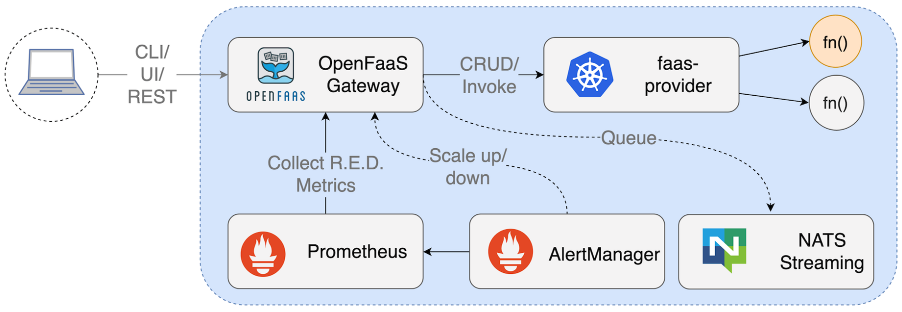

Figure 2.5: OpenFaaS Architecture [67]

2.2.4.2 Openwhisk

The architecture of Apache OpenWhisk [34] comprises a distributed, event-driven compute platform that is built to run serverless functions. It is based on a microservices architecture that includes components such as the controller, invoker, and database services. The controller is responsible for managing and coordinating the platform, receiving events and triggering actions, and enforcing access control policies. The invoker is responsible for executing the functions and providing feedback on their execution. The database services provide persistence for platform metadata and functions. Architecture of OpenWhisk is pictorially depicted in Figure 2.6. OpenWhisk leverages a highly scalable, publish-subscribe model that enables the system to quickly process high-volume events and concurrently execute a large number of actions. The platform can be extended with additional services, such as external databases, object stores, and message queues, to provide additional functionality for building and deploying complex, multi-tier applications. OpenWhisk's architecture encompasses a polyglot, loosely-coupled system that employs a choreographed collection of microservices to dynamically allocate compute resources and execute functions in response to events. The system leverages advanced concepts such as auto-scaling, self-healing, and elastic resource allocation to provide a highly available and scalable platform for executing serverless functions. With its emphasis on event-driven computing, OpenWhisk's architecture is well-suited for building applications that require fast, real-time processing of events and data streams.

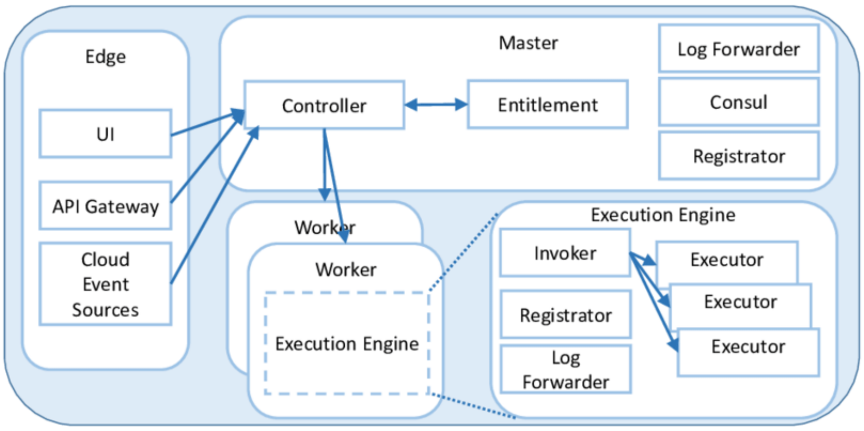

Figure 2.6: OpenWhisk Architecture [68]

2.2.4.3 Knative

Knative [20] is a platform for building, deploying, and managing modern serverless workloads on Kubernetes. The architecture of Knative as shows in 2.1 is designed to simplify the development, deployment, and management of serverless applications, and it consists of several key components that work together to provide a complete serverless platform.

The components of Knative's architecture include:

- **Knative Serving:** This component is responsible for managing the lifecycle of serverless applications, including managing their deployment, scaling, and handling of incoming requests. It uses a custom resource definition (CRD) and a set of controllers to manage the state of applications and handle events. Architecture of Knative Serving is shown as Figure 2.1.
- **Knative Eventing:** This component provides event-driven and asynchronous communication between components, enabling the development of event-driven microservices.

It uses a pub/sub model, and provides a way to manage event sources and event consumers, as well as event processing pipelines.

- **Knative Build:** This component is a CI/CD platform that provides a way to build and package applications into containers, and integrate them with Knative Serving and Eventing. It uses Kubernetes build and packaging tools such as Kaniko, Buildah, and Skaffold.

The underlying infrastructure that powers Knative is Kubernetes, which provides a powerful and flexible platform for managing containers and services. Knative leverages Kubernetes to provide scalability, high availability, and resiliency to serverless applications, and makes it possible to build and run complex serverless workloads on Kubernetes.

2.3 Multi-Cluster Kubernetes

Multi-cluster Kubernetes is an environment that uses multiple Kubernetes clusters. These clusters can be located on the same physical host or on different hosts within the same data center, or even in different clouds across different countries, enabling a multi-cloud environment [69]. Multi-cluster Kubernetes does have multitude of advantages such as: Tenant Isolation, High Availability, etc. In Kubernetes, the two challenges which arise when setting up a multi-cluster topology are as follows:

- **Synchronization:** Mechanism to synchronize cluster object status one-way or two-way between clusters depending on design of the cluster topology. In a one-way cluster, one cluster acts as provider and other as consumer i.e., one cluster can offload pods to other cluster, by not the other way around. In two-way clusters, both the clusters can offload pods onto each other, each cluster has the capability to act as provider and consumer cluster at the same time.

- **Interconnection:** Mechanism to enable communication between services is different clusters.

So for further discussion, we will split the explanation into two section which discusses projects which helps to setup multi-cluster control plane and projects for network interconnection.

2.3.1 Multi-Cluster Control Plane

This section discusses about projects whose work are on the area of setting up a control plane for a multi-cluster Kubernetes setup. Comparison of those projects are shown in Table 2.1

Open-source projects mentioned in Table 2.1 are compared on different aspects such as:

- **Seamless Scheduling:** Seamless scheduling in a multi-cluster Kubernetes setup refers to the ability to schedule workloads across multiple clusters without the need for manual intervention or complex configuration. It allows organizations to manage their

Criteria	**Liqo** [70]	**Admiralty** [71]	**Tensile-Kube** [72]	**KubeFed** [73]	**Argo CD** [74]	**Fleet** [75]	**FluxCD** [76]
Seamless Scheduling	Yes	Yes	Yes	No	No	No	No
Decentralized Governance	Yes	Yes	Yes	No	Yes	Yes	Yes
Application Extra APIs	Yes	Yes	Yes	No	No	No	No
Dynamic Cluster Discovery	Yes	No	No	No	No	No	No

Table 2.1: Comparison of Multi-Cluster Control Plane projects [77]

resources efficiently by distributing workloads across different clusters based on their specific needs and priorities. With seamless scheduling, organizations can ensure that their applications are running on the most appropriate cluster and have access to the necessary resources, leading to better performance and reliability.

- **Decentralized Governance:** Decentralized governance in a multi-cluster Kubernetes setup refers to the distribution of decision-making authority across different clusters. It means that each cluster is managed independently, with its own set of policies and procedures, and that decisions are made at the local level rather than being centralized. At the same time, it requires coordination and communication between different teams to ensure that the overall system is functioning effectively and efficiently.
- **Application Extra APIs:** In a multi-cluster Kubernetes setup, application extra APIs are custom APIs that are designed to enable specific functionality for an application. These APIs are implemented on top of the Kubernetes API and allow developers to create custom resources and controllers that can be used to manage their applications across multiple clusters. For example, an application extra API can be used to manage application deployments, scale applications, or manage storage resources. By providing additional abstraction and automation layers, application extra APIs can simplify the deployment and management of complex applications in a multi-cluster environment.
- **Dynamic Cluster Discovery:** Dynamic cluster discovery in a multi-cluster Kubernetes setup refers to the automatic discovery of new clusters as they are added to the environment. It allows organizations to manage a large number of clusters more easily by automatically incorporating them into the overall system without requiring manual intervention. This can be accomplished through the use of discovery services or other mechanisms that automatically detect the presence of new clusters and integrate them into the existing infrastructure. Dynamic cluster discovery enables organizations to scale their Kubernetes environment more efficiently and with less administrative overhead, which can result in significant cost savings and increased operational agility.

2.3.2 Network Interconnection Projects

This section discusses about projects whose work are on the area of setting up a network interconnection for a multi-cluster Kubernetes setup. Comparison of those projects are shown in Table 2.2

Criteria	**Liqo [70]**	**Cilium [78]**	**Submariner [79]**	**Skupper [80]**	**Istio [81]**	**Linkerd [82]**
Architecture	Overlay Network and Gateway	Node to Node traffic	Overlay Network and Gateway	L7 Vitrual Network	Gateway based	Gateway based
Interconnection setup	Peer-To -Peer, Automatic	Manual	Broker-based, manual	Manual	Manual	Manual
Secure Tunnel Technology	Wireguard	No	IPsec	TLS	TLS	TLS
CNI Agnostic	Yes	No	Yes	Yes	Yes	Yes
Multi-Cluster Services	Yes	Yes	Limited	Yes	Yes	Yes
Seamless Cluster Extension	Yes	Yes	Yes	No	No	No
Support for Overlapped IPs	Yes	No	No	Yes	Yes	Yes

Table 2.2: Comparison of Network Interconnection projects [77]

Open-source projects mentioned in Table 2.2 are compared on different aspects such as:

- **Architecture:** Architecture refers to the overall design and structure of the network infrastructure that connects different clusters. This requires careful consideration of factors such as network topology, routing protocols, IP addressing, and network security policies. A well-designed network architecture is essential for enabling seamless communication and workload management across multiple clusters in a Kubernetes environment.
- **Interconnection Setup:** An interconnection setup refers to the method or approach used to establish connectivity and communication between different clusters. There are several possible approaches to interconnecting clusters, including peer-to-peer, broker-based, and manual methods.
- **Secure Tunnel Technology:** A secure tunnel technology refers to method of establish an encrypted connection between clusters to ensure communication and data transfer.

Secure tunneling can be used to establish a direct connection between clusters or to route traffic through an intermediary such as a load balancer or a proxy server. Some examples of secure tunnel technologies include Secure Shell (SSH) tunneling, Transport Layer Security (TLS) tunneling, and Virtual Private Network (VPN) tunneling.

- **CNI Agnostic:** CNI agnostic refers to the ability of a networking solution to work with any Container Network Interface (CNI) plugin, regardless of the specific implementation used by each cluster. CNI is a specification for networking in container orchestration platforms like Kubernetes, which allows different networking solutions to be used with a consistent interface.
- **Multi-Cluster Services:** Multi-cluster services refer to a mechanism for enabling services to be exposed across multiple clusters as a single entity. Multi-cluster services provide a way to distribute workloads across multiple clusters while maintaining a consistent view of the service for clients.
- **Seamless Cluster Extension:** Seamless cluster extension refers to the ability to easily add new clusters to an existing cluster topology without interrupting the ongoing operation of the system. With seamless cluster extension, a new cluster can be added to the existing infrastructure without the need to manually reconfigure services or modify networking settings.
- **Support for overlapped IPs:** Support for overlapped IPs refers to the ability of the networking solution to handle IP address conflicts that may arise when multiple clusters are interconnected. Overlapping IP addresses can occur when different clusters have been assigned the same IP address range, which can cause network communication issues. To avoid these conflicts, a networking solution must be able to handle IP address overlap by isolating each cluster's IP address range and mapping the IP addresses to unique addresses in the wider network. This allows the different clusters to communicate with each other without any IP address conflicts.

2.3.3 Liqo

Liqo is an open-source project that enables dynamic and seamless Kubernetes multi-cluster topologies, supporting heterogeneous on-premise, cloud and edge infrastructures [70]. As shown in Table 2.1 and table 2.2, Liqo has the competence to provide multi-cluster control plane and also provides a network fabric [83], which extends Kubernetes network model across multiple clusters, enabling communication between pods within cluster and with pods running in different clusters (East - West communication) between pods deployed in independent clusters.

2.3.3.1 Peering and Offloading

Peering [84] involves establishing a unidirectional resource and service consumption relationship between two clusters, where one cluster (referred to as the consumer) is granted the

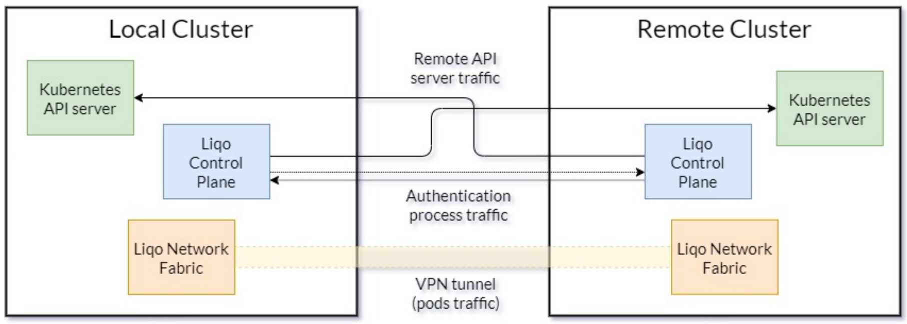

Figure 2.7: Out-of-band Control Plane setup in Liqo [84]

ability to offload tasks to another remote cluster (referred to as the provider), but not the other way around. In this scenario, the consumer creates an outgoing peering connection towards the provider, while the provider is subjected to an incoming peering connection from the consumer. This allows for the two clusters to interconnect and enables the consumer to access the services and resources provided by the provider cluster.

The virtual node abstraction plays a crucial role in enabling workload offloading[85] in a multi-cluster Kubernetes setup. This concept involves creating a virtual node in the local cluster after the peering process has been completed, which represents and aggregates the subset of shared resources from the remote cluster. By doing so, the local cluster can be extended seamlessly, and the new node and its capabilities can be considered by the vanilla Kubernetes scheduler while determining the best location for executing workloads. Additionally, this approach is in full compliance with standard Kubernetes APIs, making it possible to interact with and inspect offloaded pods as if they were running locally in the cluster.

Liqo's implementation of the virtual node abstraction leverages an updated version of the Virtual Kubelet [86] project. A virtual kubelet [87] can be used as a replacement for a traditional kubelet, but operates on non-physical nodes. In the Liqo context, it interacts with both the local and remote clusters' Kubernetes API servers to carry out three main tasks [85]:

- Generate the virtual node resource and maintain its status according to the negotiated configuration.
- Offload the local pods assigned to the corresponding (virtual) node to the remote cluster while synchronizing their status.
- Ensure that the ancillary artifacts (e.g., ConfigMaps, Services, Secrets) necessary for the proper execution of the offloaded workloads are reflected and synchronized between the local and remote clusters. This capability is referred to as resource reflection.

Liqo connects two different clusters using peering methodology, which includes four main tasks, which are as follows:

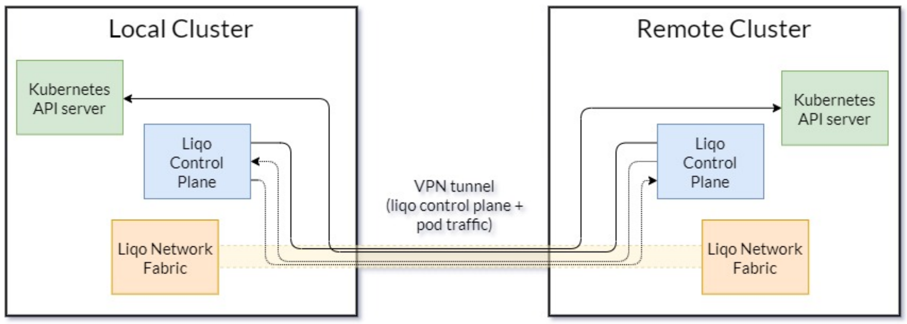

Figure 2.8: In-band Control Plane setup in Liqo [84]

- **Authentication:** Token is generated in the source cluster, which provides resource to target cluster to deploy workloads. Generated token is then shared with target cluster which authenticates and grants access to limited liqo-related resources, such as kubernetes objects which should reflect in target cluster when the pod is offloaded. Those Kubernetes objects are: *Pods, Services, EndPointSlices, Ingresses, PersistentVolumeClaims, PersistentVolumes, ConfigMaps, Secrets*

- **Parameters Negotiation:** Automatic exchange of parameters between two clusters for establishing a peering relationship, with details on shared resources, VPN setup, and no manual intervention required.

- **Virtual Node Setup:** The consumer cluster creates a virtual node for managing resources shared by the provider, allowing for transparent task offloading. This process follows standard Kubernetes practices and requires no modifications to application deployment APIs.

- **Network Fabric Setup:** The two clusters establish a secure cross-cluster VPN tunnel by configuring their network fabric based on agreed parameters, enabling seamless communication between local and offloaded pods independent of CNI plugin and configuration.

Characteristics	**Out-of-band**	**In-band**
Control Plane Traffic	Flows outside VPN tunnel	Flows inside VPN tunnel
Configuration	Dynamic	Static
Multi-Endpoint support	Yes	No

Table 2.3: Out-of-band peering vs In-band peering

Peering in Liqo can be done in two ways: Out-of-band peering as shown in Figure 2.7 and In-band peering as shown in Figure 2.8. Comparison of those two types of setup is listed in Table 2.3.

2.3.3.2 Network Fabric

Network Farbic [83] ensures inter-pod communicability across all clusters. Since each cluster has independent mechanism to setup CNI, which assigns unique IP for every pod. Since, the CNI is configured only on one cluster level, two pods in different clusters having same IP is possible. To avoid such IP conflicts within pods in different cluster, network fabric solution should adopt the mechanism of Network Address Translation (NAT). Usually it is a mapping of multiple local private IPs to a single IP visible outside cluster. This will be a costly operation in terms of encapsulating every data packets and transferring and de-encapsulation at the receiver's end. When CNI can be configured in such a way that the IPs assigned are from disjointed ranges for each cluster, then a NAT-less approach will be adopted by network fabric implementation.

This is due to the inherent variability of cluster configurations, including differing CNI plugin usage, which precludes the possibility of guaranteeing a consistent and non-overlapping PodCIDR. As such, in cases where PodCIDR disjointness exists, address translation mechanisms may be necessary to sustain inter-pod communicability. Interconnection between peered clusters are done using secure VPN tunnels, with the help of WireGuard [88]. Liqo Gateway is responsible for populating routing table and NAT rules by leveraging *iptables*. The overlay networks are also used to forward all traffic from local pods to remote cluster pods through liqo gateway. Network gateway is pictorially depicted in Figure 2.9

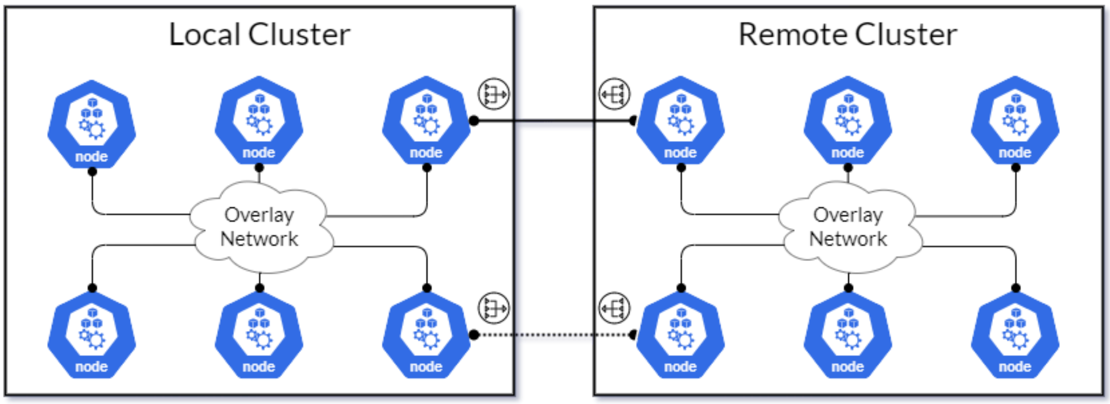

Figure 2.9: Network Fabric - Liqo [83]

2.4 Green Software and Intensity Specification

Green Software refers to software systems designed and developed to optimize resource utilization, minimize waste, and reduce the carbon footprint of the software development life cycle. This encompasses a variety of practices such as energy-efficient algorithms, efficient data management, sustainable development processes, and environmentally conscious disposal of software systems. From a technical perspective, Green Software is characterized by the implementation of energy-efficient algorithms, optimized resource utilization, reduction of waste in software development processes, and the use of eco-friendly computing practices. These include but are not limited to utilizing virtualization techniques, efficient data management, and the application of sustainable development methodologies, as well as adherence to environmentally conscious software disposal practices. The development of Green Software often entails a sophisticated interplay between software engineering, energy-efficient computing, and environmental sustainability.

2.4.1 Green Software Foundation

The Green Software Foundation is an organization dedicated to reducing the environmental impact of software systems through the utilization of innovative technologies and best practices in software engineering. Their primary goal is to promote the development and adoption of eco-friendly software by creating technical specifications and methodologies that promote sustainable software development.

As part of its mission, the Green Software Foundation (GSF) engages in various initiatives and projects aimed at reducing the carbon footprint of software systems. These projects are centered around the implementation of green software engineering principles and practices, including the optimization of software resource utilization, the adoption of energy-efficient hardware and cloud infrastructures, and the implementation of lifecycle management strategies that minimize the carbon emissions associated with software development, deployment, and maintenance. Through these efforts, the GSF aims to establish itself as a leading voice in the advancement of sustainable software development and deployment practices, and to drive the widespread adoption of environmentally responsible software development practices within the industry.

The Green Software Foundation (GSF) drives various initiatives and projects aimed at reducing the carbon footprint of the software industry. Some examples of their projects include:

- Development and promotion of the Software Carbon Intensity (SCI) specification, which outlines the methodology for calculating the carbon footprint of software applications.
- Collaboration with software companies, researchers, and experts to identify and develop innovative solutions to reduce the carbon impact of software systems.
- Promotion of energy-efficient software engineering practices through educational programs, workshops, and training sessions.
- Engagement with stakeholders in the software industry to drive industry-wide adoption of best practices for reducing the carbon footprint of software.
- Research and development of open-source tools and technologies to enable software engineers to measure, monitor, and reduce the carbon impact of their systems.

These initiatives and projects are aimed at promoting sustainability in the software industry and mitigating the environmental impact of software systems through technology, education, and collaboration.

2.4.2 Software Carbon Intensity

The Software Carbon Intensity [89] technical specification is a detailed guide for determining the carbon footprint of a software application by quantifying its associated carbon emissions. This methodology involves calculating the total carbon emissions produced during the application's life cycle and converting it into a rate that facilitates practical emission reduction efforts, known as abatement. The carbon intensity of a software application is influenced by the carbon intensity of electricity consumption during its deployment, making it important to take into account the time and location of usage. The carbon intensity rate provides valuable information for software engineers and developers in the design, development, and deployment stages of software application development, as it gives them a means to evaluate the carbon impact of their work. Software Carbon Intensity can be calculated using following formula as shown in Equation 2.1

$$Sotware\ Carbon\ Intensity = ((E * I) + M)/R \tag{2.1}$$

Where:

- (E) - Energy consumption for different components of the software boundary over a given time period.
  Examples:
  - CPU/GPUs at different percentages of utilisation
  - Data Storage
  - Memory allocation
  - Data transferred over a network
- (I) - Emissions factors. These may be regional yearly averages to begin with, but should ideally be marginal, and more granular than that.
- (M) - Embodied emissions data for servers, mobile devices and laptops
- (R) - The functional unit defines how your application scales. For instance, if your application scales by APIs then choose API as your functional unit.

3 Related Work

Despite a plethora of research in the domain of serverless scheduling, carbon-aware computing in data centers, and scheduling techniques for Kubernetes, there is limited investigation at the intersection of these fields. Thus, this research will examine the current advancements in these areas, and posits how our research sets itself apart as a unique contribution, in the field of carbon-aware scheduling scheme specifically designed for serverless workloads.

3.1 Serverless Scheduling

The scheduling of serverless functions can be compartmentalized into two distinct stages: the runtime configuration phase and the function scheduling phase. In the runtime configuration phase, the serverless platform retrieves the function's source codes and required resource allocation from a database. The serverless platform assesses the current state of the system, including the resource utilization of active instances and the concurrency of running instances. In the function scheduling phase, serverless platforms typically rely on a general-purpose task scheduler or delegate this responsibility to a Platform-as-a-Service (PaaS) platform, such as Kubernetes [19], as is the case with the Knative platform [20]. The PaaS platform then schedules functions according to its own discretion, ensuring efficient utilization of resources and meeting the desired Quality of service (QoS) requirements. General process of a serverless scheduler is illustrated in Figure 3.1.

Serverless workloads are characterized by their burstiness, short and variable execution times, statelessness, and utilization of single cores [90]. However, traditional general-purpose schedulers are not equipped to handle such workloads effectively. Kaffes et al [90] have proposed a centralized scheduler that can allocate functions to processor cores instead of physical servers, which better addresses the unique requirements of serverless workloads. In this paper, the authors put forward the benefits of implementing a centralized scheduler, specifically citing two advantages: 1) its ability to possess a comprehensive overview of the cluster resources, thereby providing enhanced elasticity to users, and 2) its facility to adapt to varying workload demands with ease. Our research also focuses on creating a centralized scheduler that can comprehend the overall cluster resources, however, the authors suggest that the core-level granularity as described in the paper will limit optimal utilization of these resources. This is because most serverless functions do not require a full core allocation, rather, it is often necessary to consolidate multiple functions onto a single core, making such detailed granularity unnecessary.

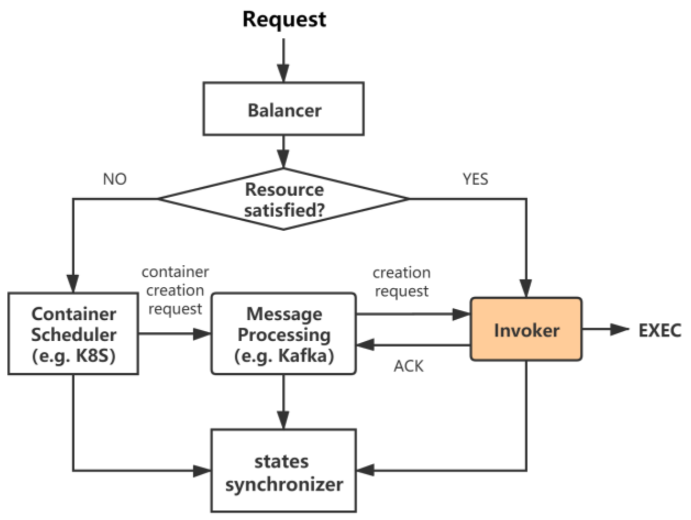

Figure 3.1: General process of serverless scheduler [91]

FnSched, a scheduler introduced by Suresh et al [92], is a cost-effective solution that is designed to minimise the provider resource cost but also ensures performance requirements for serverless functions. The scheduler categorizes functions based on resource utilization and lifetime behavior, enabling runtime regulation of CPU shares for colocated functions. FnSched also has the ability to dynamically scale host resources, with the capability to add or remove hosts as needed. Built on top of OpenWhisk [34]'s controller component, FnSched outperforms traditional systems, such as OpenWhisk and Linux scheduler, by reducing host usage by 36-55%. In their work, they adopt a policy of sacrificing performanace in place of cost minimization. This is achieved by regulating CPU shares between functions and actively managing resources in a compute-intensive manner [93]. On the other hand, GreenCourier has a different objective, which is to reduce carbon emissions during function execution rather than minimize cost. The motivation and goal of GreenCourier are different from FnSched, as it does not compromise function performance for the sake of cost reduction.

Szalay et al [94] propose a novel solution for scheduling real-time serverless workloads in multi-node clusters. The solution includes a heuristic partitioning scheduling algorithm and an analytical model to deploy real-time functions in containers. To use the solution, users must provide the tolerated response time and period of the function. The response time refers to the maximum time it takes for the function's response to reach the user's ingress point, while the period defines the frequency of function requests. In their approach, available cores are partitioned into two categories: one exclusively reserved for real-time functions and the other for non-real-time functions. When the load on real-time functions is high, non-real-time functions may be preempted, potentially leading to starvation of non-real-time functions. The authors state that the scheduler will leave as many processors as possible available, but do not provide a mechanism to ensure this. This scheduling strategy is known as "Partitioned

Earliest Deadline First scheduling". One of the assumptions made in their research is that real-time serverless functions operate on a deterministic network both between hosts and within hosts. While this is a challenging assumption, the authors believe that this requirement will be met in the future. GreenCourier and RT-FaaS get along in terms of scheduling functions in a multi-node clusters, but other than that there is no similarities between two research.

3.2 Sustainable scheduling in data centers

Most of the research in sustainable scheduling in data centres has been focused on energy-aware approaches. In this section, we will review the current research on sustainable scheduling, with a focus on energy-awareness, and highlight how GreenCourier differs from the other approaches.

Alahmadi et al [95] propose a novel approach for scheduling, sharing and migration of virtual machines to reduce energy consumption. They report that average resource utilization is as low as 20% and energy consumption of idle resources are as high as 60% [96]. Hence, in this paper they propose scheduling a task on the minimum number of virtual machines using First Fit Decreasing approach. In First Fit Decreasing approach, they assume the tasks come sequentially and they are assigned to the first node that can accommodate it, where all the nodes are order in decreasing order of utilization rate. On top of First Fit Decreasing approach, they implement a enhancement of reuse and migration technique. Approach on this paper matches almost with the approach of GreenCourier, where new tasks are preferred to be deployed in a specific node or cluster. But the parameter on which decisions are taken are completely different, in GreenCourier's case it depends on carbon efficiency of that region, where as this paper concentrates on deploying task on node with high utilization rate. And the tasks are defined as Operating System (OS) processes, but in GreenCourier the tasks are handled as pods/functions. And GreenCourier is capable of scheduling tasks over clusters distributed over different regions.

Zhu et al [97] propose a real-time task scheduling approach in virtualized cloud environments that uses a rolling-horizon scheduling architecture. This is an energy-aware scheduling algorithm called Energy Aware Rolling Horizon (EARH), which is capable of balancing task schedulability and energy conservation. In the EARH approach, jobs are composed of batches of tasks are added to a queue maintained by the controller. The controller creates a plan for the execution order of tasks within and between jobs. When the controller determines that a job is likely to finish within its estimated deadline, it scales up the number of virtual machines, and when the load is lower, it scales down the number. EARH performs a similar function to horizontal pod auto scaling in Kubernetes, but instead of creating pods, it creates virtual machines to run tasks. EARH is limited to handling jobs composed of batch tasks, and its goal is to minimize energy consumption while executing a set number of jobs. Unlike GreenCourier, EARH lacks any intelligent selection logic and focuses solely on energy conservation, while GreenCourier employs a node selection logic to minimize carbon emissions into the environment.

Goiri et al [98] present a parallel batch job scheduler for a data center that is powered by a photovoltaic solar array and uses the electrical grid as a backup source. GreenSlot utilizes prediction technology to forecast the availability of solar energy and schedules jobs to maximize the consumption of green energy while minimizing the consumption of brown energy. *Brown energy* [98, 99] have been introduced to refer to the energy produced from non-renewable sources that pollute the environment, in opposition to *green energy* which is generated from clean and renewable sources. The scheduler prioritizes brown energy consumption during times of low energy costs. This approach has been shown to increase the consumption of green energy by 117% and decrease energy costs by 39%. The scheduler is designed for tasks that are not time-sensitive, meaning that they can be scheduled for future execution, rather than being executed immediately upon receipt. In conclusion, GreenSlot and GreenCourier have different goals and serve different use cases. While GreenSlot aims to maximize the consumption of green energy and reduce the use of brown energy in a data center, it requires additional information such as the number of nodes, expected running time, and deadlines. It schedules batch jobs that are not time-sensitive and can wait for the availability of green energy. On the other hand, GreenCourier is a scheduler that prioritizes minimal carbon emission in function execution real-time, rather than maximizing the use of green energy. It employs an intelligent node selection logic to choose the cluster with the greenest energy at a given point in time. Both solutions cater to different business requirements and serve unique purposes.

The study conducted by Li et al. in [100] introduces the GreenWorks framework, specifically geared towards hybrid data centers powered by a mixture of renewable energy sources. The framework features a cross-layer power management strategy that accounts for the unique temporal characteristics and capacity limitations of each energy source. This research highlights the limitations of current green data center approaches, which typically rely on one of three methods (load scaling, battery discharge, or load following), and fail to capture the potential synergies that can be achieved through a multi-source energy mix.

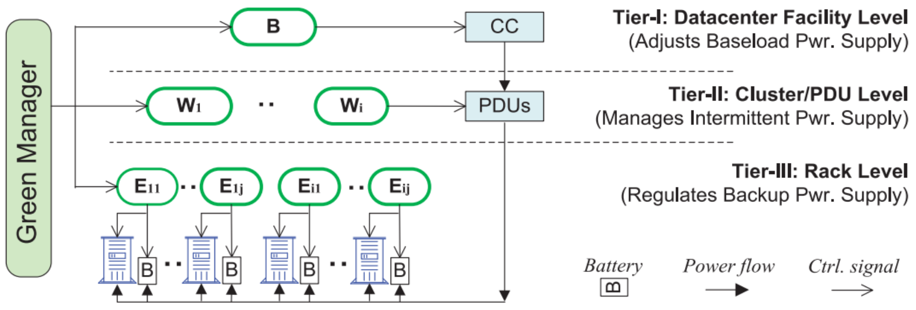
Green Manager
B
CC
Tier-I: Datacenter Facility Level
(Adjusts Baseload Pwr. Supply)
W1
Wi
PDUs
Tier-II: Cluster/PDU Level
(Manages Intermittent Pwr. Supply)
E11
E1j
Ei1
Eij
Tier-III: Rack Level
(Regulates Backup Pwr. Supply)
B
Battery
Power flow
Ctrl. signal

Figure 3.2: GreenWorks power management hierarchy [100]

As a result, they tend to result in suboptimal design decisions. Their system consists of two elements: green workers - system-specific power control modules for each different source, 2) green manager - a global controller of different green workers. This work is more a coordination strategy where various source of power is present, rather than taking control at the level of tasks which are controlled or scheduled using different strategy. In this paper, they explain the necessity of different workers, rather than merely adjusting server load or power budget. Solution they provide in terms of GreenWork framework is a hierarchical coordinator as shown in Figure 3.2. GreenCourier completely differs from GreenWorks in multiple level - 1) GreenCourier controls scheduling of incoming functions, does not have control over quality of electricity being used at a data center, 2) GreenCourier does not actively manage life cycle of the functions deployed, it is taken care by serverless platform (Knative [20] in this case), 3) GreenWorks provides a efficient mechanism to manage power from different sources which are intermittent by nature, GreenCourier only takes into account of real-time carbon efficiency of different regions, does not use any predictive mechanism or use history data to create a pattern.

In general, existing literature on energy aware scheduling, they focus one of following strategy:

- Scheduling tasks by increasing or decreasing frequency and voltage depending on task criticality and priority
- Concentrated scheduling over number of nodes/processor, with as minimal resource as possible
- Delaying task execution which are time immune (not necessarily executed on the moment but put in a queue and wait for preferable time), such that it still satisfied user's requirement

3.3 Scheduling in Kubernetes

Kubernetes [19] has been around since 2014, currently it is the industry's go-to solution for container deployment and orchestration. Heterogeneity in characteristics of workload in industry is well known and solution offered by Kubernetes is not enough to bridge the gap between requirement and offerings. So, lot of other projects whose intention was to extend the platform to cater customer needs were developed. One such point of extension of platform happened in scheduler module, which is the sole decision maker in terms of scheduling pods inside the cluster. Since then there has been lot of research work done on this field and GreenCourier is also no exception from above fact. Rejiba et al [64] produced their surveyed custom scheduling in Kubernetes and discuss the common problems and solution approaches in their paper.

Kaur et al [101] propose a controller, named Kubernetes-based Energy and Interference Driven Scheduler(KEIDS) for container management in edge cloud by taking in to account the emission of carbon footprints, interference and energy consumption. They design their solution based on a Multi-Objective Optimization Problem (MOOP). Proposed solution performs better in terms of energy utilization, reduced carbon footprint and minimal interference by 14.42%, 47%, 31.83% respectively, when compared with default First Come First Serve (FCFS) scheduler. Mathematical formulation of above said objectives is shows in Equation 3.1

$$\mathbf{F}\left(\mathfrak{T}_{ijkl}\right) = \min\ f\left(\mathbf{F}_1\left(\mathfrak{T}_{ijkl}\right), \mathbf{F}_2\left(\mathfrak{T}_{ijkl}\right), \mathbf{F}_3\left(\mathfrak{T}_{ijkl}\right)\right) \tag{3.1}$$

Equation 3.1 represents the MOOP for efficient scheduling of pods to the available nodes in multi-constraint setup, with a binary decision variable as defined in Equation 3.2.

$$\mathfrak{T}^{p}_{ijkl} = \begin{cases} 1 : & \text{if the } i\text{th container (associated with the } p\text{th job)} \\ & \text{is mapped to the } j\text{th pod deployed on the } k\text{th} \\ & \text{node of the } l\text{thcluster} \\ 0 : & \text{Otherwise.} \end{cases} \tag{3.2}$$

Formulated MOOP is a typical case of Integer Linear Programming, which they solve using Mosek [102]. As established, KEIDS proposes a solution to solve a multi-objective scheduler problem. In the case of GreenCourier, only objective is to minimise the carbon emitted during execution of functions, to achieve that GreenCourier uses a multi-cluster topology of geographically distributed clusters, where as KEIDS only works in a conventional setup of Kubernetes cluster.

Rocha et al [103] introduces a task-oriented and energy-aware orchestrator, named HEATS used for containerized application targeting clusters with heterogeneous resources. HEATS analyses the cluster for its performance and energy features, then opportunistically migrates them to different nodes based on the previous learning. HEATS allows users to hit a balance between performance and energy requirements. It integrates a modified version of the Kubernetes platform, with a cluster control plane that is augmented with machine learning capabilities to make informed decisions on pod migration, based on performance and energy requirements. Additionally, HEATS features a novel scheduler policy that considers a user-specified performance trade-off value to determine the degree of performance reduction that the user is willing to tolerate. Their evaluation result shows that HEATS outperformed default scheduler in terms of energy savings, by compromising on performance. Basic difference between GreenCourier and HEATS is that, GreenCourier does not actively compromise performance of the workloads at all and it is not concerned above minimising the energy consumed during the execution. GreenCourier's primary concern is reduction of carbon emission by using comparative clean energy. HEATS and GreenCourier, though a scheduler designed to scheduler pods in a energy efficient way, they both adopt two different approaches to reduce carbon emission during execution.

Townend et al [104] proposes a holistic scheduling system replaces default scheduler in Kubernetes, by taking into account of software and hardware model. By deploying this scheme, they were able to achieve 10-20% of power consumption reduction. They shows that introduction of hardware modelling into to a software-based system produces significant improvement in data center efficiency. They create such software models from past history data and other online data and predict the amount of resources that will be required to execute a workload and determine the duration to achieve that. They strive to build a predictive capability to assess the impact of each incoming container on the nodes. They use this predictive capability to intelligently schedule pods to a node in optimal way - tuned to conserve energy or improve performance or both depending on the configuration. Two main focus of this work is to improve performance or conserve energy by adapting Dynamic Voltage Frequency Scaling (DVFS) method, which clocks the performance of CPU by reduction of power. GreenCourier's focus is not either of those two, but the end goal achieved by both the work is same, as explain for previous works, the approach adopted is completely different.

Menouer [105] proposes a multi-criteria scheduling algorithm which uses Technique for the Order of Prioritisation by Similarity to Ideal Solution (TOPSIS) algorithm. This algorithm considers following criteria for scheduling:

- CPU Utilization rate
- Memory Utilization rate
- Disk Utilization rate
- Power Consumption
- Number of running containers in each node
- Time for transmitting the image selected by the user to that node

The aim of this study is to incorporate various considerations, including energy consumption, into the scheduling process to make informed decisions. The evaluation results demonstrate that this approach leads to a reduction in power consumption, as well as a decrease in both makespan and average waiting times for containers. To accomplish this, modifications were made to the existing Kubernetes framework code, extending beyond mere changes to the scheduler code alone.

It can be discerned from the above-cited literature that a prevailing commonality among the various strategies employed to minimize energy consumption is the utilization of techniques that either modulate the clock frequency of processors or defer the execution of tasks until they can be accommodated within the existing cluster configuration, rather than scaling up the infrastructure dynamically. However, this approach is only applicable to certain use-cases that permit a degree of latency in the execution of tasks, and does not address the requirement for real-time deployment of workloads that necessitate prompt accessibility for end-users. It has been observed that some of the research studies provide solutions to multi-objective optimization problems, one of the objectives being the reduction of energy consumption, which translates to a decrease in the carbon emissions generated by the workload. In the context of the GreenCourier system, a novel approach is adopted to address this objective by identifying the most energy-efficient region for deploying the functions, thereby reducing the carbon footprint during their execution. Although there exist certain technical similarities with other works, the methodology adopted by GreenCourier is distinct and dissimilar from the conventional approaches.

4 System Design

In this section, we will delve deeper into technical characteristics of the thesis including the design choices and system architecture. Overall system architecture will be broken into multiple sub-systems and detailed explanation and implementation details will be discussed.

4.1 System Architecture

The primary aim of this study is to design a scheduling framework that can effectively detect environmentally sustainable regions and prioritize scheduling serverless functions in those regions. The purpose of this scheduling approach is to reduce the carbon footprint associated with function execution. In typical commercial Function-as-a-Service (FaaS) offerings, users are not afforded the flexibility of selecting a region during runtime, but instead must specify a region during function creation. However, the characteristics of electricity in a given region can fluctuate over time due to various external factors such as light and wind availability. In order to optimize the use of renewable energy and consequently minimize carbon emissions, it is imperative to implement a policy that can dynamically select regions during runtime based on the quality of greenness in electricity available in various parts of the world or pre-selected regions of choice. To implement this kind of scheduling scheme, we need a multi-cluster setup, with a management plane cluster and multiple peer clusters, which are geographically distributed in different regions. Since Knative [20] has been decided as the serverless platform, because of its interoperability with public clouds and its dependence on Kubernetes [19], which is a production-grade container orchestration platform. And Liqo [70] to setup multi-cluster Kubernetes topology, because of its advantages over other parallel offerings as shown in Section 2.3.

The technical setup for this framework can be broadly categorized into three distinct sections: User, Management Cluster, and Peer Cluster. The user interface involves interaction with the management plane to deploy novel serverless functions and to use the load balancer incorporated at the management cluster to interface with the deployed functions. More specifically, the user establishes functions using Knative service specification by interfacing with the Knative control plane that is deployed in the management cluster. Within the management cluster, several disparate components are deployed, including Knative Serving, Liqo, GreenCourier, Metrics Server, and Virtual Kubelet [86], which serves to connect peer clusters through Liqo. All of the aforementioned components rely on Kubernetes to operate; therefore, as depicted in Figure 4.1, all of these components are deployed within a Kubernetes cluster. The Knative Serving components are essential to deploying functions and making them available to serve user requests. As previously discussed in Section 2.3.3, Liqo is utilized to establish a multi-cluster topology. The fundamental requirement for establishing a multi-cluster Kubernetes configuration is to have a multi-cluster control plane and network interconnectivity between clusters, both of which Liqo provides out-of-the-box. Liqo utilizes virtual kubelet to cloak a peer cluster as a virtual node and connect it to the management cluster as a node. GreenCourier is built on top of Kubernetes scheduler framework, therefore it is not required to develop and maintain core functionalities of Kubernetes scheduler. Kubernetes Scheduler Framework as discussed in Section 2.2.3 is used to build GreenCourier. GreenCourier extends *Score* extension points and implements custom logic to prioritize nodes running in region which uses comparatively clean energy. GreenCourier is accountable for querying the carbon score from the metrics server and making runtime scheduling decisions to place pods in any of the peer clusters as shown in Figure 4.2. This decision is made using the carbon score reported by the metrics server, and GreenCourier assigns pods to a peer cluster that has the highest carbon score. If GreenCourier is unable to deploy pods to a peer cluster that has the highest carbon score, it assigns them to a peer cluster that has the next highest score. Metrics Server is capable to connecting to different sources which provides marginal emissions rate MOER. It represents the emission rate of the electricity [27] and it is quantified in terms of CO_2 lbs/MWh or g/kWh, depending on the type of source, unit is determined. As part of Peer Cluster, it is required to deploy Liqo as part of the cluster. Since, Liqo establishes network fabric, which maintains routing table and Network Address Translation (NAT) tables, by taking directions from Liqo control plane. As a peer cluster in a multi-cluster environment facilitated by Liqo, the deployment of Knative Serving components may not be necessary, as the Kubernetes objects created by Knative, namely the ReplicaSets, are automatically managed by Kubernetes itself. Since the management cluster exercises control over the pods deployed in peer clusters and the interconnection between clusters is established at the Kubernetes level, Knative Serving components are not required in the peer cluster. Instead, the overlay network established by Kubernetes and Liqo can be used by Knative to route traffic to the deployed pods.

In conclusion, the scheduling framework proposed in this study aims to dynamically select environmentally sustainable regions during runtime and prioritize scheduling serverless functions in those regions to reduce the carbon footprint associated with function execution. This framework requires a multi-cluster setup, including a management plane cluster and multiple peer clusters, and utilizes Knative as the serverless platform and Liqo to establish a multi-cluster topology. The management cluster hosts several components, including Knative Serving, Liqo, GreenCourier, Metrics Server, and Virtual Kubelet, all of which rely on Kubernetes to operate. GreenCourier is responsible for querying the carbon score from the metrics server and making runtime scheduling decisions to place pods in any of the peer clusters. Metrics Server is capable of connecting to different sources to provide marginal emissions rate. The pods objects are created when a function is invoked in Knative, which are then offloaded to one of the peer clusters, where the function is executed.

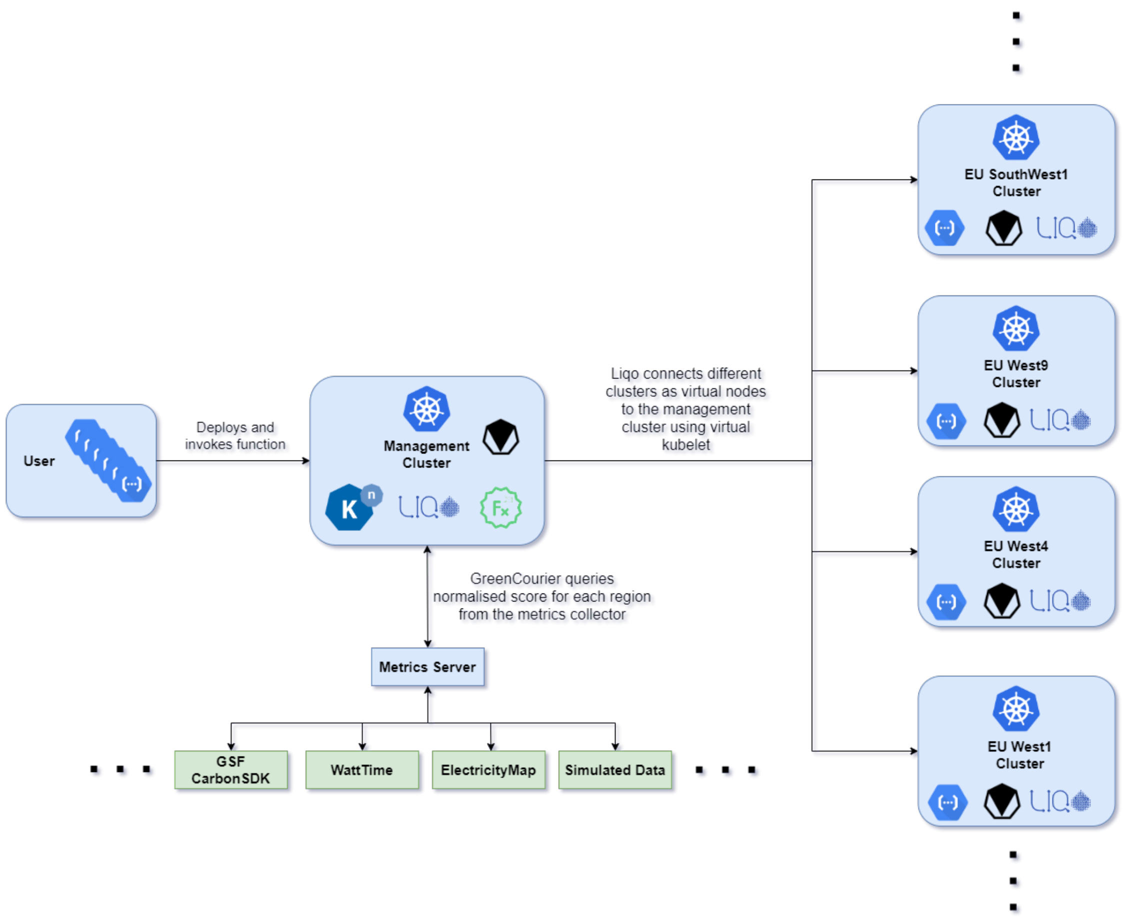
User
Deploys and invokes function
Management Cluster
Liqo connects different clusters as virtual nodes to the management cluster using virtual kubelet
GreenCourier queries normalised score for each region from the metrics collector
Metrics Server
GSF CarbonSDK
WattTime
ElectricityMap
Simulated Data
EU SouthWest1 Cluster
EU West9 Cluster
EU West4 Cluster
EU West1 Cluster

Figure 4.1: System Architecture - GreenCourier

4.2 Scheduling Workflow

Typical workflow of deploying functions in GreenCourier is shown in Figure 4.2. At first, ❶ user defines function as a specification file which contains information regarding container image and runtime configurations. To use GreenCourier for scheduling functions, user should set the following field - *.spec.schedulerName* to kube-carbon-scheduler. Using this specification file, user sends request to Knative Serving control plane to create function. Knative picks up the specification file ❷, and creates associated Kubernetes Objects like ReplicaSet and Service. When there is state change in ETCD [106], Kubernetes detects the state change and pulls the update from ETCD ❸, which sends changes in Kubernetes objects, in this case ReplicaSet. Kubernetes then acts on this request and creates associated Pod objects and updates the creation back to ETCD ❹. GreenCourier also listens to state change associated with Pod objects. Since new pods are created by Kubernetes control plane (to be precise, replicaset controller), GreenCourier scheduler gets newly created pod objects ❺. GreenCourier in parallel keeps scraping carbon score metrics from metrics server ❻, which has capability of querying carbon intensity data from multiple sources. Retrieved carbon score is stored in local cache within GreenCourier and this score is used to rank nodes during scheduling cycle. After evaluating every node, node with highest score is selected. This selected node is assigned with newly created pod object. To do this, GreenCourier edits *.spec.nodeName* field in pod specification and pushes the updated specification to ETCD ❼. Liqo does the work of Kubelet [87] in our setup, which is to look for pods which are assigned with node name that Kubelet is responsible for. In our case, Liqo cloaks a cluster as a virtual node in management cluster, with the help of Virtual Kubelet [86]. When Liqo sees a pod assigned to the virtual node which it is responsible for, it queries pod object from ETCD ❽. Once pod object is queried, Liqo initiates offloading ❾, more on offloading is explained in Section 2.3.3.1. Once the peer cluster, reconciles the pod creation request, it updates the state back to management cluster through Liqo ❿. Liqo then updates local routing table rules in such a way, that traffic to that pod is sent through secure tunnel established by Liqo between peer cluster and management cluster. It is required to update the endpoints objects created in ancillary during function creation. So Liqo initiated endpoint updation by sending update request to ETCD ⓫. Since EndPoint objects are listened by Kubernetes control plane, update is pulled by control plane ⓬. When such EndPoint objects are updated, Kubernetes initiates function updation ⓭, because function object created by Knative should be updated with IPs of pods running in peer cluster, so that Knative will forward function invocation to respective pod. Local snapshot of objects like Functions should be updated with latest version in Knative Serving components, so Knative initiates and updates local snapshot of Function object ⓮.

4.3 Coscheduling and Cluster Authorization

In this segment, we will delve into the intricacies of coscheduling and its pertinence to GreenCourier. Furthermore, the challenge of cluster authorization, a paramount aspect in the research procedure, will be scrutinized. To fulfill its mandate, GreenCourier necessitates

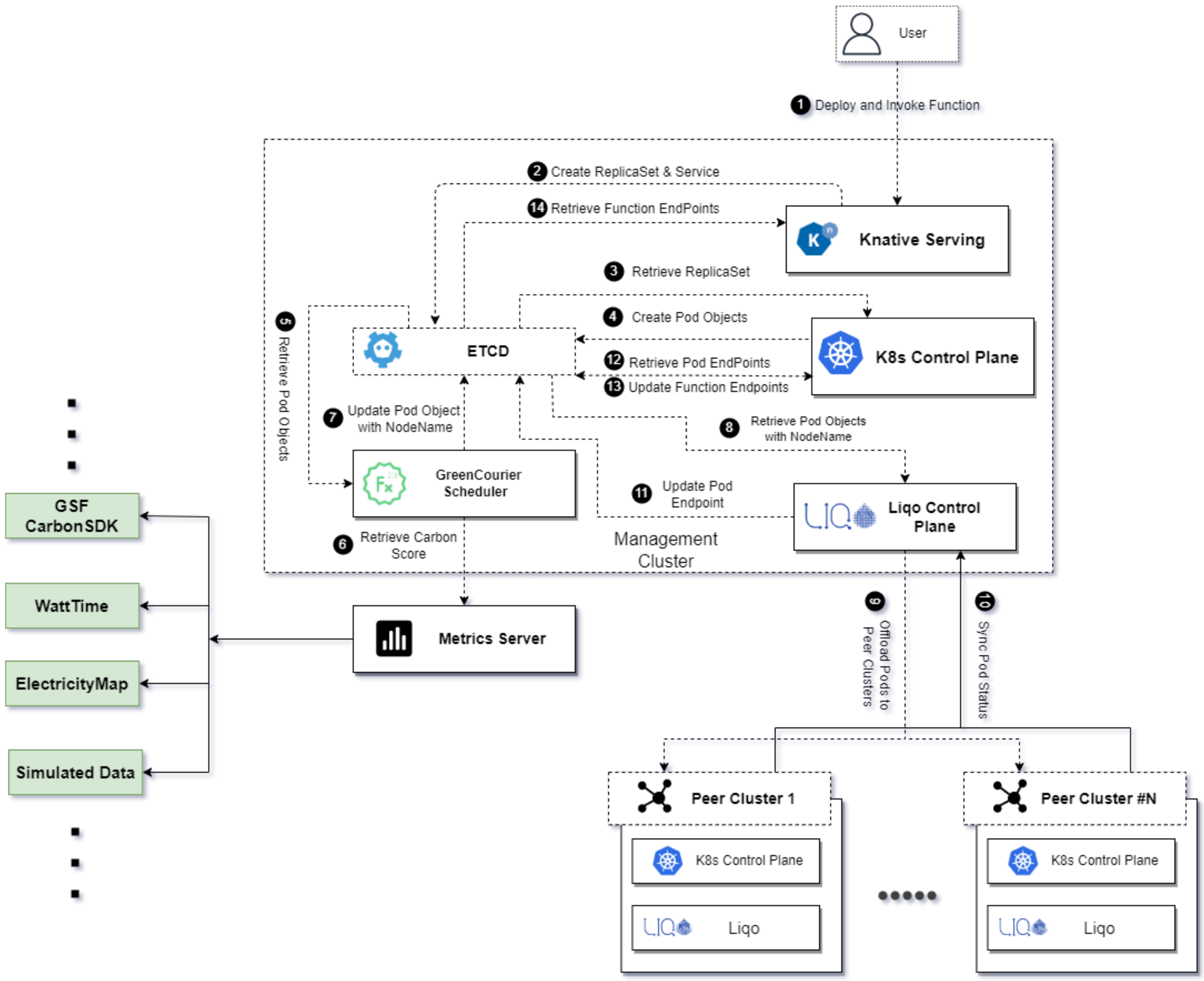

Figure 4.2: Workflow of typical Function Scheduling in GreenCourier

privileged access to the status of Kubernetes cluster objects, as well as the capability to perform modifications to these objects. This presents a significant obstacle that must be overcome during the course of research.

4.3.1 Coscheduling

Coscheduling refers to the simultaneous execution of two or more scheduling-related processes in concurrent systems. Essentially, two separate processes utilize the same snapshot of cluster resource status and run in parallel to deploy workloads for the same resource. This can result in a race condition, so to avoid this, one of the processes must be notified not to handle a particular resource request so the other process can take over and perform the necessary actions. GreenCourier operates in a similar manner, running in parallel with the Kubernetes scheduler, which also selects unassigned pods and attempts to assign them to a node through the application of various predicate and priority scheduling algorithms. Kubernetes itself provides capability to explicitly mention scheduler name in pod specification file. Scheduler name can be explicitly set in *.spec.schedulerName* field.

```
apiVersion: v1
kind: Pod
metadata:
    name: annotation-second-scheduler
    labels:
        name: multischeduler-example
spec:
    schedulerName: GreenCourier
    containers:
    - name: pod-with-second-annotation-container
      image: thandayuthapani/kube-carbon-scheduler:latest
```

4.3.2 Cluster Authorization

As a kubernetes scheduler, GreenCourier needs to watch on kubernetes objects and has to take appropriate action when there is changes in respective kubernetes objects. Kubernetes enforces authentication and authorization for every API requests sent. Kubernetes reviews following API request attributes: *user, group, extra, API, request path, API request Verb, HTTP request verb, resource, subresource, namespace, API group*. In short, GreenCourier should get explicit access rights for set of Kubernetes objects under particular resource and different verbs that GreenCourier can execute on that particular resource. For example, GreenCourier should have access to *GET, LIST, UPDATE, DELETE* verbs for Pod. Similarly other Kubernetes objects like ConfigMap, PersistentVolume, PersistentVolumeClaims, etc., are required for proper functioning of schedulers. Available request verbs are as follows: *GET, LIST, WATCH, UPDATE, PATCH, DELETE, DELETECOLLECTION*.

The implementation of Role-based access control (RBAC) in Kubernetes provides a secure solution to the authorization challenge faced by GreenCourier. This feature empowers users to assign an RBAC policy to GreenCourier, providing it the necessary authorization to access privileged Kubernetes objects and perform updates on cluster objects. The RBAC policy specifies the Kubernetes objects and request verbs that GreenCourier is permitted to access and execute, thereby ensuring a secure and controlled environment for resource utilization. The implementation of RBAC in Kubernetes is achieved through the use of *ClusterRole* and *ClusterRoleBinding* objects. A new *ClusterRole* object can be created to define the RBAC policy, and a specific *ClusterRole* can be assigned to a workload by using the *ClusterRoleBinding* object, granting it privileged access to objects defined in the scope. Many default Role-based access control (RBAC) policy are created by Kubernetes cluster, out of those three *ClusterRole* objects are required for GreenCourier to get access to Kubernetes objects. Those three *ClusterRole* objects are: *system:kube-scheduler, system:volume-scheduler, extension-apiserver-authentication-reader*. *ClusterRole* objects are bound to GreenCourier using *ClusterRoleBinding* objects. It can be done using following configuration:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
name: my-scheduler
namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
name: GreenCourier-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: GreenCourier
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: GreenCourier-as-volume-scheduler
subjects:
- kind: ServiceAccount
  name: GreenCourier
  namespace: kube-system
```

```
roleRef:
  kind: ClusterRole
  name: system:volume-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
name: GreenCourier-extension-apiserver-authentication-reader
namespace: kube-system
roleRef:
  kind: Role
  name: extension-apiserver-authentication-reader
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: GreenCourier
  namespace: kube-system
```

4.4 Metrics Server

The metrics server plays a crucial role in the functioning of the Carbon-Aware scheduling system as it acts as the central repository for calculating and normalizing the carbon efficiency score. The metrics server employs a straightforward methodology where it collects data from designated sources and subsequently exposes a REST API that can be utilized by the scheduler to retrieve the normalized score. This arrangement streamlines the process of obtaining the carbon efficiency score and ensures its seamless integration into the scheduling framework.
Currently, there are two different sources supported by metrics server, where it can readily scrape data by simple change in configuration of metrics server. Those two sources are as follows: *WattTime [26] and Green Software Foundation (GSF)'s Carbon Aware SDK [107]*. Metrics server is written in such a way, any new sources can be added with ease.

4.4.1 WattTime Data Source

WattTime leverages real-time grid data, advanced algorithmic techniques, and machine learning to present unprecedented visibility into the marginal carbon emission rate of the local electricity grid. The Marginal Operating Emissions Rate (MOER) is a quantification of the rate of emissions produced by the electricity generators in response to variations in the demand for energy on the local electrical grid, at a specific point in time. And MOER data from WattTime is in units of pounds of emissions per megawatt-hour (e.g. CO2 lbs/MWh). Quantification of emissions can be achieved through utilization of the marginal emissions rate MOER value by computing energy usage and multiplying that quantity by the MOER. The resulting quantity represents the CO2 emissions, which can be calculated as the sum of the product over the time-series. This method is often employed to assess the efficacy of load-shifting strategies aimed at mitigating greenhouse gas emissions.

4.4.2 Carbon Aware SDK

The Carbon-Aware SDK is a cutting-edge tool for building advanced software solutions that embody a carbon-conscious approach to energy consumption. By incorporating real-time carbon intensity data, applications can dynamically optimize energy utilization by selectively utilizing sources that are most environmentally-sustainable. This results in decreased carbon footprint and a more ecologically responsible approach to software development. With the Carbon-Aware SDK, one has the ability to create software that leverages dynamic meteorological data to choose the most favorable times for operation, such as when wind generation is at its peak. Many sources which provide same carbon efficient score reports data in lbCO2/kWh or gCO2/Wh and many other units. Carbon-Aware SDK tries to aggregate those data under one umbrella and report the data required by applications in gCO2/kWh.

4.5 Intelligent Node Selection

This section endeavors to delve into the intricacies of the algorithms of carbon-aware scheduling and geo-aware scheduling. The systematic demonstration of these algorithms will be furnished in the ensuing Section 4.5.1 and 4.5.2, conveyed via a pseudocode representation.

4.5.1 Carbon-Aware Node Selection

During scheduling cycle of GreenCourier, it runs through various piece of plugins which can be disabled through scheduler configuration. Following is the pseudocode representation carbon-aware logic implemented as part of research work as shown in Algorithm 1.

The scheduler implements a carbon-aware scheduling algorithm which leverages the carbon intensity values for a range of regions to perform an evaluation and ranking of said regions. The algorithm utilizes a multi-scheme evaluation process to determine the optimal region based on predefined criteria such as carbon intensity, though it may also incorporate additional parameters and weighting as needed. The algorithm is configured to prioritize the ranking according to carbon intensity, however, it offers the ability to modify its ranking criteria as needed to suit the specific requirements of the use case scenario. Once the most suitable region i.e., the greenest region in determined, the carbon-aware scheduler assigns pods to that region, until the cluster resource is exhausted or podSpec has restriction stopping from deploying in that region because of dependency or incompatibility. By default, WattTime is the source of carbon-aware scheduler, but it also supports Carbon Software Development Kit (SDK) from Green Software Foundation (GSF) at the moment.

In the GreenCourier Scheduler, multiple pre-integrated plugins can be utilized or disabled based on the scheduler configuration. The scheduler begins by collecting the list of enabled predicates (as shows in Section 2.2.2) from the configuration file and loading the relevant plugin code into cache. An empty array object, referred to as the "nodeList," is declared to store the nodes that meet all the conditions specified by the predicate plugins. The nodeList is then used to iterate through each node, evaluating it against every enabled predicate plugin to determine its suitability for pod scheduling. In essence, the process of node selection in a scheduling system entails the utilization of predicate plugins to perform a comprehensive assessment of each node. Any node that does not pass the evaluation criteria established by the enabled predicate plugins is excluded from further consideration. The resulting subset of nodes that have met all the predicates are then subjected to a ranking algorithm facilitated by score plugins, which assigns a score to each node based on specified metrics, resulting in a ranked list of nodes.

The GreenCourier scheduling system incorporates a filter phase that enforces stringent constraints on nodes or pod objects in the cluster, thereby eliminating non-compliant nodes from further consideration in the pod scheduling process (as explained in Section 2.2.2). The scoring phase employs a comparable methodology, executing an evaluation of nodes utilizing the active priority plugins, including the geo-aware plugin implemented for this study. The region of the node is determined by reading the annotations set by the administrator during cluster creation *line 3-4*, followed by the retrieval of real-time carbon efficiency scores from the metrics server *line 5*. The metrics server, developed as part of this research, provides a means of obtaining the carbon efficiency scores at 5-minute intervals. The scores are processed in a local cache within the program, using a simple map implementation (region as key and carbon score of that particular region as value) *line 6*, normalized to a specified ceiling, in our case the scores are normalised in range from 0 to 100 *line 8*, and the node with the highest score is selected *line 9*. The selected node's name is then assigned in the podSpec field and updated in the API server, ultimately being stored in the ETCD database *line 10-11*.

Algorithm 1 CarbonAware Scheduler

```
 1: Scoring Logic:
 2: function CALCULATE_SCORES(PodObject, ListOfNodeObjects)
 3:     for $Node \in ListOfNodes$ do
 4:         Region = Node.Annotation("region")
 5:         NodeScore = RetriveDataFromMetricsServer(Region)
 6:         UpdateAndStoreNodeScore(Region)
 7:     end for
 8:     NormaliseNodeScores()
 9:     Node = GetNodeWithHighestScore()
10:     AssignNodeForPod(PodObject, Node)
11: return PodObject
12: end function
```

4.5.2 Geo-Aware Node Selection

The Geo-Aware scheduling scheme as shown in Algorithm 2 was introduced in this research to evaluate its performance against the Carbon-Aware scheduler and to compare the results against the commonly used scheduling schemes for serverless functions, including the default scheduler code. The Geo-Aware scheduling scheme uses annotations created by the cluster administrator during cluster creation to determine the score of each node. These annotations are used to evaluate the nodes based on the distance between the management clusters and peer clusters. That distance is pre-calculated in case of GeoAware scheduler as this was only used for evaluation purposes, atleast in the scope of this thesis. This section provides a comprehensive examination of the workings of the Geo-Aware scheduler.

The scheduler begins by collecting the list of enabled predicates from the configuration file and loading the relevant plugin code into cache. An empty array object, referred to as the "nodeList," is declared to store the nodes that meet all the conditions specified by the predicate plugins. The nodeList is then used to iterate through each node, evaluating it against every enabled predicate plugin to determine its suitability for pod scheduling. In essence, the process of node selection in a scheduling system entails the utilization of predicate plugins to perform a comprehensive assessment of each node. Any node that does not pass the evaluation criteria established by the enabled predicate plugins is excluded from further consideration. The resulting subset of nodes that have met all the predicates are then subjected to a ranking algorithm facilitated by score plugins, which assigns a score to each node based on specified metrics, resulting in a ranked list of nodes. The region of the node under consideration is identified through the parsing of annotations set by the cluster administrator during the inception of the cluster creation process (*line 4-5*). A comprehensive score is then calculated based on the relative proximity of the node to the management plane. The score calculation methodology is contingent on the compilation of regions where the clusters are deployed, and is established through the evaluation of the relative distance between regions *line 6*. Node distance value is calculated by finding the geographical distance between two control plane region and region of that particular node. Since, GeoAware scheduler is used only for evaluation, pre-calculated values are hard-coded as part of code base. But it is easier to extend the logic of GeoAware scheduler to collect the data from metrics-server, as GreenCourier does. The score computation procedure is executed within the confines of the program's local cache storage mechanism (*line 7*) and is then normalized to an established ceiling (*line 9*). Subsequently, the node with the highest score is identified and its name is recorded in the podSpec field (*line 10*). The updated podSpec, inclusive of the assigned node name, is then transmitted to the API server for persistent storage in the ETCD database (*line 11-12*).

Algorithm 2 GeoAware Scheduler

```
1: Scoring Logic:
2: function CALCULATE_SCORES(PodObject, ListOfNodeObjects)
3:     for $Node \in ListOfNodes$ do
4:         CurrentRegion = Node.Annotation("region")
5:         NodeDistance = CalculateNodeDistance(CurrentRegion, ControlPlaneRegion)
6:         UpdateAndStoreNodeDistance(Region)
7:     end for
8:     NormaliseNodeScores(NodeDistance)
9:     Node = GetNodeWithHighestScore()
10:    AssignNodeForPod(PodObject, Node)
11: return PodObject
12: end function
```

5 Experiments and Evaluation

GreenCourier being developed for the goal of having a carbon-aware scheduling scheme for serverless functions, it is pertinent that we test the scheme against existing scheduling schemes which are used to schedule serverless in industry. Two of the scheduling scheme used are: GeoAware as mentioned in Section 4.5.2 and default scheduling scheme implemented by kubernetes [31]. Above mentioned evaluation should give us insights into performance of GreenCourier, both expectations and limitations.

5.1 Metrics and Experiments Configuration

5.1.1 Evaluation Metrics

The evaluation of GreenCourier aims to gauge its proficiency in mitigating the carbon footprint of cloud computing systems by means of a comparative analysis with GeoAware scheduling scheme and the conventional scheduling scheme. This evaluation will be based on three critical performance indicators, namely:

- Carbon Emission during function invocation.
- Placement of workloads in efficient region.
- Scheduling time and response time.

The results of the evaluation will provide insights into the carbon emission, placement efficiency, and scheduling and binding latency of GreenCourier compared to other scheduling schemes. The results will also help to identify areas for improvement and potential optimizations for GreenCourier.

Placement of Workloads in Efficient Region: This aspect evaluates the effectiveness of GreenCourier in placing pods in regions with efficient energy sources, compared to other scheduling schemes. Metrics used for this evaluation include the number of workloads placed in regions with clean energy sources, and the energy consumption of these regions compared to regions powered by non-renewable energy sources.

$$Placement\, Efficiency = \frac{no.\, function\ instances\ deployed\ in\ possible\ carbon\ efficient\ region}{total\ function\ instances} \tag{5.1}$$

Carbon Emission during Function Invocation: This aspect evaluates the carbon emissions generated during function invocations in the GreenCourier system compared to the other scheduling schemes. Metrics used for this evaluation is based on Green Software Foundation (GSF)'s Software Carbon Intensity specification project[89]. The methodology used for this evaluation will involve analyzing the energy consumption of the hardware and infrastructure used to run the functions, as well as the sources of energy used to power the hardware. The calculation of carbon emissions produced during function invocation can be achieved by multiplying the energy consumption, represented in kilowatt hours (kWh), and the carbon intensity value obtained from a carbon intensity specification, along with the Marginal Operating Emissions Rate (MOER) value obtained from WattTime. Since the functions can be deployed at various regions, we can approximate the MOER value by taking weighted average of number of instances deployed in particular region.

$$AverageMOER = \frac{\sum_{i=1}^{n} number\,of\,instance\,in\,a\,region\;*\;MOER}{\sum_{i=1}^{n} number\,of\,instance\,in\,a\,region} \tag{5.2}$$

We can then use the calculated average MOER value and plug it in Software Carbon Intensity (SCI) formula:

$$SCI = ((E * I) + M)/R \tag{5.3}$$

Where:

- (E) - Energy consumption of software.
- (I) - Marginal Emissions factors.
- (M) - Embodied emissions.
- (R) - The functional unit.

By incorporating the computed Average Marginal Carbon Intensity (MOER) value as I in Equation 5.3, while disregarding the Embodied Emissions as they remain unaffected by the implementation of GreenCourier, the equation can be succinctly formulated as follows:

$$SCI = (E * AverageMOER)/R \tag{5.4}$$

To calculate the energy consumed by a machine, we will consider a example - A one core virtual machine, which uses Intel Skylake i9-7980XE processor with clock rate of 2.4 GHz, has Thermal Design Power (TDP) of 165 W. Assuming the 50% of the core is used at any point, energy consumed by the processor for 24 hours is:

$$Energy = (TDP\;*\;Load\;*\;Time) = 165 * 50\% * 24 = 1.98kWh$$

Scheduling time and response time: This aspect evaluates the latency of the scheduling and binding process in the GreenCourier system compared to the other scheduling schemes.

Metrics used for this evaluation include the time taken to schedule and bind a task, and the impact of this latency on the overall completion time of the task. The methodology used for this evaluation will involve analyzing the time taken for the scheduling and binding process and comparing it to the other scheduling schemes. Here we split the scheduling time into scheduling latency taken by scheduler to assign node to a pod and binding latency taken by other components like Kubelet [87] (in normal setup) and Liqo [70] (in a geographically distributed multi-cluster setup) to actually deploy the pod and sync the status to control plane. This is accomplished by monitoring events (as described in Section 2.2.1.2) emitted by the Kubernetes cluster whenever there is a modification in the state of the workload or pod. The metric of response time holds paramount importance in the realm of serverless deployment as it quantifies the amount of time it takes for the server to process and respond to a client's request. A low response time is indicative of a highly performant system, which is imperative for delivering a superior user experience and fostering user engagement. Within a serverless deployment, the response time metric can play a crucial role in determining the overall system performance, and a slow response time can have a detrimental effect on the user experience and result in decreased business value. Hence, the monitoring and optimization of response time is crucial for ensuring the viability and sustainability of a serverless deployment.

5.1.2 Benchmark and Functions

For the purpose of benchmark, we use k6 [108], a open-source project driven by grafana labs. Grafana k6 is a free and open-source load testing tool designed for engineering teams. It prioritizes ease-of-use and productivity, making performance testing accessible. It is developer-focused and can be easily customized to meet specific requirements. We utilize functions of varying characteristics but with a relatively short execution time, which can be considered as having a temporal life. List of all the functions and a short description about them can be found in Table 5.2.

Cluster Type	Region	Instance name	# of instances	# of vCPUs	RAM
Management	europe-west3-a	e2-standard-16	1	1×16	$1 \times 64GB$
Peer	europe-southwest1-a	e2-standard-4	4	4×4	$4 \times 16GB$
	europe-west9-a	e2-standard-4	4	4×4	$4 \times 16GB$
	europe-west1-b	e2-standard-4	4	4×4	$4 \times 16GB$
	europe-west4-a	e2-standard-4	4	4×4	$4 \times 16GB$

Table 5.1: Cluster setup used while benchmarking GreenCourier

5.1.3 Experiment setup

To effectively evaluate GreenCourier under heavy loads and benchmark its performance, a multi-cluster setup was established using 80 vCPUs and 320 GB of RAM resources. Cluster setup included a management cluster in Frankfurt region (europe-west3-a) in Google Cloud Platform (GCP). Other than the management cluster, 4 other peer clusters were deployed in geographically distributed manner as shown in Table 5.1. Four peer clusters were deployed in following regions: europe-southwest1-a (Madrid, Spain), europe-west9-a (Paris, France), europe-west1-b (St. Ghislain, Belgium), europe-west4-a (Eemshaven, Netherlands).

As previously articulated, the load testing tool employed was Grafana k6. The test suite arrangement for load testing was established as 110 Virtual Users (VUs) with a duration of 5 minutes, thereby translating to a concurrent volume of 110 parallel requests per second over a span of 300 seconds. This experiment was conducted with a repeated iteration for each scheduling setup and each of the 8 distinct functions as outlined in Table 5.2. This resulted in a comprehensive evaluation of GreenCourier through 24 distinct scenarios, comprising of the 3 scheduling configurations, i.e., GreenCourier, GeoAware, and the default scheduler schemes serving as the baseline for performance comparison.

Function Name	**Description**
CNN-Serving	SqeezeNet [109] CNN based function which uses ImageNet[110] dataset to train the model and serve the request
Float operations	Simple function which executes simple math and trigonometric operations on set of data
LR-Serving	A Linear regression model which is used to predit the TD-IDF [111] features of a document.
LinPack	Function that used linalg library [112] to solve set of linear matrix equation
Matrix Multiplication	Funtion implementation of matrix multiplication using NumPy library [113].
Pyaes	Simple function which uses Pyaes [114] library to encrypt data
RNN-Serving	Pytorch [115] based Recurrent Neural Networks implementation of simple language model
Chameleon	An XML/HTML rendering function using chameleon [116] library.

Table 5.2: Short description on functions used for benchmarking GreenCourier

5.2 Pod Placement Efficiency

The Pod placement efficiency metric is employed to evaluate the ability of various scheduling schemes to place pods in regions with optimal carbon efficiency. Although the default scheme and the GeoAware scheme do not distinguish between clusters based on their carbon efficiency, they may occasionally deploy pods to high-efficiency clusters due to the influence of the "PodSpreadTopology" (as described in Section 2.2.2) plugin in the default scheme or resource availability in the GeoAware scheme. The results, as illustrated in Figure 5.1, demonstrate that the CarbonAware scheduler outperforms the other two scheduling schemes by significant margins.

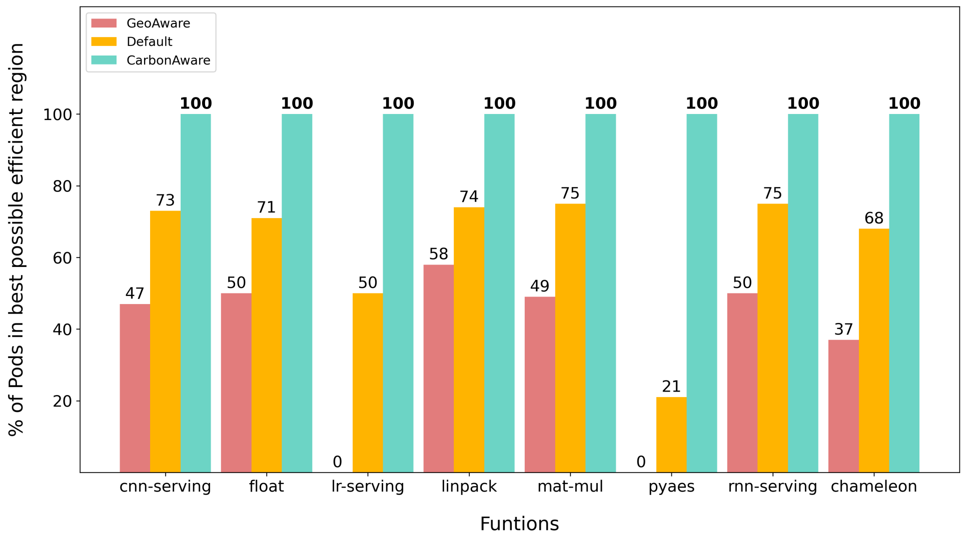

Figure 5.1: Pod placement efficiency comparison between GeoAware scheduler 4.5.2, default scheduler [61] and GreenCourier

The GreenCourier scheduling scheme exhibits a substantial improvement in the proficiency of identifying and positioning pods in regions that exhibit superior carbon efficiency. This is evidenced by the average improvement of 36.6% when compared to the conventional default scheduling approach. Conversely, the GeoAware scheduling scheme exhibits an adverse performance in the placement of pods in carbon-efficient regions. This inferiority is particularly pronounced when the number of instances is comparatively low(lr-serving, pyaes). This can be attributed to the priority given by the GeoAware scheduler to regions such as europe-west1-b and europe-west4-a, as indicated in Table 5.3. However, it is worth noting that these regions exhibit low carbon scores when compared to europe-southwest1-a and europe-west9-a, which in turn contributes to the inadequate performance of the GeoAware scheduling scheme in the pod placement metric. In quantifiable terms, the GeoAware scheduler demonstrates a 63.7% deficiency in comparison to the CarbonAware scheduler in terms of accurately identifying environmentally friendly regions for deploying pods. This information further highlights the superiority of the default scheduling scheme over the GeoAware scheme. As previously mentioned, the default scheme focuses on maximizing availability by spreading pods throughout the setup, leading to a certain number of pods being deployed in environmentally conscious regions. However, it does not maximize the number of pods in those regions. To put it in concrete terms, the default scheduling approach performs 50% better in terms of placing pods in environmentally conscious regions within a geographically distributed multi-cluster topology setup when compared to the GeoAware scheduling approach.

5.3 Scheduling time and Response Time

This section commences the evaluation process by contrasting the time taken during scheduling time and response time performance of three different scheduling configurations.

5.3.1 Scheduling Latency

As discussed previously, the scheduling time can be split into two sub-metrics - scheduling latency and binding latency. Scheduling latency is the time taken for a scheduler to evaluate every nodes in a cluster during pod scheduling and assign a node to the pod, it is pictorially depicted as highlighted white area shown in Figure 2.3. This was calculated by listening to cluster events, by finding difference between timestamp associated with pod creation event and node assignment event emitted by replicaset controller and scheduler respectively. Figure 5.2 depicts the scheduling latency between two schemes - GreenCourier and default. The chart shows a box plot comparison of scheduling latency metrics between GreenCourier and default implementation of Kubernetes scheduler. In the plot, we draw box from 25th percentile or 1st quartile (minimum) to 75th percentile or 3rd quartile (maximum). Line in the middle signifies median value. And the whiskers represent minimum and maximum value. The comparison between GreenCourier and the default scheduler is conducted as the baseline, and not with the GeoAware scheme, as the performance of the scheduler component in the GreenCourier system is almost similar to that of the traditional scheduling scheme. Moreover, the implementation of the GeoAware scheduling scheme is also closely aligned to that of the GreenCourier, thus rendering the inclusion of an additional baseline with GeoAware unnecessary. To put it in more concrete terms, when comparing the performance of GreenCourier with the default scheme, it was observed that the average time taken by GreenCourier was 539 ms, which is a mere 24 ms more than the average time taken by the default scheduler, which was 515 ms. It can be noted that when the overall scheduling latency is taken into consideration, both the GreenCourier and default scheduler schemes exhibit similar behavior in terms of their minimum and maximum values, as well as the 25th and 75th percentile values as shown in Figure 5.2.

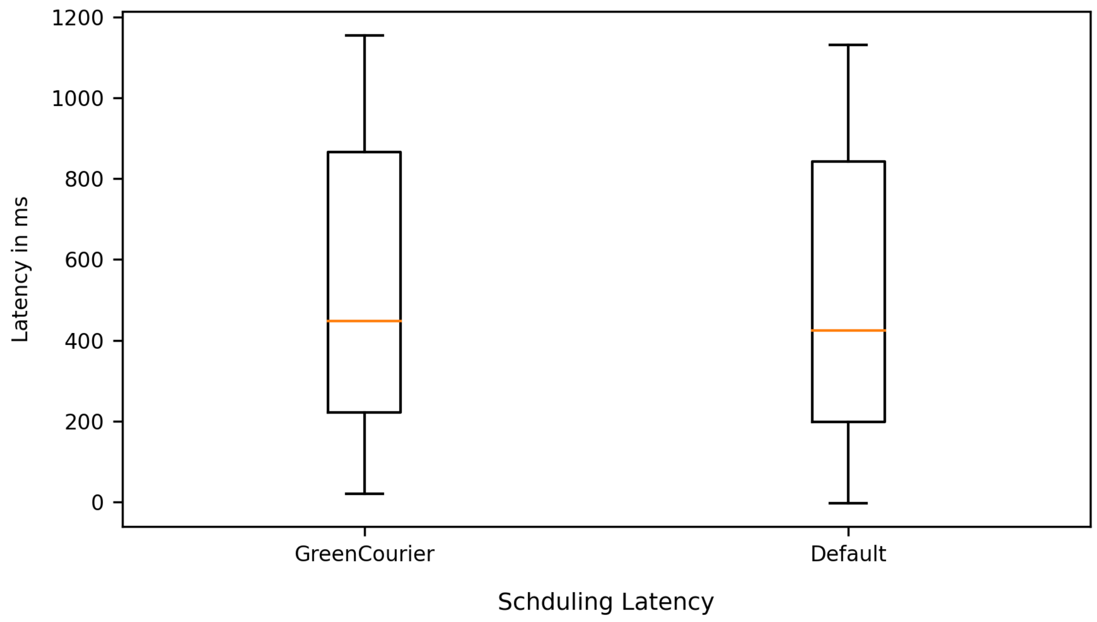

Figure 5.2: Scheduling latency comparison between GreenCourier and Default Scheduler

5.3.2 Binding Latency

Binding latency is the time taken from node assignment by scheduler to pod get started and status of the pod is updated to running state in Kubernetes. Binding Latency is calculated similarly as scheduling latency, which is by listening to cluster events. Especially node assignment event from scheduler and pod running event from kubelet or Liqo depending on the case, it is pictorially depicted as highlighted yellow area shown in Figure 2.3. In evaluating the binding latency of the GreenCourier system, we compare it to the conventional setup that uses a single cluster with multiple worker nodes. In the conventional setup, the kubelet [87] component is responsible for deploying pods in worker nodes and updating their status in the control plane. However, in the GreenCourier system that employs Liqo for multi-cluster topology setup, the virtual kubelet [86] component of Liqo is used to masquerade a cluster as a node from the viewpoint of the management cluster. This leads to an additional layer of synchronization in the GreenCourier system, as the status of the peer cluster is sent to the management cluster through the virtual kubelet and Liqo, while in the conventional setup, there is no need for this extra synchronisation step as there is no overhead associated with maintaining a multi-cluster topology.

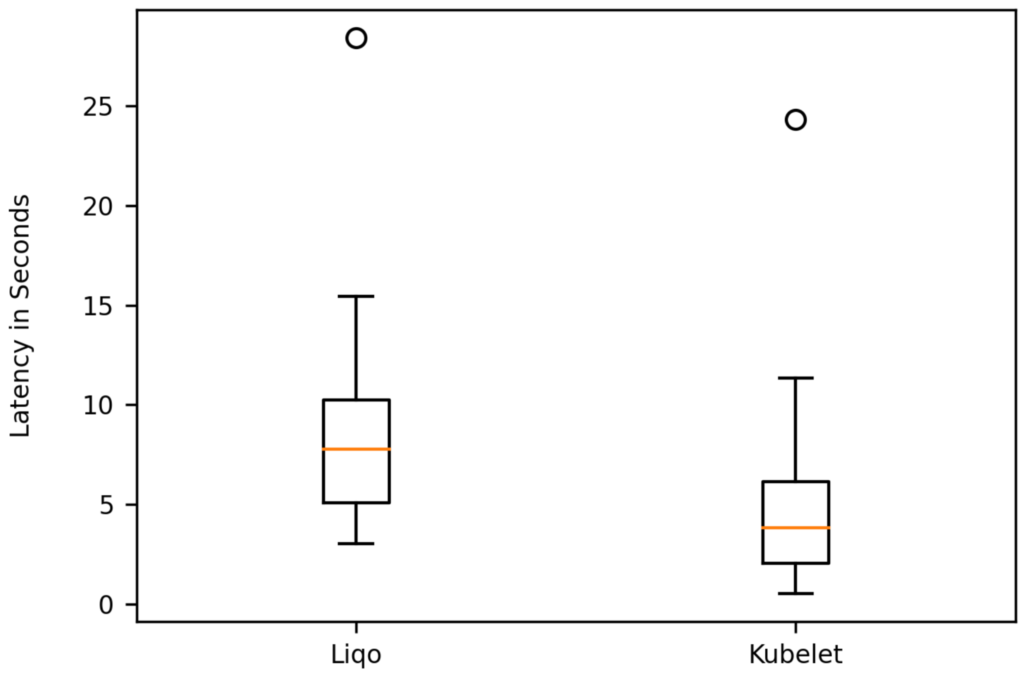

Figure 5.3: Binding latency comparison between Liqo and Kubelet

In this evaluation, we compare the binding latency of the GreenCourier system, which is based on the multi-cluster topology facilitated by Liqo, against the conventional default setup of a single cluster with multiple worker nodes. The conventional setup involves kubelet, which is responsible for picking up the pod and deploying it on a worker node and then synchronizing the status back to the control plane. However, in the multi-cluster topology of GreenCourier, the virtual kubelet of Liqo acts as an intermediary between the peer clusters and the management cluster, thereby adding an additional layer of synchronization.

As anticipated, the binding latency of Liqo surpasses that of kubelet, as illustrated in Figure 5.3. The chart shows a box plot comparison of scheduling latency metrics between GreenCourier and default implementation of Kubernetes scheduler. In the plot, we draw box from 25th percentile or 1st quartile (minimum) to 75th percentile or 3rd quartile (maximum). Line in the middle signifies median value. And the whiskers represent minimum and maximum value. And the circles represent outlier values, which is one of the data points collected, which is significantly different than the rest of the values observed during evaluation. The GreenCourier system, by its nature, offloads pods to the more carbon-efficient or "green" clusters, putting the control-plane components of the peer clusters under stress. Meanwhile, the other clusters remain unscathed from the sudden increase in load.

In comparison, the default scheduler assigns pods using the "PodTopologySpread" scheme, which strives to evenly distribute pods among worker nodes or peer clusters in the context of Liqo. As a result, the kubelet on the worker node is not put under the same pressure to ensure the creation, starting, and checking of the liveness probe of the pods, when compared to the GreenCourier system. The latter prioritizes scheduling as many pods as possible in a single peer cluster in a carbon-efficient region, which exacerbates the difference in binding state delay.

Furthermore, the introduction of a supplementary cycle of inter-cluster synchronization in the GreenCourier system exacerbates the overhead incurred. This overhead stems from the inherent need for managing a multi-cluster topology, in order to leverage the carbon efficiency of specific regions, in the GreenCourier setup. The augmenting of resource abstraction always incurs latency, owing to the additional level of synchronization required to be performed. There is a potential additional latency introduced due to the inter-regional communication transpiring between the geographically dispersed clusters, as all inter-cluster communication must traverse the public internet, unlike the intra-VPC communication in a conventional setup. This is likely to have a substantial impact on the binding latency of Liqo.

The mean binding latency of Liqo in the multi-cluster topology setup is observed to be approximately 8.28 seconds, which is substantially higher than that of the conventional single cluster setup utilizing kubelet, which exhibits an average binding latency of 4.53 seconds as shows in Figure 5.3, when the system is under stress. This disparity can be attributed to the increased complexity introduced by the utilization of Liqo, which necessitates inter-region communication between geographically dispersed clusters, thereby increasing the binding latency by the added overhead from public internet communication, in comparison to communication within a VPC as is the case in the conventional setup.

5.3.3 Response Time

Although the primary objective of this research is to optimize Carbon efficiency, a thorough analysis of any potential side effects associated with the proposed scheduling scheme is essential. From the user's perspective, it is imperative to ensure that the usage experience remains uncompromised, i.e. without any adverse impact on latency and service quality. The response time metric results obtained from the load testing are depicted in Table 5.4. An examination of the results highlights that the CarbonAware scheme utilized in the GreenCourier system has the lowest performance, followed by the default scheme. On the other hand, the GeoAware scheme exhibits the most optimal response time when compared to the other two schemes, as anticipated. This section will delve into the reasons behind the superior performance of the GeoAware scheme and the underperformance of the CarbonAware scheme.

Region	Carbon Score	GeoScore
europe-southwest1-a	**100**	15
europe-west9-a	**98**	71
europe-west1-b	71	**100**
europe-west4-a	56	**97**

Table 5.3: Scores assigned by Carbon plugin and Geo plugin during load test

Function Name	Scheduling Scheme	Response Time in ms			
		Average	Median	p(90)	p(95)
CNN-Serving	Default	195.77	106.31	267.18	362.41
	GeoAware	190.45	96.41	321.81	621.37
	CarbonAware	212.58	111.04	309.76	465.38
Float	Default	1310	996.48	2050	2530
	GeoAware	1250	1070	2110	2590
	CarbonAware	1470	1420	2370	2790
LR-Serving	Default	24.7	15.12	31.78	56.86
	GeoAware	22.88	13.67	34.39	36.11
	CarbonAware	26.78	33.5	41.97	54.33
LinPack	Default	942.04	736.8	1510	1800
	GeoAware	866.53	650.07	1300	1550
	CarbonAware	1010	738.54	1450	1760
Matrix Multiplication	Default	463.48	386.05	835.62	1000
	GeoAware	425.83	378.92	730.41	845.23
	CarbonAware	513.4	315.53	723.63	916.79
Pyaes - Block Cipher	Default	19.38	15.07	23.12	35.44
	GeoAware	19.13	14.45	35.25	36.3
	CarbonAware	21.04	26.46	42.13	47.03
RNN-Serving	Default	48.27	46.4	65.84	79.11
	GeoAware	47.39	42.84	67.65	87.85
	CarbonAware	55.45	48.97	86.21	120.97
Chameleon - HTML/XML Template Engine	Default	53.19	40.79	102.76	127.25
	GeoAware	48.94	42.14	62.21	85.62
	CarbonAware	60.69	51.7	87.33	136.85

Table 5.4: Response time of scheduling schemes during load test

As noted, the four regions were selected based on their carbon efficiency and geographical scores, with two regions exhibiting the highest carbon efficiency scores and the remaining two regions showcasing the highest geographical scores. It is imperative to mention that the carbon efficiency scores are subject to alteration every 5 minutes, however, the geographical scores remain constant until a new node region is added or an existing node region is removed from the scheduling process.

Table 5.4 presents statistical data from a load test conducted on the system, including average, median, 90th percentile and 95th percentile values. As demonstrated in the Table 5.4, the CarbonAware scheme exhibits a slower response rate, with an geometric mean of 10.26% reduction compared to the default scheme. This degradation in performance can be attributed to the selection of green regions, which are the farthest regions in our test setup, as indicated in Table 5.3 (europe-southwest1-a and europe-west9-a regions). The low geo score of these regions suggests that the management cluster is situated at a relatively farther distance from the peer cluster, compared to other peer clusters in the registered regions. The GeoAware system demonstrates superior performance compared to the default scheme, as demonstrated by a 4.2% improvement in response rate. This is because the deployment of workloads in regions closest to the management cluster contributes to better performance. When GeoAware performs better than the default scheme, it can be inferred that it should also perform better than the CarbonAware scheme. The data in Table 5.4 indicates that GeoAware exhibits a 16.24% improvement in response rate compared to the CarbonAware scheme. Hence, it can be concluded that the GeoAware system delivers a better performance in terms of response rate.

5.4 Carbon Emission

Despite the remarkable efficiency demonstrated by GreenCourier in identifying and deploying pods in carbon-efficient regions, it is crucial to delve deeper into the actual carbon emissions generated when a particular scheduling scheme is employed. The evaluation can be performed utilizing the modified Software Carbon Intensity (SCI) metric proposed by Green Software Foundation, as demonstrated in Equation 5.4.

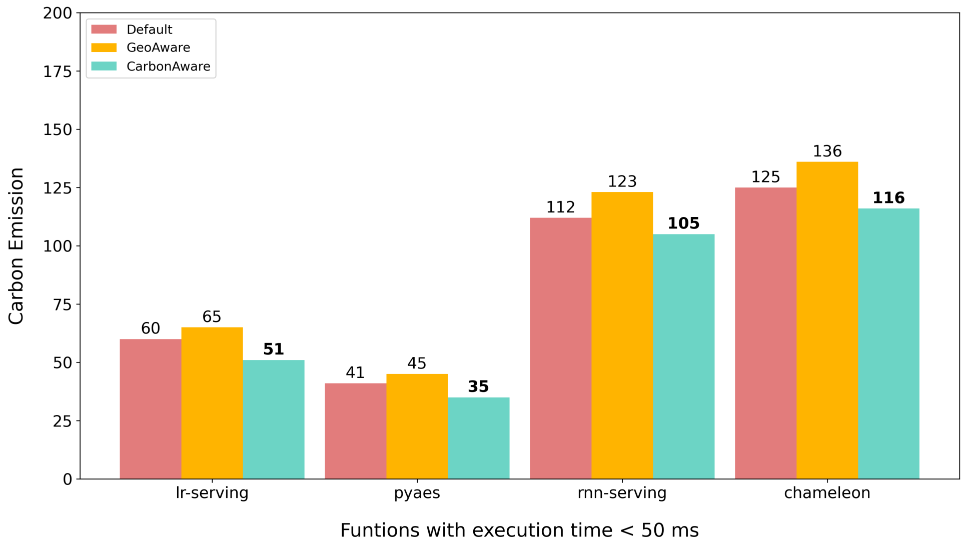

Figure 5.4: Carbon emissions comparison between default scheduler [61], GeoAware scheduler 4.5.2 and GreenCourier for functions with execution time < 50 ms

In accordance with Section 5.1.1, the average Metric of Energy and Resource (MOER) value, as well as the energy consumption by the cluster resources can be calculated. Subsequently, the carbon intensity can be estimated in gCO_2eq (grams of carbon equivalent). The carbon emission comparisons between the three scheduling schemes are demonstrated in Figures 5.4 and 5.5. It is imperative to note that due to the significant scale difference in execution time between the listed 8 functions, the illustration of comparison has been provided in two separate figures for the convenience of visualization.

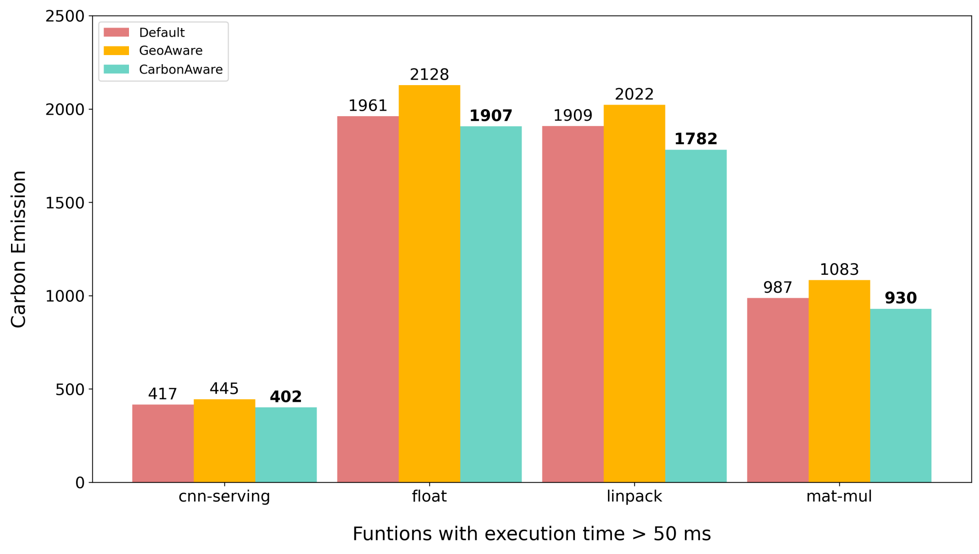

Figure 5.5: Carbon emission comparison between default scheduler [61], GeoAware scheduler 4.5.2 and GreenCourier for functions with execution time > 50 ms

After conducting thorough evaluations, it was determined that the GreenCourier scheduling scheme outperforms both the default and GeoAware scheduling schemes in terms of carbon emission reduction. GreenCourier demonstrated 8.7% and 17.8% lower carbon emission levels compared to the default and GeoAware schemes per function execution, respectively. This disparity can be attributed to the fact that the GeoAware scheme tends to place pods in regions with comparatively low carbon efficiency, resulting in a higher carbon footprint. These results, represented in Figure 5.4 and 5.5, are expressed in μg and can be scaled to accommodate multiple requests by multiplying the value by the number of function invocations. In our evaluation, only EU regions were taken into consideration. As shown in Table 5.3, all the regions selected were having carbon scores of more than 50, which implies the regions selected were having comparatively better scores than the regions which were left out. This does significantly influences our results. If regions with lower carbon scores or if there were high difference between scores of the best and the worst regions, results would have shown better performance numbers in terms of carbon emissions, i.e., the pod placement efficiency numbers would have translated in much better performance for GreenCourier. In conclusion, evaluation results shows that by using GreenCourier's scheduling scheme, 9% of carbon emission was reduced per function invocation.

5.5 Discussion

GreenCourier's unique placement strategy of considering carbon efficiency while deploying pods sets it apart from the other two schemes. This results in lower carbon emissions compared to default and GeoAware, as illustrated in the previous figures and analysis. However, it should be noted that each scheduling scheme has its own strengths and weaknesses, and the ideal choice of a scheduler would depend on the specific requirements and priorities of the user. Additionally, it is important to consider the trade-offs and limitations of each scheme while making a decision.

The operational reliability of the three scheduler implementations is equivalent, as they are all founded on the underlying core scheduler code and necessitate similar computational resources to execute within the cluster environment. However, a drawback of the GreenCourier implementation is its reliance on a metrics server to collect carbon score data, unlike the other two implementations which do not have any additional dependencies beyond the cluster control plane. Despite this external dependency, the communication overhead between the GreenCourier scheduler and metrics server is minimal and does not negatively impact the scheduling latency. In the event of metrics server unavailability, the GreenCourier scheduler will rely on cached, albeit potentially expired, carbon score data for real-time scheduling. This may result in suboptimal "green" region scheduling, yet the functionality for node identification and pod placement remains intact. In the event of an inability to retrieve a carbon score data update even for a single round of scheduling, GreenCourier will operate like the standard scheduler, meaning it will still be able to allocate pods to nodes in the Kubernetes cluster, but will not engage in "green" region-based intelligent scheduling.

In order to make use of electricity produced by renewable or low-carbon emission sources, it is crucial to have a geographically distributed multi-cluster setup that can make real-time decisions on function deployment based on the cleanliness of energy at a given time. Currently, none of the cloud providers allow for the creation of a geographically distributed Kubernetes cluster in different parts of the world, as they only allow nodes of a cluster to be in one region. Thus, the topology of creating a geographically distributed multi-cluster is critical. However, due to the extra layer of abstraction involved, there may be a performance drop in terms of longer time to synchronize the status of all clusters with the management cluster, which was 82% (as shown in Section 5.3.2) more than the conventional cluster setup. This is due to the added layer of abstraction in the form of a geographically distributed multi-cluster setup. The extra time taken in the binding phase can be attributed to the added complexity of synchronizing the status of multiple clusters with the management cluster, which is a necessary step in this setup. The higher binding latency indicates the trade-off that is made between reduced carbon emission and increased latency in the binding phase. However, it should be noted that despite this increased latency, the overall reduction in carbon emission achieved through the use of GreenCourier is still significant. Additionally, improving the efficiency and speed of the Liqo component could result in better performance in terms of reduced binding latency and improved overall performance of the multi-cluster setup. Furthermore, investigating and implementing strategies to increase concurrency in Liqo could potentially mitigate the drawbacks of the geographically distributed multi-cluster setup and enhance the system's ability to effectively synchronize the status of all clusters. As such, future research efforts should focus on investigating and optimizing the performance of Liqo to further improve the functionality and efficiency of the multi-cluster setup.

Our GreenCourier scheduling approach was subjected to a comprehensive evaluation within a multi-cluster, geographically dispersed setup, comprising one central management/-consumer cluster with 16 vCPUs and 64 GB RAM and 4 peer/provider clusters located in different regions, each with 4 homogeneous nodes of e2-standard-4 VMs (4 vCPUs, 16 GB RAM). The total cluster capacity was 80 vCPUs and 320 GB RAM, which is a representative estimation of typical production-level clusters in the industry. The load testing tool employed was k6, and the results, as depicted in Table 5.4, indicated that functions deployed through GreenCourier, which leverages the CarbonAware scheduling strategy, exhibited the highest response time compared to the default strategy and the GeoAware strategy by a margin of 10% and 16% respectively. Additionally, it is noteworthy that the results highlight the need for further optimization of the GreenCourier approach, particularly with respect to the CarbonAware scheme to reduce the response time of functions. GeoAware scheduling scheme does have significant advantage over other two scheduling schemes in this aspect.

The use of a more efficient scheduling algorithm that takes into account not only carbon efficiency, but also geographical proximity performance of GreenCourier. The GreenCourier scheduling scheme was developed with the primary goal of reducing carbon emissions into the environment. While it was theoretically anticipated that the response time would take a hit with the implementation of this scheme. As expected, the results of the evaluation showed that the response time was worse than the default strategy and the GeoAware strategy. However, it is important to note that the primary motivation of this research was not to improve response time but to minimize carbon emissions. Improvements to the network address translation scheme adopted by Liqo could potentially reduce the effect on response time, but the current scheme is not suspected to be the bottleneck. Nevertheless, users should be aware that the primary focus of GreenCourier is reducing carbon emissions, and the trade-off is a potentially slower response time.

Another limitation of the GeoAware and GreenCourier scheduling schemes is that they tend to concentrate the workload on certain nodes, leading to imbalanced load distribution. This behavior is designed to maximize performance (e.g. low response time or reduced carbon emissions), but it also increases the likelihood of node failure and subsequent performance drop until the cluster returns to its desired state. In contrast, the default scheduling scheme spreads the deployment of functions evenly across all nodes, reducing the risk of performance drop during node failure. However, it should be noted that node failure is a possibility in the default scheme as well. The choice of scheduling scheme ultimately depends on the desired trade-off between performance and stability. Based on the results of the evaluation, it appears that the default scheduling scheme has an advantage over the GeoAware and GreenCourier schemes. The default scheme provides a more balanced workload distribution, reducing the risk of node failure and subsequent performance drop. It is important to note that the choice of scheduling scheme ultimately depends on the specific requirements and constraints of each use case.

The GreenCourier scheme successfully achieves its goal of reducing carbon emissions through intelligent function deployment in selected regions. In the evaluated test setup, the GreenCourier scheme resulted in a 36.6% and 63.7% better placement of pods in environmentally-sustainable regions (as shown in Section 5.2) compared to the default and GeoAware schemes, respectively. These reductions translated to an 8.7% and 17.8% (as shown in Section 5.4) reduction in grams of CO_2 equivalent gCO_2eq compared to the default and GeoAware schemes.

6 Conclusion and Future Work

The objective of this study is to devise and assess a new scheduling framework for serverless functions that can efficiently allocate these functions to environmentally sustainable regions, reducing the carbon footprint during function execution. In this regard, a metrics server was developed to collect Marginal Operating Emissions Rate (MOER) data from various sources, including WattTime and Carbon-aware SDK provided by Green Software Foundation. The performance of the GreenCourier scheduling framework was then extensively evaluated against Kubernetes default implementation and geo-aware scheduling scheme. The findings of the study demonstrate that GreenCourier outperforms the other two scheduling schemes in terms of pod placement in environmentally sustainable regions and carbon emissions during function execution. However, it lags in terms of response time, indicating the need for further research to address this issue.

GreenCourier can be further improved by adding High Availability (HA) capability to its deployment. Although Kubernetes does provide the ability to deploy a scheduler in HA mode, it cannot ensure consistency between different instances due to the external dependency of GreenCourier on the metrics server. Another potential enhancement to GreenCourier is the implementation and exposure of customized metrics related to its performance and scheduler state. This will reduce the need for manual intervention during evaluation and enhance its monitoring capabilities.

In light of the results of the evaluation, it is clear that the GreenCourier scheduling scheme has certain limitations that should be taken into consideration. These limitations include a potentially slower response time compared to the default and GeoAware schemes, as well as a higher risk of node failure due to imbalanced workload distribution. To overcome these drawbacks, significant time and resources should be invested in finding ways to mitigate these limitations and improve the overall performance of the GreenCourier scheme. Improvements to the network address translation scheme adopted by Liqo could potentially reduce the effect on response time, but the current scheme is not suspected to be the bottleneck. Future research should explore alternative network translation schemes to determine if they have a positive impact on response time.

Another area of performance degradation is observed during the binding latency of pods, which involves status synchronization of offloaded pods between the target and management clusters. To address this limitation, significant effort should be directed towards parallelizing the status synchronization at the Liqo level, as the current network bandwidth between the two clusters is not fully utilized. This will help mitigate the loss of response time performance.

In conclusion, serverless computing is a rapidly developing field that requires considerable investment of time and resources to create innovative advancements. We believe that our work on developing and implementing a new carbon-aware scheduling framework will inspire further research in the area of sustainability in serverless computing.

List of Figures

List of Tables

Acronyms

Amazon EMR Amazon Elastic MapReduce. 9

API Application Programming Interface. 9–12, 17, 24, 28

AWS Amazon Web Services. 2, 6, 8

CNI Container Network Interface. 22, 23, 26

DNS Domain Name Service. 12

DVFS Dynamic Voltage Frequency Scaling. 35

FaaS Function-as-a-Service. 1, 2, 7–9, 18, 31, 37

FCFS First Come First Serve. 34

GCP Google Cloud Platform. 1, 6, 51

GPU Graphics Processor Unit. 8

GSF Green Software Foundation. v, 27, 44, 45, 50, 59, 64

HA High Availability. 64

HTML HyperText Markup Language. 52

HTTP Hypertext Transfer Protocol. 9

IaaS Infrastructure-as-a-Service. 1, 8

IP Internet Protocol. 22, 23, 26, 40

IPsec Internet Protocol Security. 22

JSON JavaScript Object Notation. 3

ML Machine Learning. 3, 4

MOER Marginal Operating Emissions Rate. 3, 38, 44, 45, 50, 64

MOOP Multi-Objective Optimization Problem. 34

NAT Network Address Translation. 26, 38

OS Operating System. 31

PaaS Platform-as-a-Service. 29

QoS Quality of service. 29

RAM Random Access Memory. 51

RBAC Role-based access control. 43

RNN Recurrent Neural Networks. 52

SCI Software Carbon Intensity. 50, 59

SDK Software Development Kit. iv, 45, 64

TCP Transmission Control Protocol. 11

TDP Thermal Design Power. 50

TLS Transport Layer Protocol. 22

UDP User Datagram Protocol. 11

VPC Virtual Private Cloud. 57

XML Extensible Markup Language. 52

Bibliography

[1] A. S. G. Andrae and T. Edler. "On Global Electricity Usage of Communication Technology: Trends to 2030". In: *Challenges* 6.1 (2015), pp. 117–157. ISSN: 2078-1547. DOI: 10.3390/challe6010117. URL: https://www.mdpi.com/2078-1547/6/1/117.

[2] e. P.R. Shukla J. Skea. "Climate Change 2022: Mitigation of Climate Change". In: *Cambridge University Press, Cambridge, UK and New York, NY, USA.* (2022). DOI: 10.1017/9781009157926. URL: https://report.ipcc.ch/ar6wg3/pdf/IPCC_AR6_WGIII_FinalDraft_FullReport.pdf.

[3] *The Intergovernmental Panel on Climate Change*. https://www.ipcc.ch/. Accessed: 2022-06-01.

[4] M. Dayarathna, Y. Wen, and R. Fan. "Data Center Energy Consumption Modeling: A Survey". In: *IEEE Communications Surveys and Tutorials* 18.1 (2016), pp. 732–794. DOI: 10.1109/COMST.2015.2481183.

[5] *Data centers are more energy efficient than ever*. https://blog.google/outreach-initiatives/sustainability/data-centers-energy-efficient. Accessed: 2022-06-01.

[6] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson. "What Serverless Computing is and Should Become: The next Phase of Cloud Computing". In: *Commun. ACM* 64.5 (Apr. 2021), pp. 76–84. ISSN: 0001-0782. DOI: 10.1145/3406011. URL: https://doi.org/10.1145/3406011.

[7] *Tracking Google's carbon-free energy progress*. https://sustainability.google/progress/energy/. Accessed: 2022-06-01.

[8] *Environmental sustainability - Microsoft*. https://www.microsoft.com/en-us/corporate-responsibility/sustainability. Accessed: 2022-06-01.

[9] *Sustainability - Oracle Cloud*. https://www.oracle.com/corporate/citizenship/sustainability/clean-cloud.html. Accessed: 2022-06-01.

[10] *How much energy do data centers use?* https://davidmytton.blog/how-much-energy-do-data-centers-use/. Accessed: 2022-06-01.

[11] *GreenCourier: Towards sustainable Serverless Computing*. https://taikai.network/gsf/hackathons/carbonhack22/projects/cl8om5s2c6222501xclyvpc9h0/idea. Accessed: 2023-02-05.

[12] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. "The Rise of Serverless Computing". In: (2019). DOI: 10.1145/3368454. URL: https://doi.org/10.1145/3368454.

[13] S. Allen, C. Aniszczyk, C. Arimura, B. Browning, L. Calcote, A. Chaudhry, A. G. Doug Davis Louis Fourie, Y. Haviv, D. Krook, O. Nissan-Messing, C. Munns, K. Owens, M. Peek, and C. Zhang. *CNCF Serverless Whitepaper v1.0*. https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf. Accessed: 2023-02-05. 2022.

[14] V. Ishakian, V. Muthusamy, and A. Slominski. *Serving deep learning models in a serverless platform*. DOI: 10.48550/ARXIV.1710.08460. URL: https://arxiv.org/abs/1710.08460.

[15] *Global Serverless Architecture Market To Reach USD 86.94 Billion By 2030*. https://www.reportsanddata.com/press-release/global-serverless-architecture-market. Accessed: 2023-02-05.

[16] *GCP - Cloud Functions*. https://cloud.google.com/functions/. Accessed: 2022-06-01.

[17] *AWS Functions*. https://aws.amazon.com/lambda/. Accessed: 2022-06-01.

[18] *Azure Functions*. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2022-06-01.

[19] *Kubernetes - Production-Grade container Orchestration*. https://kubernetes.io/. Accessed: 2023-02-05.

[20] *Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications*. https://knative.dev/docs/. Accessed: 2023-01-31.

[21] J. Kim and K. Lee. "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 502–504. DOI: 10.1109/CLOUD.2019.00091.

[22] *About Knative Services*. https://knative.dev/docs/serving/services/. Accessed: 2022-06-12.

[23] *ReplicaSet | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. Accessed: 2022-06-12.

[24] *Scheduler extender*. https://github.com/kubernetes/design-proposals-archive/blob/main/scheduling/scheduler_extender.md. Accessed: 2022-06-01.

[25] *Scheduler extender*. https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/624-scheduling-framework/README.md. Accessed: 2022-06-01.

[26] *WattTime - The Power to Choose Clean Energy*. https://www.watttime.org/. Accessed: 2022-06-01.

[27] *WattTime - API reference*. https://www.watttime.org/api-documentation. Accessed: 2022-06-01.

[28] *electricityMap - The leading resource for 24/7 electricity CO2 data*. `https://electricitymap.org/`. Accessed: 2022-06-01.

[29] *electricityMap - API reference*. `https://static.electricitymap.org/api/docs/index.html`. Accessed: 2022-06-01.

[30] *Let's Wait Awhile - Datasets, Simulator, Analysis*. `https://github.com/dos-group/lets-wait-awhile`. Accessed: 2022-06-01.

[31] *Kubernetes Scheduler*. `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/`. Accessed: 2022-06-12.

[32] *Serverless Trends*. `https://techbeacon.com/enterprise-it/state-serverless-6-trends-watch`. Accessed: 2022-06-01.

[33] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. "Serverless computing: One step forward, two steps back". In: *arXiv preprint arXiv:1812.03651* (2018).

[34] *Apache OpenWhisk: An open source serverless cloud platform*. `https://openwhisk.apache.org/`. Accessed: 2022-06-01.

[35] *Apache Software Foundation*. `https://www.apache.org/`. Accessed: 2022-06-01.

[36] *Knative Has Applied to Become a CNCF Incubating Project*. `https://knative.dev/blog/steering/knative-cncf-donation/`. Accessed: 2022-06-01.

[37] *Cloud Native Computing Foundation*. `https://www.cncf.io/`. Accessed: 2022-06-01.

[38] *Knative - Google Cloud Platforms*. `https://cloud.google.com/knative`. Accessed: 2023-02-05.

[39] *Knative on IBM Cloud*. `https://www.ibm.com/cloud/knative`. Accessed: 2023-02-05.

[40] *Knative - Amazon Web Services*. `https://aws.amazon.com/blogs/opensource/deploying-lambda-compatible-functions-eks-triggermesh-klr/`. Accessed: 2023-02-05.

[41] *TriggerMesh - Easily build event-driven applications*. `https://www.triggermesh.com/`. Accessed: 2023-02-05.

[42] N. Kaviani, D. Kalinin, and M. Maximilien. "Towards Serverless as Commodity: a case of Knative". In: Oct. 2019. ISBN: 978-1-4503-7038-7. DOI: `10.1145/3366623.3368135`.

[43] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. *A Review of Serverless Use Cases and their Characteristics*. 2020. DOI: `10.48550/ARXIV.2008.11110`. URL: `https://arxiv.org/abs/2008.11110`.

[44] J. Kuhlenkamp, S. Werner, and S. Tai. "The Ifs and Buts of Less is More: A Serverless Computing Reality Check". In: *2020 IEEE International Conference on Cloud Engineering (IC2E)*. 2020, pp. 154–161. DOI: `10.1109/IC2E48712.2020.00023`.

[45] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. 2019. DOI: `10.48550/ARXIV.1902.03383`. URL: `https://arxiv.org/abs/1902.03383`.

[46] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. *Serverless Computing: One Step Forward, Two Steps Back*. 2018. DOI: `10.48550/ARXIV.1812.03651`. URL: `https://arxiv.org/abs/1812.03651`.

[47] M. Steinbach, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict. “TppFaaS: Modeling Serverless Functions Invocations via Temporal Point Processes”. In: *IEEE Access* 10 (2022), pp. 9059–9084. DOI: `10.1109/ACCESS.2022.3144078`. URL: `https://doi.org/10.1109/ACCESS.2022.3144078`.

[48] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. “Agile Cold Starts for Scalable Serverless”. In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: `https://www.usenix.org/conference/hotcloud19/presentation/mohan`.

[49] *What is Serverless?* `https://www.ibm.com/topics/serverless`. Accessed: 2023-01-31.

[50] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. “Cirrus: A Serverless Framework for End-To-end ML Workflows”. In: *SoCC 2019 - Proceedings of the ACM Symposium on Cloud Computing*. Association for Computing Machinery, Nov. 2019, pp. 13–24. ISBN: 9781450369732. DOI: `10.1145/3357223.3362711`.

[51] M. Chadha, A. Jindal, and M. Gerndt. “Towards Federated Learning Using FaaS Fabric”. In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. WoSC’20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 49–54. ISBN: 9781450382045. URL: `https://doi.org/10.1145/3429880.3430100`.

[52] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt. “FedLess: Secure and Scalable Federated Learning Using Serverless Computing”. In: *2021 IEEE International Conference on Big Data (Big Data)*. 2021, pp. 164–173. URL: `https://doi.org/10.1109/BigData52589.2021.9672067`.

[53] M. Elzohairy, M. Chadha, A. Jindal, A. Grafberger, J. Gu, M. Gerndt, and O. Abboud. “FedLesScan: Mitigating Stragglers in Serverless Federated Learning”. In: *arXiv preprint arXiv:2211.05739* (2022). URL: `https://doi.org/10.48550/arXiv.2211.05739`.

[54] C. P. Smith, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict. “FaDO: FaaS Functions and Data Orchestrator for Multiple Serverless Edge-Cloud Clusters”. In: *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*. 2022, pp. 17–25. DOI: `10.1109/ICFEC54809.2022.00010`. URL: `https://doi.org/10.1109/ICFEC54809.2022.00010`.

[55] T. Pfandzelter and D. Bermbach. “tinyFaaS: A Lightweight FaaS Platform for Edge Environments”. In: *2020 IEEE International Conference on Fog Computing (ICFC)*. 2020, pp. 17–24. DOI: `10.1109/ICFC49376.2020.00011`.

[56] M. Chadha, A. Jindal, and M. Gerndt. "Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021, pp. 478–483. DOI: `10.1109/CLOUD53861.2021.00062`. URL: `https://doi.org/10.1109/CLOUD53861.2021.00062`.

[57] B. Przybylski, M. Pawlik, P. Zuk, B. Łagosz, M. Malawski, and K. Rzadca. "Using unused: non-invasive dynamic FaaS infrastructure with HPC-whisk". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2022, pp. 1–15.

[58] A. Jindal, J. Frielinghaus, M. Chadha, and M. Gerndt. "Courier: Delivering Serverless Functions Within Heterogeneous FaaS Deployments". In: *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21)*. UCC '21. Leicester, United Kingdom: Association for Computing Machinery, 2021. ISBN: 978-1-4503-8564-0/21/12. DOI: `10.1145/3468737.3494097`. URL: `https://doi.org/10.1145/3468737.3494097`.

[59] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt. "Estimating the Capacities of Function-as-a-Service Functions". In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC '21 Companion. Leicester, United Kingdom: Association for Computing Machinery, 2021. ISBN: 978-1-4503-9163-4/21/12. DOI: `10.1145/3492323.3495628`. URL: `https://doi.org/10.1145/3492323.3495628`.

[60] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[61] *Scheduling algorithm in Kubernetes*. `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/`. Accessed: 2022-06-01.

[62] *Simplified Kubernetes Architecture*. `https://mohan08p.medium.com/simplified-kubernetes-architecture-3febe12480eb`. Accessed: 2023-01-31.

[63] *The Tweleve Factor App*. `https://12factor.net/`. Accessed: 2023-01-31.

[64] Z. Rejiba and J. Chamanara. "Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches". In: *ACM Comput. Surv.* 55.7 (Dec. 2022). ISSN: 0360-0300. DOI: `10.1145/3544788`. URL: `https://doi.org/10.1145/3544788`.

[65] *Scheduling Framework*. `https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/`. Accessed: 2023-02-01.

[66] *OpenFaaS makes it simple to deploy both functions and existing code to Kubernetes*. `https://www.openfaas.com/`. Accessed: 2023-01-31.

[67] *The power of interfaces in OpenFaaS*. `https://blog.alexellis.io/the-power-of-interfaces-openfaas/`. Accessed: 2023-01-31.

[68] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Isahagian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. "Serverless Computing: Current Trends and Open Problems". In: Jan. 2017, pp. 1–20. ISBN: 978-981-10-5025-1.

[69] *What is multi-cluster Kubernetes?* `https://www.mirantis.com/cloud-native-concepts/getting-started-with-kubernetes/what-is-kubernetes-multi-cluster/`. Accessed: 2023-02-05.

[70] *What is Liqo?* `https://docs.liqo.io/en/v0.7.0/`. Accessed: 2023-02-01.

[71] *Admiralty - The simplest way to deploy applications to multiple Kubernetes clusters*. `https://admiralty.io/docs/`. Accessed: 2022-06-01.

[72] *tensile-kube - enables kubernetes clusters work together*. `https://github.com/virtual-kubelet/tensile-kube`. Accessed: 2023-02-05.

[73] *Kubernetes Cluster Federation*. `https://github.com/kubernetes-sigs/kubefed`. Accessed: 2023-02-05.

[74] *Argo CD - Declarative GitOps CD for Kubernetes*. `https://argo-cd.readthedocs.io/en/stable/`. Accessed: 2023-02-05.

[75] *Fleet - GitOps at scale*. `https://github.com/rancher/fleet`. Accessed: 2023-02-05.

[76] *Flux CD - tool for keeping Kubernetes clusters in sync*. `https://github.com/fluxcd/flux2`. Accessed: 2023-02-05.

[77] *Simplifying multi-clusters in Kubernetes*. `https://www.cncf.io/blog/2021/04/12/simplifying-multi-clusters-in-kubernetes/`. Accessed: 2023-02-05.

[78] *eBPF-based Networking, Observability, Security*. `https://github.com/cilium/cilium`. Accessed: 2023-02-05.

[79] *Submariner - A tool built to connect overlay networks of different Kubernetes clusters*. `https://github.com/submariner-io/submariner`. Accessed: 2023-02-05.

[80] *Multicloud communication for Kubernetes*. `https://github.com/skupperproject/skupper`. Accessed: 2023-02-05.

[81] *Istio - A open source service mesh*. `https://istio.io/latest/about/service-mesh/`. Accessed: 2023-02-05.

[82] *Linkerd - A different kind of service mesh*. `https://github.com/linkerd/linkerd2`. Accessed: 2023-02-05.

[83] *Network Fabric - Liqo*. `https://docs.liqo.io/en/v0.7.0/features/network-fabric.html`. Accessed: 2023-02-01.

[84] *Peering - Liqo*. `https://docs.liqo.io/en/v0.7.0/features/peering.html`. Accessed: 2023-02-01.

[85] *Offloading - Liqo*. `https://docs.liqo.io/en/v0.7.0/features/offloading.html`. Accessed: 2023-02-05.

[86] *Virtual Kubelet*. `https://virtual-kubelet.io/`. Accessed: 2022-06-01.

[87] *Kubelet | Kubernetes docs*. `https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet`. Accessed: 2022-06-01.

[88] *WireGuard - Fast, Modern, Secure VPN Tunnel*. https://www.wireguard.com/. Accessed: 2023-02-01.

[89] *Software Carbon Intensity (SCI) Specification*. https://github.com/Green-Software-Foundation/software_carbon_intensity. Accessed: 2023-01-31.

[90] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis. "Centralized Core-Granular Scheduling for Serverless Functions". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019. ISBN: 9781450369732. DOI: 10.1145/3357223.3362709. URL: https://doi.org/10.1145/3357223.3362709.

[91] Y. Li, Y. Lin, Y. Wang, K. Ye, and C.-Z. Xu. "Serverless Computing: State-of-the-Art, Challenges and Opportunities". In: *IEEE Transactions on Services Computing* (2022), pp. 1–1. DOI: 10.1109/TSC.2022.3166553.

[92] A. Suresh and A. Gandhi. "FnSched: An Efficient Scheduler for Serverless Functions". In: *Proceedings of the 5th International Workshop on Serverless Computing*. WOSC '19. Davis, CA, USA: Association for Computing Machinery, 2019. ISBN: 9781450370387. DOI: 10.1145/3366623.3368136. URL: https://doi.org/10.1145/3366623.3368136.

[93] A. Jeatsa, B. Teabe, and D. Hagimont. "CASY: A CPU Cache Allocation System for FaaS Platform". In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2022, pp. 494–503. DOI: 10.1109/CCGrid54584.2022.00059.

[94] M. Szalay, P. Mátray, and L. Toka. "Real-time task scheduling in a FaaS cloud". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021, pp. 497–507. DOI: 10.1109/CLOUD53861.2021.00065.

[95] A. Alahmadi, A. Alnowiser, M. M. Zhu, D. Che, and P. Ghodous. "Enhanced First-Fit Decreasing Algorithm for Energy-Aware Job Scheduling in Cloud". In: *2014 International Conference on Computational Science and Computational Intelligence*. 2014. DOI: 10.1109/CSCI.2014.97.

[96] G. von Laszewski, L. Wang, A. Younge, and X. He. "Power-aware scheduling of virtual machines in DVFS-enabled clusters". In: Oct. 2009, pp. 1–10. DOI: 10.1109/CLUSTR.2009.5289182.

[97] X. Zhu, L. T. Yang, H. Chen, J. Wang, S. Yin, and X. Liu. "Real-Time Tasks Oriented Energy-Aware Scheduling in Virtualized Clouds". In: *IEEE Transactions on Cloud Computing* 2.2 (2014), pp. 168–180. DOI: 10.1109/TCC.2014.2310452.

[98] Í. Goiri, R. Beauchea, K. Le, T. D. Nguyen, M. E. Haque, J. Guitart, J. Torres, and R. Bianchini. "GreenSlot: Scheduling energy consumption in green datacenters". In: *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–11. DOI: 10.1145/2063384.2063411.

[99] *What is Brown Energy?* https://en.wikipedia.org/wiki/Brown_energy. Accessed: 2023-02-05.

[100] C. Li, R. Wang, D. Qian, and T. Li. "Managing Server Clusters on Renewable Energy Mix". In: *ACM Trans. Auton. Adapt. Syst.* 11.1 (Feb. 2016). ISSN: 1556-4665. DOI: 10.1145/2845085. URL: https://doi.org/10.1145/2845085.

[101] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman. "KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem". In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4228–4237. DOI: 10.1109/JIOT.2019.2939534.

[102] *Mosek - A software package for the solution of linear, mixed-integer linear, quadratic, mixed-integer quadratic, quadratically constraint, conic and convex nonlinear mathematical optimization problems*. https://www.mosek.com/. Accessed: 2023-02-05.

[103] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. *HEATS: Heterogeneity- and Energy-Aware Task-based Scheduling*. 2019. DOI: 10.48550/ARXIV.1906.11321. URL: https://arxiv.org/abs/1906.11321.

[104] P. Townend, S. Clement, D. Burdett, R. Yang, J. Shaw, B. Slater, and J. Xu. "Invited Paper: Improving Data Center Efficiency Through Holistic Scheduling In Kubernetes". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 156–15610. DOI: 10.1109/SOSE.2019.00030.

[105] T. Menouer. "KCSS: Kubernetes container scheduling strategy". In: *The Journal of Supercomputing* 77 (May 2021). DOI: 10.1007/s11227-020-03427-3.

[106] *A distributed, reliable key-value store for the most critical data of a distributed system*. https://etcd.io/. Accessed: 2023-02-05.

[107] *Carbon Aware SDK by Green Software Foundation*. https://github.com/Green-Software-Foundation/carbon-aware-sdk. Accessed: 2023-02-05.

[108] *k6 - A load testing tool*. https://k6.io/docs/. Accessed: 2023-02-05.

[109] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and lt;0.5MB model size*. DOI: 10.48550/ARXIV.1602.07360. URL: https://arxiv.org/abs/1602.07360.

[110] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[111] *Understanding TF-ID: A Simple Introduction*. https://monkeylearn.com/blog/what-is-tf-idf. Accessed: 2023-02-05.

[112] *Linear Algebra - NumPy - API Reference*. https://numpy.org/doc/stable/reference/routines.linalg.html. Accessed: 2023-02-05.

[113] *NumPy - Python library used for working with arrays*. https://numpy.org/. Accessed: 2023-02-05.

[114] *A pure-Python implementation of the AES*. https://pypi.org/project/pyaes/. Accessed: 2023-02-05.

[115] *A machine learning framework based on the Torch library*. `https://pytorch.org/`. Accessed: 2023-02-05.

[116] *HTML/XML template engine for Python*. `https://chameleon.readthedocs.io/en/latest/`. Accessed: 2023-02-05.