


Holistically Optimizing Geospatial Serverless Workflow Deployment for Sustainable Computing

Master Thesis**Author(s):**

Gsteiger, Viktor 

Publication date:

2024

Permanent link:

<https://doi.org/10.3929/ethz-b-000695846>

Rights / license:

In Copyright - Non-Commercial Use Permitted



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Master's Thesis Nr. 512

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Cloud Infrastructure Research for Reliability, Usability, and Sustainability Lab, University
of British Columbia

Holistically Optimizing Geospatial Serverless Workflow Deployment for Sustainable
Computing

by

Viktor Gsteiger

Supervised by

Prof. Ana Klimovic, Prof. Mohammad Shahradd

November 2023–May 2024

D IN FK

Abstract

In the face of growing environmental concerns, the demand for sustainable computing practices has never been more urgent. The environmental footprint of the cloud industry is rapidly growing due to the evermore computing required for today’s tasks and complex workflows. At the same time, carbon emissions from energy production still vary greatly, both geographically and temporally. The question arises whether it is possible to use the differences in grid carbon intensities to lower cloud applications’ carbon emissions by optimizing the geospatial deployment of today’s prevalent workflows. Serverless computing, a paradigm of mostly stateless computing gaining widespread acceptance among users and providers alike, is an example of the modern cloud workflow landscape that offers itself as a proving ground for geospatial deployment due to the relative ease of migration.

This thesis introduces CARIBOU as a holistic framework that enables the dynamic and fine-grained offloading of complex serverless workflows over geo-distributed regions, taking advantage of differences in grid carbon intensities for sustainable computing practices. CARIBOU dynamically adapts to the ever-changing realities of energy grids and applications, where a change in input size or computational load can lead to significant differences in optimal deployment. We empower developers to utilize the framework with minimal effort, providing a simple API for developing complex workflows that require no change from the providers’ side. The developer can define fine-grained location and metric constraints and multiple objectives to maximize customizability. This allows the developer to communicate the complexities of their workflow requirements and, at the same time, enables a large host of applications to benefit from offloading.

This thesis shows how our framework can reduce operational carbon emissions by up to 90.2% across Europe. Additionally, we evaluate how the framework observes the developer-defined constraints when making deployment decisions. Implementing the CARIBOU framework provides the infrastructure for future research into the individual components. It proves that building such a framework is both feasible and beneficial for the goal of sustainable computing. By showcasing the feasibility of carbon-aware geospatial deployment, CARIBOU pushes the boundaries of existing cloud application research in a new direction.

Acknowledgements

I want to thank my UBC supervisor, Mohammad Shahrad, for guiding me throughout the project and always providing insightful feedback. Your patience and perseverance got this project over the finish line. You helped me when I needed new insights and always asked the hard questions. Additionally, I would like to thank my ETH supervisor Ana Klimovic for allowing me to work on this project on the other side of the planet in a **geo-distributed** fashion and supporting me along the way. Furthermore, I am very grateful to my fellow students in the CIRRUS Lab, especially Pin Hong (Daniel) Long and Yiran (Jerry) Sun, for their hard work on this project with me, but also to Amin, Arshia, Changyuan, Ghazal, Nalin, Nima, Parshan, and Praveen for their invaluable feedback at the weekly meetings and all the fruitful discussions during my time in Vancouver.

I would also like to express my gratitude to the Zeno Karl Schindler Foundation, Mitacs, and the Swiss-European Mobility Programme. Your support has enabled me to conduct this project at the University of British Columbia in Vancouver, Canada. I would also like to thank the University of British Columbia for hosting me through the Visiting International Students Program, allowing me to work on-site, making me feel welcome, and allowing me to gain invaluable experiences in an international research environment. Lastly, I would also like to thank ETH Zürich for the institutional support that made this exchange possible.

My most enormous thanks go to my mother, friends, and especially my partner Viola. Thank you for being a part of my journey and your continued support.

Contents

List of Figures	iv
List of Tables	vi
Acronyms	vi
Declaration	viii
1 Introduction	1
2 Background	6
2.1 Carbon Variances and Electrical Grids	6
2.2 Cloud Service Providers	10
2.3 Cloud Applications	12
2.4 Serverless	17
2.5 Application Abstraction	21
2.6 Data Transmission	22
2.7 Summary and Contextualization	24
3 Motivation	26
3.1 Motivational Example	26
3.2 Feasability of Geospatial Shifting	28
3.3 Existing Solutions	29
4 Framework Design	33
4.1 Desired Properties	34
4.2 Framework Overview	34
4.3 Workflow Model	36
4.4 Metrics	40
4.5 Determining Optimal Deployments	46
4.6 Dynamic Triggering of Policy Determination	47
4.7 Application Deployment and Re-Deployment	49
4.8 Cross-Regional Workflow Execution	51

5	Framework Implementation	55
5.1	Overview	55
5.2	Component Triggers	57
5.3	Deployment Plan	57
5.4	Interactions	58
5.5	Framework Components	64
6	Framework Evaluation	79
6.1	Experimental Setup	79
6.2	Benchmark Workflows	81
6.3	Carbon Related Evaluation	83
6.4	Other Objectives	93
6.5	Framework Overhead	94
7	Discussion	99
7.1	Results	99
7.2	Limitations	101
8	Related Work	103
8.1	Serverless Workflow Deployment	103
8.2	Sky Computing	104
8.3	Provider Perspective	105
8.4	Edge Computing	105
9	Conclusion	106
9.1	Future Work	108
	Bibliography	110
A	Workflow Configuration	132
A.1	Configuration File	132
A.2	IAM Policy File	133
B	Deployment Plan	134
B.1	Example Deployment Plan	134
C	Developer Permissions	136
C.1	Required Permissions for Framework Developer Access	136
D	Framework Implementation Details	138
D.1	Framework Source Code Files	138
D.2	Framework Component Interaction	139

E	Extended Evaluation Plots	140
E.1	Extended Efficiency of Self-Adaptive Re-Deployment	140

List of Figures

2.1	Outline of the different electricity grids in Europe.	7
2.2	Carbon intensity of selection of European countries between 29th of June 2023 to the 1st of January 2024.	9
2.3	Public European AWS locations.	11
2.4	Ping between European AWS data centers.	23
3.1	Motivational workflow example.	26
3.2	Motivational offloading example.	28
4.1	CARIBOU framework overview.	35
4.2	Example workflow DAG.	37
4.3	Example source code functions.	38
4.4	Example path.	39
4.5	Complex example workflow DAG.	40
4.6	Monte Carlo simulation example.	42
4.7	Illustrative example of workflow invocation.	53
5.1	Workflow architecture.	56
5.2	Adaptive triggering policy.	66
5.3	Metric Manager overview.	75
6.1	DNA Visualization workflow structure.	81
6.2	Image Processing workflow structure.	81
6.3	Text2Speech Censoring workflow structure.	82
6.4	Video Analytics workflow structure.	82
6.5	MapReduce workflow structure.	82
6.6	Workflow deployment region combination (Europe) relative execution carbon emissions.	84
6.7	Workflow deployment region combination (North America) relative execution carbon emissions.	86
6.8	Execution / Transmission ratio compared with potential car- bon savings when shifting geospatially.	86
6.9	Tolerances and respective deployments (Europe).	87
6.10	Tolerances and respective deployments (North America).	88

6.11	Deployment decisions MapReduce over six days (Europe). . .	89
6.12	Deployment decisions MapReduce over six days region limited (Europe).	90
6.13	Deployment decisions MapReduce over six days (North Amer- ica).	91
6.14	Carbon forecast MAPE over multiple days.	92
6.15	Normalized relative carbon different solving frequencies per week.	92
6.16	Workflow execution time evaluation.	94
6.17	Workflow execution cost evaluation.	95
6.18	Workflow execution carbon evaluation.	96
E.1	Deployment decisions DNA Visualization over six days (Eu- rope).	140
E.2	Deployment decisions Image Processing over six days (Europe).141	
E.3	Deployment decisions Text2Speech Censoring over six days (Europe).	141
E.4	Deployment decisions Video Analytics over six days (Europe). 142	

List of Tables

2.1	Energy mix by country in 2022.	8
2.2	Country to Cloud Service Provider Region Mapping.	11
2.3	Required qualities of cloud applications for offloading.	15
2.4	Energy Consumption of data transmission per GB in different studies.	24
4.1	Units and Descriptions for modeling cloud application carbon emissions.	44
4.2	Units and Descriptions for modeling data transmission carbon emissions.	45
5.1	Component trigger timings.	57
6.1	Benchmark Solve and Migration overhead numbers.	98
8.1	Overview of different capabilities of frameworks for serverless cloud workflow deployment.	104
D.1	Python files to framework component	138
D.2	Component interaction tables.	139

Acronyms

AWS Amazon Web Services

DAG Directed Acyclic Graph

DM Deployment Manager

DMi Deployment Migrator

DP Deployment Plan

DS Deployment Solver

DU Deployment Utility

FaaS Function as a Service

HBSS Heuristic-biased Stochastic Sampling

IC Invocation Client

MM Metrics Manager

RECs Renewable Energy Credits

Declaration

The framework developed as part of this thesis was a team effort. While the author of this thesis has had a significant stake in the work done, it is also essential to acknowledge and point out the parts where other authors' contributions were significant. The following attribution is a best-effort attempt at contributing to the work done appropriately. All contributors actively ideated many of the framework's implementations, arguments, and solutions. The following list outlines the parts where other main contributors were significantly involved in the development:

- Pin Hong (Daniel) Long:
 - Section 4.4: The main contributor to the metrics modeling and lead the effort to quantify the carbon emissions.
 - Section 4.5: The main contributor to the brute force solver.
 - Subsection 5.5.7: Main contributor to the Metrics Manager implementation.
 - Section 6.5: Main contributor to both the runtime and cost overhead figures, added for completeness.
- Yiran (Jerry) Sun:
 - Section 5.5.2: Co-Contributor to the Deployment Manager together with the author.
 - Section 8.1: Co-Contributor to the overview table.

Additionally, the author of this thesis must acknowledge the use of generative artificial intelligence (genAI). The author restricted the usage of genAI to the plotting scripts of figures and minor latex formatting questions where the author used ChatGPT [168] and Github Co-Pilot [86] for supporting the scripting, no data was generated by genAI. The written text of this thesis is the author's own words. The author additionally utilized the grammar correction functionality of Grammarly [99]. However, the author strictly refrained from using any of the genAI functionalities of Grammarly Premium.

Chapter 1

Introduction

In these globally uncertain times, one topic remains constant: the growing environmental impact of humans on this planet. The global research community broadly agrees that energy consumption, and specifically energy usage, needs to become more sustainable and efficient. Significant disparities in both production carbon impact and usage still exist. Cloud applications, similar to most other industries, such as aviation, are similarly under scrutiny. The global environmental footprint of the ICT sector must be observed and reduced. Recent studies put the sector's contribution to global carbon emissions at approximately 1.8% to 3.9% [1, 40, 159, 163]. While the ICT sector encompasses much more than cloud applications that execute in data centers, it is still an indicator of the community's relative weight and corresponding responsibility. With the rise of computing requirements due to artificial intelligence [102], streaming, and remote work [156, 177], the electricity usage numbers are rising again after a mid-2000 plateau [60]. According to the International Energy Agency, the electricity usage of data centers will double by 2026 from 460 Terawatt-hours (TWh) in 2022 to 1'000 TWh [106]. Sustainable energy sources such as wind and solar alone can not cover this increase in usage [184]. On the contrary, the instability of renewable energy sources due to the dependency on weather and climate and lengthy processes for getting permissions [76] create new issues regarding service stability [82]. Even though electricity grid operators add new battery capacities at rapid rates to flatten the production peaks of renewables [176], these only help to a certain extent. The increased energy consumption is unevenly distributed and focuses on specific population-heavy locations where the current approach of deploying computing close to the customers generates heavy loads on data centers that are frequently not situated in the least carbon-intensive electricity grids [198].

To believe that providers will solve the problem for the customers is naive. While the providers increasingly claim carbon neutrality or have net-zero goals [23, 92], they achieve these primarily through Renewable Energy

Credits (RECs) or similar accounting strategies that, while improving the situation by providing capital for future renewable investments, do not ensure the actual usage of renewable energy [2,87]. Cloud service providers rely on the local energy grid, tapping into the same sources with the same carbon intensity as everyone else, adding more strain on the grid in peak times. While we, as researchers from the outside, need more information on the inner workings of a commercial data center to work in that direction, we do have information on the daily and seasonally fluctuating carbon intensities of electrical grids. This fluctuating carbon presents a viable opportunity to shift cloud application workloads to data centers located in grids with lower carbon intensity, thereby leveraging grid variations over time and in a broader sense.

Cloud service providers might shift internal workloads across regional boundaries, but the providers do not pursue this for client workflows [39,223] because the developers of workflows cannot communicate priorities and tolerances. Internally, cloud providers such as Google have already been practicing moving workloads to regions with a higher share of renewable energy sources since 2021 [90]. While cloud service providers have made strides in helping customers understand and manage their carbon footprint, as seen at Amazon Web Services (AWS) [9], Google [94], Microsoft [152], and IBM [104], no cloud provider offers the dynamic workload shifting so far. From a customer perspective, the deployment and regional allocation of resources remain static and manual, where the cloud providers may indicate what data centers are greener [93] without any dynamic tooling to adapt to changes in grid intensities. Shifting cross-regionally could yield more substantial improvements than staying confined to the regional borders. However, it would introduce many new challenges, such as compliance constraints or end-to-end latency tolerances, that must be solved.

Today, prevalent applications are likely to be highly interconnected and complex constructs, exemplified in the popular micro-service architecture [44, 121]. These complex workflows consist of many smaller functions, each solving an application logic problem while benefiting from parallelization and load balancing. Serverless computing is a modern paradigm where the developer is supported by not worrying about provisioning and paying by usage. The function as a service offering represents stateless and short-lived functions for logic execution, providing a valuable testing ground for cloud applications more broadly. The statelessness and independence of serverless computing units, usually functions, make them a great candidate as they simplify moving the execution to beneficial regions.

Compliance constraints, such as GDPR [74] in Europe or HIPAA [166] in the US, may lead to requiring parts of the computation to remain in "sticky" regions where offloading is impossible. The current approach of mandating that the whole workflow remains in this "sticky" region, where the workload can not be moved out of, might result in significant carbon overhead.

Requiring a fast response to client requests adds another dimension where latency tolerances might not allow offloading due to added transmission latency.

This thesis introduces CARIBOU¹ to provide a proof-of-concept that a framework that enables cross-regional offloading while remaining overhead aware is possible and feasible. To efficiently facilitate the geospatial shifting, the framework must be able to take advantage of constantly shifting carbon intensities while accounting for any tolerance or constraint communicated by the workflow developer. CARIBOU must additionally consider the structure or workloads of workflows, latency, cost, transmission carbon overheads, external data accesses, and carbon swings by the grid.

The framework sits between existing cloud service providers and developers, utilizing the providers' products while enabling developers to create and deploy serverless workflows. The serverless workflows are deployed geospatially by CARIBOU to take advantage of carbon fluctuations while solving regulatory and logistical challenges. The API of the framework allows developers to create and deploy complex workflows to the cloud providers. CARIBOU actively manages these workflows by automatically re-deploying them fine-grained per workflow stage to beneficial regions according to the developer-defined constraints and tolerances. While re-deploying workflows, the framework takes advantage of regions' carbon, cost, and end-to-end latency differences. CARIBOU requires minimal changes from the developers to their existing workflows while requiring no change from the provider. The proposed framework is our proving ground to answer the following research questions in an initial version:

The core question for this thesis based on the outlined challenges for geospatial carbon reduction is: **Can we reduce the carbon emissions of complex cloud workflows by optimizing the geospatial deployment fine-granularly?**

More specific questions defining this thesis are:

- "How can developers efficiently communicate tolerances, objectives, and constraints and define complex workflows in source code?"
- "What metrics are required to model the application regarding objectives and tolerances, what data are these based on, and how do we transform the raw data to be useful for an end-to-end modeling of complex workflows?"
- "When and in what way is a geospatial deployment generated to maximize the carbon reduction while minimizing the framework's overhead?"

¹Carbon Aware SeRverless GeospatIal Balancing DeplOyment Utility

- "How can a complex serverless workflow be deployed, migrated, and executed cross-regionally without requiring developer input or changes to the defined application?"

Contributions

The contributions of this thesis focus on the implementation of a framework that enables us to answer the above research questions while offering a comprehensive solution to fine-grained geospatial serverless function deployment to overcome current limitations and showcase a new avenue of research:

- Identifying shortcomings of existing deployment solutions with carbon as a primary objective
- Conducting the required research into cloud application carbon modeling to identify functions to provide the carbon emissions when given a complex workflow.
- Introducing three metrics the framework can operate as tolerances and objectives: carbon, cost, and end-to-end latency. For each, we developed the corresponding modeling of the metrics.
- Developing a way to model complex workflows based on past empirical data.
- Creating the required theoretical model that allows the abstraction of the workflows.
- Providing an easy-to-use developer interface that enables the definition of complex workflow structures while providing tools to communicate tolerances, objectives, and constraints.
- Implementing tools to deploy and migrate serverless workflows geospatially, requiring no manual steps from end-users or developers.
- Solving the question of how to determine the optimal deployment and when to do so.
- Creating the framework with extensibility in mind; adding new languages, metrics, and components must remain simple and require little effort to provide the infrastructure for future research.
- Evaluating the framework comprehensively regarding the research questions while not being blinded by shortcomings.

Organization

The remainder of this thesis is structured as follows:

- **Chapter 2: Background** on the fundamental mechanisms of electrical grids, complex applications, serverless applications, and appli-

cation modeling required to enable a framework to take advantage of the potential benefits of geospatial deployment.

- **Chapter 3: Motivate** the specific comprehensive solution implemented in this thesis by showcasing desired properties, limitations in the existing solutions, and a motivational example application.
- **Chapter 4:** Outline the **design** of our framework regarding workflows the framework needs to support, required functionalities, and outline high-level design decisions.
- **Chapter 5:** Introduce the **implementation** of the framework by showcasing how the components interact, when and how the subparts execute, and outlining specific component-wise implementation challenges that need to be solved.
- **Chapter 6: Evaluation** of the improvements our framework offers over static, single-region deployments to answer the research question while comparing to other solutions and illustrating the framework overhead.
- **Chapter 7:** High-level **discussion** of the results and the limitations of our framework.
- **Chapter 8:** Introducing **related works** that offer serverless application deployment and other solutions for carbon-reduction techniques concerning deployment.
- **Chapter 9: Conclusion** and potential future avenues of research.

Definition 1. Definitions that are reused throughout the thesis and represent essential non-trivial concepts are highlighted in green boxes and introduced when we first use them.

Insight 1. During the **background** chapter, we will regularly summarize the most important insights to highlight the relevant findings that either motivated or guided framework design aspects or the research questions.

Chapter 2

Background

Before we delve into the concrete motivation of our proposed solution and outline the design, we first need to provide a background on the essential concepts we must consider when building a framework for geospatial cloud application offloading. We will first introduce and investigate electrical grids' role in carbon emissions and highlight the fundamental regional differences that enable us to effectively offload (§2.1). Following, we outline the properties of cloud service providers relevant for geospatial offloading and how they relate to the underlying electrical grids (§2.2). Next, we characterize cloud applications to gather insights into what applications may benefit from offloading while also introducing the techniques used for modeling the carbon emissions of cloud application executions (§2.3). We will continue with serverless and introduce different characteristics of serverless that enable it as an ideal candidate for offloading (§2.4). Succeeding, we will discuss current cloud application abstractions required to split a complex workflow into smaller, offloadable tasks to enable fine-grained offloading (§2.5). Lastly, we will talk about data transmissions within cloud applications from a latency and carbon perspective. We will highlight that a framework must address these when offloading complex interconnected workflows (§2.6).

2.1 Carbon Variances and Electrical Grids

Offloading cloud applications is motivated by the existence of different electrical grids and the respective differences in carbon intensities between the grids. In the following subsection, we will first give some background on the makeup of electrical grids (§2.1.1), connect them to their respective carbon intensity (§2.1.2), and study the effective differences, highlighting where the differences would be favorable for offloading (§2.1.3).

2.1.1 Electrical Grids

Electrical grids consist of electricity sources, a distribution grid, and consumers. Each consumer is connected to a specific grid provider that produces energy through one or more different energy sources [105]. The energy mix can consist of renewable energy such as solar, wind, and hydroelectric power or fossil-based energy such as coal or gas. Different grids may exchange energy in production or consumption spikes. However, these exchanges are known, regulated, and quantifiable for accounting purposes [64, 209].

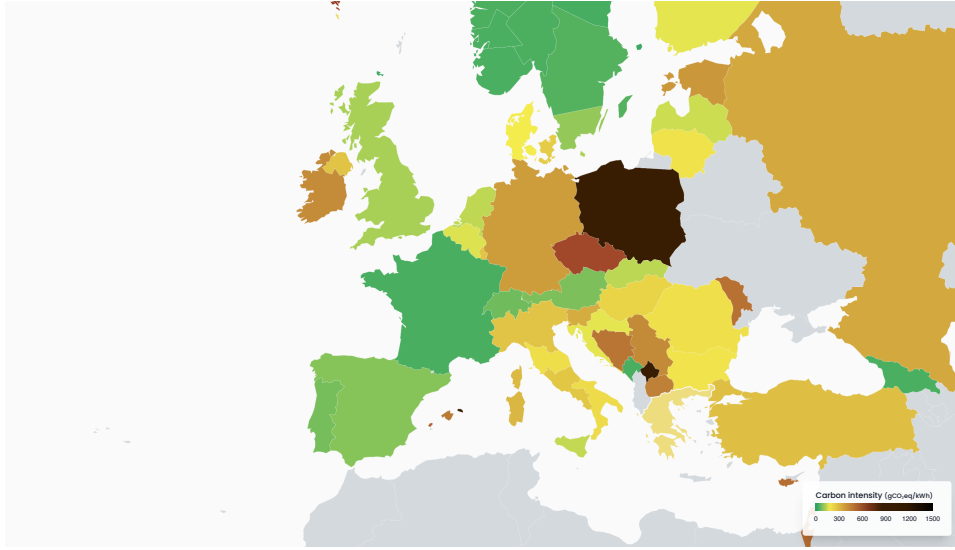


Figure 2.1: Outline of the different electricity grids in Europe. The color indicates the carbon intensity of each grid from green (low) to brown (high). Screenshot taken on Wednesday, 18th of April 2024 at 10:30 am Pacific Daylight Time (PDT) (source: Electricity Maps [73]).

2.1.2 Carbon Intensity of the Electrical Grid

Now that we have a broad perspective on electrical grids, we want to estimate the respective carbon intensity of a grid and outline how this intensity is calculated. Related research into electrical grids uses $gCO_2\text{-eq}/kWh$ to estimate the electrical grid's carbon intensity.

Definition 2. $gCO_2\text{-eq}/kWh$: gCO_2 equivalent per kilowatt hour of the emission of different greenhouse gases based on the global-warming potential over 100 years of these gases

Notably, most grid providers publish their respective energy mix. By knowing the mix and estimating the carbon intensity of energy sources,

a model for the carbon intensity of a grid at a specific time can be provided [189, 197]. Electricity Maps [73] publishes the historical data of grid carbon intensities, which we used for our evaluation, under an Open Database License (ODbL) [167]. This historical data will allow us to focus on other contributions instead of requiring us to spend time modeling this. Electricity Maps additionally offers an API where, given a location, the API returns a live carbon intensity estimate or historical data for the past seven days. Given a grid provider’s energy mix, Electricity Maps models the carbon intensity, measured in $gCO_2\text{-eq}/kWh$, of that specific grid over time. If current data is unavailable, Electricity Maps forecasts an estimated carbon intensity. Electricity Maps calculates the carbon intensity of the whole lifecycle of different electricity sources based on the IPCC (2014) Fifth Assessment Report [29].

2.1.3 Carbon Intensity Variances

Knowing how the carbon intensity is modeled allows us to evaluate the effective differences between electrical grids, highlighting the potential benefits of offloading workloads to different grids. Based on Figure 2.1 that already shows stark differences at a specific time, we can take a more analytical approach and give some additional background on the variances in carbon intensities over region and time. We will highlight the eight regions: Germany, Ireland, Northern Italy, the UK, Spain, Switzerland, France, and Sweden. We selected these regions for an in-depth discussion in this section for multiple reasons, including their large population densities, large secondary or tertiary industry sectors, proximity, availability of compute hubs, different regulatory zones, and varied energy sources. We list the energy sources for the example year 2022 in Table 2.1. We chose the year 2022 since some government agencies did not produce reports for 2023 until the deadline of this thesis. While there is a general trend towards more carbon-neutral energy sources, the general patterns of 2022 can also be observed in 2023.

Country	Primary Energy Source	Secondary Energy Source
Germany [68]	Coal (33.3%)	Wind (24.1%)
Ireland [165]	Gas (41%)	Wind (28%)
Northern Italy [199]	Gas (63.9%)	Hydroelectric (10.6%)
UK [80]	Gas (38.4%)	Wind (24.7%)
Spain [66]	Gas (29.5%)	Wind (22.4%)
Switzerland [81]	Hydroelectric (48.2%)	Nuclear (39.9%)
France [67]	Nuclear (71.2%)	Various Renewables (17.7%)
Sweden [4]	Hydroelectric (41%)	Nuclear (29%)

Table 2.1: Energy mix by country in 2022.



Figure 2.2: The carbon intensity of the European countries discussed so far from the 29th of June 2023 to the 1st of January 2024. We chose two week-long windows to highlight better short-term variability (data source: Electricity Maps [73]).

We illustrate the electricity grids' carbon intensity over a year's timespan. This illustration allows us to investigate how the eight regions fare concerning carbon intensities. For this investigation, we will use Figure 2.2 to draw insights from these regions:

- Grid carbon intensity varies widely due to the local energy sources of each location. Sweden, France, and Switzerland enjoy low carbon intensities because their primary energy sources are hydro and nuclear power. Germany, Ireland, and Northern Italy have the highest carbon intensities due to their reliance on fossil energy sources. Still, they are all in densely populated areas, meaning many potential customers live nearby. Alternatively, for example, Ireland [206] is favored for data centers due to the access to both the European market and the British market due to special post-Brexit agreements and low corporate taxes.
- Carbon intensity follows a daily pattern. Germany exemplifies this pattern by relying on wind and solar as renewable energy sources.
- Carbon footprints and patterns vary across nearby regions. Even though Northern Italy and Switzerland are only about 217 KM apart, their carbon intensity varies up to sixfold, highlighting a significant gap in the optimization space.

Insight 2. Considerable differences exist between the carbon intensities of electrical grids, even those close to each other. Notably, these differences in carbon intensities are not random but follow distinct seasonal and daily patterns. If we harness these differences in daily and seasonal patterns by moving work to a region with lower carbon intensity at a specific time, we could offer significant reductions in carbon intensity.

2.2 Cloud Service Providers

A cloud service provider is a company that provides on-demand, scalable computing resources. In the following subsections, we will outline the characteristics of cloud service providers relevant to this thesis’s scope (§2.2.1) as well as connect the providers to the discussion about carbon intensities of electrical grids (§2.2.2). This thesis focuses on AWS since it is the most popular and geographically distributed cloud service provider. While considering additional cloud service providers might be interesting in the future, AWS, with its diverse set of globally distributed regions, suffices to showcase the beneficial effects of offloading.

2.2.1 Cloud Service Provider Regions

Most large cloud service providers geographically spread out their operations over multiple locations. These diverse locations are chosen based on regulatory requirements on data residency, locality to a cluster of customers, and additional disaster recovery. The offerings in these geographically diverse locations are mostly indistinguishable and homogeneous with slight differences in offerings and service architecture [57, 195]. The terms in this section refer to AWS, but similar concepts and terms apply to all major providers. AWS distributes computing resources globally in multiple geographical areas called a region. A region comprises multiple availability zones, each isolated for local disaster recovery and availability reasons. The locations are kept private due to security concerns, as this is a critical system infrastructure. An approximation is sufficient to map regions to electricity grids.

All previously introduced regions (§2.1.3) also correspond to AWS regions. We illustrate the approximate locations in Figure 2.3 and list the mapping in Table 2.2

2.2.2 Cloud Service Providers Energy Sources

All major cloud service providers either have extensive documentation [9, 90, 104, 152] or sometimes offer calculation tools [94] to estimate the carbon footprint of their applications. In this documentation, cloud service

Country	Region Code
Germany	eu-central-1
Ireland	eu-west-1
Northern Italy	eu-south-1
UK	eu-west-2
Spain	eu-south-2
Switzerland	eu-central-2
France	eu-west-3
Sweden	eu-north-1

Table 2.2: Country to Cloud Service Provider Region Mapping.

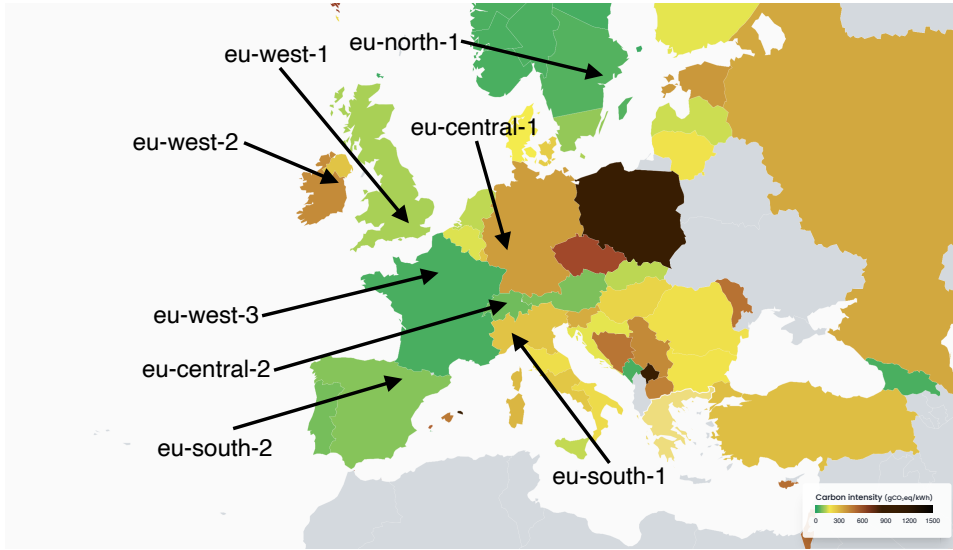


Figure 2.3: Approximate locations of the eight public European AWS regions. Screenshot taken on Wednesday, 18th of April 2024 at 10:30 am PDT (source: Electricity Maps [73]).

providers generally do not specifically disclose information on their energy sources. Furthermore, while some providers, such as AWS, build and maintain their renewable energy projects [8], only some are fully self-sustaining, given the complexity of energy production. Moreover, while providers such as Google have made strides in enabling more efficient and transparent ways to enable cloud service providers to buy cleaner energy [91], more work from both energy providers and consumers is needed to ensure genuinely 100% carbon neutral operations.

All cloud service providers participate in so-called RECs [210], time-based energy attribute certificates (T-EACs) [58, 164], offset markets, or similar accounting practices. The basic methodology is that every MWh of renewable energy produced creates one certificate. These certificates give

the owner the right to claim to use renewable electricity. Offsets are a similar accounting method; however, they count metric tons of CO_2 saved outside the company’s utilization scope. The discussion happening extensively in corresponding expert communities [43, 45, 172, 229] about the legitimacy of such carbon accounting systems and effectiveness is out of the scope of this thesis. However, the methodology only works as a theoretical carbon offset. The overproduction of renewable energy during one period has yet to be efficiently transferred to another, even though advances in battery technology are promising [176]. Additionally, since the data center and the renewable source may not necessarily be in the same grid, the data center still consumes the potentially ”dirty” local energy. Primarily since building renewable energy capacity far away from actual usage centers will not ensure an efficient energy distribution due to a lack of power lines [154] and an antiquated power grid [198]. Thus, if the cloud service providers claim a decrease in net operational carbon emissions from using and buying renewable energy, it is not the same as using solely renewable energy sources at all times [52, 218, 219]. It is legitimate to take the energy mix at the time of energy consumption as the base of the energy source.

By guessing the location at the granularity of an electrical grid, we can estimate the current energy mix used by a specific cloud service provider’s data center. Subsequently, using the methodology introduced (§2.1.2), the carbon intensity per kWh can be calculated.

Insight 3. Cloud service providers offer mostly homogeneous services in globally distributed regions. These homogenous offerings present a significant opportunity for workloads hosted in the cloud to be relocated to regions with the lowest carbon intensity, potentially reducing carbon emissions. Even though the service providers claim and aim to reduce their carbon footprint, they still rely on the underlying electrical grid to power their data centers. More than relying on cloud providers and their green data centers is required; considering the underlying grid, a more fine-grained approach is necessary.

2.3 Cloud Applications

Since we now have an understanding of the electricity grids (§2.1) as well as the connection to the cloud service providers (§2.2.2), we now have to talk about the workloads that run at these providers. Cloud service providers nowadays offer a broad host of services, each with their specialties and differences. For example, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), or Function as a Service (FaaS) are just a few fundamental cloud service offerings. AWS, for example, currently offers around 200 fully featured services. The development, let alone the

configuration of both the deployment and execution of a modern cloud application, is non-trivial.

Additionally, many applications are nowadays interconnected and complex [121, 140]. Dependencies between both instances of the same service and other services are common, and most cloud applications’ architectures feature many different components. Academia offers tools to alleviate the problem of cloud application complexities in general and, more specifically, regarding multi-cloud deployments [77, 220]. The complexity of a modern application makes the search space for optimization regarding deployment and resource usage estimation a complex problem for individual developers.

2.3.1 Characterization of Cloud Applications

In addition to their application logic, cloud applications can be defined by their end-to-end latency, carbon intensity, and corresponding cost. These represent the three metrics for a cloud application that we consider essential in the context of this thesis. Each of these metrics, in the broadest sense, comprises *execution* and *transmission*, where execution is the actual computation done by the application and transmission is any data transmitted as part of the application process. All three metrics additionally depend on the cloud application characterization. While there is yet to be a consensus on the exact terminology when categorizing cloud applications, we focus on the following:

Duration

Generally, there are three categories of the duration of cloud applications: (1) *continuous*: These kinds of applications are up and running as long as the application exists. These usually offer user-facing logic that needs to be able to serve requests. Applications in this category usually fall into the IaaS or PaaS category, where the client sets up a server for the hosted infrastructure or platform that serves requests. They usually do not have an end date and run until canceled. (2) *long running*: While these applications, similarly to the continuous, run for a longer time, they usually have an end date. Applications in this category are usually categorized to take up to several days to finish. However, these kinds of applications can use many resources as they are more likely to be over-provisioned concerning memory and compute requests [138]. Examples of this application include machine learning training, simulations, and big data analysis applications. (3) *short running*: most cloud applications fall into this category. For example, traces from Alibaba [138] show that 90% of batch jobs, which make up a significant portion of short-running applications, run less than 15 minutes. These applications, due to their short-lived nature, are usually time-sensitive. Examples include FaaS applications, CI/CD pipelines, and

nightly deployments. Another complexity is introduced by the timing of application executions, namely service level objectives (SLOs) and agreements (SLAs). Many applications include SLAs with their end-users that guarantee a specific end-to-end latency upon invocation [169]. SLOs introduce latency tolerances that need to be observed in the applications [69].

Time of Execution

This category is defined by when an application is executed relative to the submission of its specification or an invocation. There are generally two categories in this dimension: (1) *ad-hoc*: applications that need to run immediately upon a particular event or trigger happening. These applications usually do not tolerate a significant delay in execution, and forecasting the invocation timing of such applications is a topic of its own. Examples include FaaS invocations, CI/CD runs, machine learning training, and extensive data analysis. (2) *scheduled*: These are applications where the developer defines the execution date at task submission. These applications usually range from start to end in terms of latency tolerance. They can include nightly deployments, analysis jobs, and simulations.

Preemptibility

When an application is preemptible, its execution can be stopped and continued later. Such applications either write checkpoints or have another way to ensure statefulness if required. Composite and stateless applications are usually preemptible due to the nature of the application architecture. However, not every composite or stateless application is delay-tolerant about finishing time. Thus, whether an application is preemptible depends on the application structure and latency delay tolerance.

Location Stickiness

An application is sticky to a location if there is any reason from a contractual or data residency perspective why this application cannot be moved out or into another region. A location can be a specific country, encompassing multiple regions or region-specific. Examples of why an application might be location sticky include HIPAA in the U.S. [166], PIPEDA in Canada [98], or GDPR [74] in the European Union. Additionally, the banking or health sector is usually closely regulated regarding client data [205]. Location stickiness can also be partial. Since the application consists of multiple parts, some steps of the application logic may be sticky and need to be executed in a specific region, while others are less locality-sensitive.

Applications Ability to Offload

When we discuss offloading to the extent of this thesis, we refer to applications that can be moved from one cloud service provider region to another. This discussion necessitates some characteristics of an application. In Table 2.3, we outline which qualities applications need to have to be offloadable.

Duration	Time of Execution	Preemptibility	Location Stickiness
Continuous	Ad-hoc ✓	Preemptible ✓	Location-Sticky
Long Running ✓	Scheduled ✓	Non-Preemptible ✓	Non-Location-Sticky ✓
Short Running ✓	Continuous		Partially Location-Sticky ✓

Table 2.3: The qualities Required for Application Offloading are highlighted with a checkmark. Any application that combines these qualities is theoretically offloadable.

2.3.2 Modeling Cloud Application Carbon Emissions

After identifying what applications are off loadable, we need to model an application’s carbon emissions to calculate the potential improvement we can gain from offloading. Even though the spectrum of available, different applications is broad, all applications if broken down to their core, are hosted on physical machines in data centers and auxiliary buildings spread across the globe. While the level of virtualization on top of the physical machine varies from service to service, introducing different levels of unknown factors and overheads, every application relies on similar computing, storage, memory, and IO components. This knowledge, together with the provided information about memory, computing, IO, and storage consumed, gives us a good idea of the makeup of a cloud application and its most basic components.

The exact setup of the utilized software and hardware stack for one of the services is proprietary on different levels. However, one can still make assumptions based on the open-source components and information publicly shared about the approximate makeup of the underlying architecture and machines [37]. We can estimate the resource usage, specifically the consumed kWh of a cloud application.

We can break down the resource usage of any cloud application into two categories: (1) *operational emissions*: encompassing the resources used to execute an application [122] and (2) *embodied emissions*: the lifecycle carbon emitted by the hardware components and infrastructure needed to host the application [108]. In the interest of this thesis, we will focus on the operational emissions rather than the embodied emissions. This choice was made by us because, on the one hand, the data is scarcer on the involved hardware components, and, secondly, we assume that the computer architecture used by the same provider, AWS, should be more or less homogenous, as evaluated by related research into workload placements [57, 213].

Prior research on estimating the carbon emissions of a cloud application, or any application, can be grouped into several categories. Existing overview studies [65] highlight the many modeling approaches that exist. One category of carbon emission modeling in research [56, 228] has abstracted the carbon emissions of application executions using the fossil fuel percentage of a region times runtime and memory used for a comparative study of regions. Another research cluster has approached the problem more granularly and proposed a power times the carbon intensity of the grid per kWh model. Specific research has focused on the power consumption of a particular server or operation, quantifying it as a function of time spent processing times power of computer host [145], a function of the rate of handling service requests [84], per CPU of a specific cluster [182], or as complex models of energy charges [61]. Additionally, research into the energy usage of Deep Learning [201] and Machine Learning [123] has provided models to quantify the usage of computation workloads. Some research has outlined a carbon emission model based on computing unit (CPU), memory, storage, and network to quantify the carbon footprint of computation [124], decide on where to schedule serverless functions [48], for sustainable resource management [88], or exploring power-performance tradeoffs in database systems [217]. We want to highlight the second model since it presents the most reliable estimation base when given the public information available to FaaS applications, memory (E_{mem}), CPU (E_{comp}), IO (E_{IO}), and storage ($E_{storage}$). Additionally, the power usage effectiveness (PUE) as well as the grid carbon intensity (I_{grid}) must be taken into account when calculating the overall carbon emissions. Equation 2.1 summarizes the findings of these models to a generalized model.

$$Carbon_{ex} = I_{grid} \times (E_{comp} + E_{mem} + E_{IO} + E_{storage}) \times PUE \quad (2.1)$$

Insight 4. Modern Cloud Applications are complex constructs with per-component level dependencies and tolerances. The characteristics an application needs to show to enable offloading the whole or parts of the applications show a wide range of potential beneficiaries for offloading. Moving an entire application from one region to another is often challenging due to end-to-end latency tolerances, compliance, or the invocation pattern. Calculating the execution carbon intensity is possible under some assumptions about a cloud application’s underlying resources and data center architecture. The three metrics of an application that we will focus on, *carbon intensity*, *end-to-end latency*, and *cost* can be most broadly subcategorized into *execution* and *transmission*.

2.4 Serverless

We introduce the serverless service paradigm to outline a specific type of workload that is a good proof-of-concept candidate for offloading. Serverless computing is an emerging paradigm in cloud computing. Rob Sutter, a senior developer advocate at AWS for serverless, provides four functional characteristics of serverless computing [204]:

1. No infrastructure provisioning, no management,
2. Automatic scaling,
3. Pay for value, and
4. Highly available and secure.

If we say serverless, we do not mean the software is not running on servers. Instead, the end user does not care about the specifics of the infrastructure involved. If the developer does not have to worry about infrastructure provisioning, this frees up resources for developers to focus on the application logic.

A critical aspect of the serverless paradigm is an abstraction of the involved cost model. In serverless computing, the client pays for the services used instead of continuously paying for resources regardless of their usage. This pay-per-use scheme allows for a fine-grained cost accounting where resources used directly impact the cost incurred. The exact cost model depends on the actual serverless service model. Additionally, serverless services are often stateless, allowing for more effortless movement of where computation occurs.

FaaS is one specific cloud service model that is serverless. In the FaaS service model, developers write code in high-level languages such as Python, Go, Node.js, Java, and many more. The developer can upload their code directly to a provider or package into a container, such as Docker images. Packaging the code into containers allows for easier function migration and gives the developer more power to control the execution environment. They then additionally configure an event that triggers the execution of this code, together with potential input. Examples of triggers include HTTP requests, messages in distributed messaging queues, and more. Developers can deploy at the granularity of functions, where each function represents a piece of code. For each serverless function, the developer needs to define a handler function. This handler function is invoked when an end-user invokes the FaaS instance.

The cost model of serverless functions is similar for all providers. There is a cost per invocation and a cost associated with the computing resources used. The cost per computing resource is expressed as gigabyte seconds of memory used. The rounding granularity regarding the used GB-seconds of memory is different between the different providers, from 1ms at AWS [18] to 100ms at most other providers [95], which is also why concepts such as Sky

Computing [33, 56, 220] utilizing these differences for cost gains have gained traction. Since the client pays per GB-second, the provider is also directly incentivized to report time spent on a function, enabling a fine-grained accounting of the latency (execution time) of a FaaS function. Higher-level middleware providers can offer significant cost savings with a holistic approach to the resource FaaS. We aim to present a similar vision regarding the serverless resource focusing on carbon; however, we can transfer some insights from cost- and latency-focused studies.

Characterizations of serverless functions [72, 196] have shown that:

1. most serverless functions observe short execution times; 90% execute for less than 10s,
2. functions have a small memory footprint, usually below a few GB.

These results are not surprising, given the pricing model and current tolerances of the serverless service model. One could now conclude that serverless functions do not contribute significantly enough to the global resource consumption of cloud service providers; however, they are a growing part of the service offering of all cloud service providers with significant adoption across all major cloud service providers [62]. Thus, the resource impact of serverless functions will only increase, given the current trajectory.

One serverless function can already represent a serverless workflow; however, developers increasingly connect functions to represent larger workflows. Serverless functions thus fit well into the argument of the ever-increasing complexity of cloud applications (§2.3). By abstracting away the actual server resources and additionally working as an interface to usage-driven, stateless backend services, FaaS functions are often used as glue and connected to form more complex workflows. However, complex serverless function workflows exist [127, 142] and have benefits concerning task parallelism, prolonging the maximal execution time, or fine-grained error logging.

Serverless functions provide many aspects necessary for seamless cross-regional migration. Since an event invokes a usually stateless function, the location of the execution does not matter. Storage is mainly disaggregated from execution, where the storage offering is usually a serverless service such as *dynamoDB* [16], a distributed key-value store offered by AWS or *S3* [19], AWSs blob storage solution. The migration of a Docker image representing the application logic of a serverless function is relatively simple, as the whole execution environment, including all dependencies, is packaged. Furthermore, the cloud providers' offerings of FaaS are relatively heterogeneous within the same provider. So, executing a serverless function in one region will behave similarly to running it in another. The short-lived nature and popularity of serverless functions [112, 143, 196] enables us to characterize serverless functions specifically faster than comparable service offerings, and sub-optimal decisions with regards to offloading do not have long-lasting effects on both carbon intensity and end-to-end latency. Serverless workflows'

flexibility and ease of characterization make them ideal candidates for a proof of concept and the first step in showcasing the geospatial offloading of larger cloud applications.

2.4.1 AWS Lambda

As a case study for one serverless function model hosted on one specific provider, we will discuss the system architecture of AWS Lambda, a serverless function service provided by AWS, in more detail.

Essentially, a Lambda function uses a share of the resources of the underlying Firecracker worker [3], the virtualization system used at AWS for hosting the functions. Developers can configure AWS Lambda functions with a minimum of 128 MB and up to 10,240 MB of memory. The amount of configured memory determines the virtual CPU (vCPU) available to a function ($n_{\text{vCPU}} = \frac{\text{memory}}{1,769}$) [24]. A vCPU represents the ability to run one processing thread at a time. The maximum execution time of a lambda function is 15 minutes. Depending on its physical architecture, a Firecracker worker can collocate many AWS Lambda functions. The cost model of AWS Lambda is determined by the number of invocations and the total Gigabytes the function consumes, which is the product of total memory and duration. The price is determined by each millisecond of execution time, offering a fine-grained billing approach.

Given this information and the background we introduced for modeling (§2.3.2), we can also model the resource usage of serverless functions. Any estimation concerning resource usage of FaaS functions, especially carbon estimations of said workflows, must only be used comparatively. Since we estimate many variables due to a lack of information, we cannot claim to provide any absolute numbers on FaaS’s carbon emissions. Additionally, we hope this and similar work in this area will incentivize the cloud providers to share more information concerning the carbon intensities of services provided.

2.4.2 Serverless Workflows

Serverless functions rarely function independently and are often connected to larger applications. We introduce the term serverless workflow to encompass both a single function and a sizeable interconnected net of functions. The workflows are scalable because if one part receives more load, that specific function can scale out without scaling the complete workflow. It also allows for function-level optimizations concerning resource usage and requests and allows for complex deployment configurations. Additionally, it allows workflow developers to own their functions from development to deployment, reducing the organizational strain introduced by the separation of concerns. Optimizing resources and workflow latency soon becomes

intangible due to hidden connections and unknown dependencies.

To mitigate the issues of evermore complex workflows, providers now offer tools to manage complex state diagrams that comprise multiple interconnected components, such as AWS Step Functions [20], Google Cloud Workflows [96] or Azure Logic Applications [32]. These tools introduce concepts like conditional executions, synchronization points, fan-outs, and more. These are well-known concepts to workflow developers, and many use cases require these concepts to be functional. This development of sophisticated tools is an essential step toward the serverless ecosystem’s maturity, showing that workflow requirements are becoming increasingly complex. At the same time, there is a real need for tools that automate the meticulous deployment task, invoke functions, and control flow.

Serverless Workflow Use-Cases

To get a better idea of the type of workflow that developers usually deploy as a serverless workflows [101], we present three categories of common workflows:

1. Parallel functions: workflows that take advantage of task-level parallelism, such as those that use a divide-and-conquer approach to solve problems like weather forecasting.
2. Glue-code: Workflows that connect are usually also serverless in design and services. The workflow might also apply some processing logic before handing it off to another service.
3. Compositions and pipelines: As previously mentioned, there is an increasing interest in more complicated serverless workflows in complex compositions with control flows.

Examples of workflows that fall into these categories are data visualization tasks, image processing pipelines, small-scale text-to-speech or video processing, and more complex implementations of popular paradigms such as MapReduce. These examples have also informed our benchmark selection (§6.2).

2.4.3 Serverless Workflow Execution Latency Prediction

As outlined (§2.3.2), our primary unknown regarding modeling the resource usage is the execution latency of a workflow, more specifically, in the case of serverless workflows, the latency of an individual step of the workflow. Attempts to estimate the execution latency based on input sizes become challenging, for example, in cryptographic workflows, where the same input may result in very different execution times. Characterizations show that the execution latency is not a simple result of any specific factor [111, 222]. Even assigning a different underlying processor [57] can result in variances.

Previous work has estimated the execution latency based on past invocations and complex solutions exist [71, 129]. Since latency prediction is not a primary contribution of this thesis, we can rely on the assumption that the execution latency of a function can, to a certain amount, be estimated based on previous executions. Since many unknown factors may delay or speed up a function execution, making more fine-granular claims would be ignoring the reality of the problem.

Insight 5. Serverless workflows are a good example of modern cloud applications. Serverless functions are an example of easily migratable workloads. Many of the necessary qualities to facilitate cross-regional offloading are engrained in their service offering, making them ideal candidates for this thesis’s work. The fine-granular accounting from a cost and latency perspective, as well as the disaggregation of execution and storage, help account for a good approximation of carbon incurred on an invocation level and for migrating workloads without causing the workflow logic to break.

2.5 Application Abstraction

Much prior research has been focused on abstracting away application logic, whether for latency, bottleneck, or dependency analysis. A standard method of abstracting an application involves breaking down the source code into parts that abstract their actual execution, often represented by comparative numbers like latency. The level of abstraction and what data is essential to maintain from the building blocks of abstraction depends on the problem.

As we previously outlined (§2.3.2) to model the resource usage of a cloud application, we mainly require application information regarding end-to-end latency since everything else depends on it. Application latency analysis of linear and parallel execution is a well-known problem [202] with existing solutions. The following subsection introduces one such solution that informed our workflow model. Relevant applications are parallel and high-performance computing and cloud computing.

2.5.1 Data Dependency Graphs

One abstraction widely used in parallel computing is the dataflow representation of an application [41, 192]. In a dataflow graph, the application logic is abstracted away, and the primary resource is data, which flows between computation steps. These dataflow graphs introduce many concepts, such as conditional execution and synchronization points, which are very valuable in representing complex, micro-services-like architectures. They also empower the visualization of complex architectures through data dependency

analysis, as shown by recent comparative studies [97]. Related research into choreographies [187] introduces similar data flow-based models.

Since an application’s data flow is one-directional in the context of time, the specific graph structure representing the data dependency is a Directed Acyclic Graph (DAG). The components of these DAGs are nodes representing computation or execution and edges representing data transmission. Parallel application research has long ago embraced the DAG structure to represent applications [79]. Representing applications as DAGs enables graph optimizations on top of the DAGs while maintaining the application’s semantics. Researchers use these DAGs for end-to-end runtime research [149] also required for a per complex workflow accounting model and scheduling [161], similar to our use-case. Furthermore, DAG analysis of complex microservice architecture has given the most accurate insight into application behavior and end-to-end latency [140, 141]. Another abstraction also based on DAGs is choreographies, introducing a language for describing complex workflows at a high level of abstraction [186] offering many translatable concepts concerning application abstraction. Lastly, the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [130] standard helps define applications as graphs for enhancing cloud applications’ portability and operational management.

Insight 6. We can abstract applications into building blocks of execution and transmission. Doing so simplifies the accounting model and enables discussions of application offloading on a per-execution building block without losing any dependency information concerning transmission. Data dependency graphs enable various operations and reasoning on applications abstracted into these two components.

2.6 Data Transmission

Global-scale cloud applications only exist with the Internet, which is a network of interconnected machines that transmit data between themselves in its most simplified form. Like computing, data transmission is essential to every cloud service offering. In the following subsections, we will highlight the two relevant aspects of data transmission when considering geospatial offloading: latency and transmission cost/carbon.

2.6.1 Data Transmission Latency

Data centers are interconnected globally via high-speed links, and transmitting data from one location to another is incredibly fast compared to a few years ago. In Figure 2.4, we summarize the one-way ping speed between data centers for the European AWS data center locations.

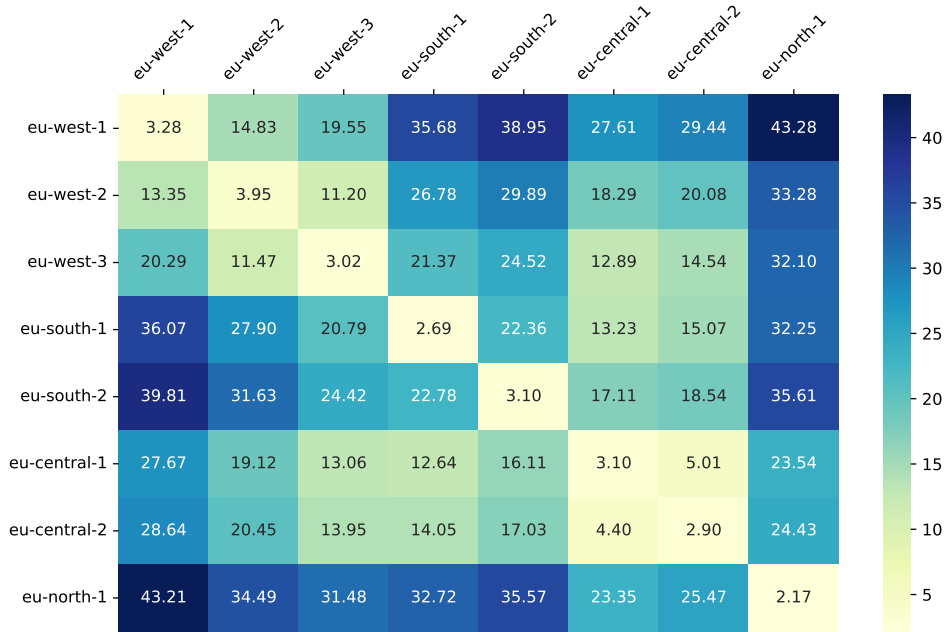


Figure 2.4: Mean one-way ping in milliseconds between European AWS data centers. Data taken on Thursday, 4th of May 2024 at 11:30 am CET (source: CloudPing [54] and WonderNetwork [185]).

Thus, connecting cloud services locally within an availability zone and cross-regionally becomes an opportunity since the applications' low latencies allow for moving data in and out of regions with relatively low latency overhead. These low latencies become especially interesting in use cases where this latency overhead can be hidden away through function-level parallelism or with applications with some tolerance for response latency.

2.6.2 Data Transmission Cost

All popular cloud service providers usually do not charge ingress fees for incoming data. However, egress fees may apply [109,120]. AWS, for example, charges egress fees per GB of data transmitted outside of a region [28]. These fees must be addressed when considering the geospatial distribution of complex applications.

2.6.3 Data Transmission Carbon Emissions

While transmission latency between clients and services and between services has always been of primary concern to cloud application developers, data transmission's carbon emissions have yet to receive the attention they deserve. Data transmissions' carbon emissions deserve additional attention.

They deserve especially attention in geospatial offloading with carbon emission reduction goals, where the carbon emissions of data transmissions might counteract our stated goal. Prior research on the carbon intensity of data transmission between two geographical locations is relatively scarce. Most research evaluates end-to-end user traffic, which integrates data transmission carbon emissions between and within data centers [137], or derive and report power consumption irrespective of transfer distance [30, 34, 178], or both [194]. The consensus is that the carbon intensity depends on the size of the data moved (S), the energy consumption of the transfer (EF_{trans}), and the carbon intensity of the traversed grid points (I_{route}). Effectively giving us Equation 2.2.

$$\text{Carbon}_{\text{tran}} = I_{\text{route}} \times EF_{\text{trans}} \times S \quad (2.2)$$

Estimates of EF_{trans} vary significantly across studies as illustrated in Table 2.4.

Year	Energy Consumption (kWh/GB)	Source
2022	0.0053	[36]
2021	0.002 to 0.007	[78]
2020	0.001	[146]
2018	0.00333	[178]
2015	0.06	[30]
2024	0.00025	[147]

Table 2.4: Energy Consumption of data transmission per GB in different studies.

Insight 7. Data transmission latency between data centers goes beyond a simple function of input size. Any viable solution for workload offloading must consider data transmission’s carbon intensity when determining where to deploy a function. The carbon intensity of data transmission is especially critical when considering the data dependencies between the building blocks of a partially offloaded application.

2.7 Summary and Contextualization

In this chapter, we introduced the necessary background to motivate the research for this thesis further and outlined the context relevant to the rest of the research. To provide the background necessary to answer the thesis questions, we discussed carbon variances in electrical grids, which open the possibility for optimization. We also highlighted that cloud providers are connected to the local electrical grids, and carbon offsetting or credit systems do not lead to less carbon intensity at the moment of consumption.

Then, by introducing the different types of cloud applications, we showed which ones are offloadable, motivating what sort of applications we are targeting. If we approach the applications as workflows that can be split into smaller parts, we increase the number of offloadable applications, necessitating fine-grained strategies. Serverless is an exemplified cloud workload encompassing serverless workflows, the ideal candidate for geospatial offloading. By showcasing the current research into carbon modeling for execution and transmission, we established that any improvement in execution carbon might be offset by transmission carbon, requiring a holistic system aware of both.

Chapter 3

Motivation

The carbon variances investigated in the background chapter (§2.1.3) together with the knowledge that offloadable applications, and specifically serverless workflows lend themselves as excellent examples, exist (§2.3.1) under the caveat that transmission carbon (§2.6.3) must be taken into account already provide a strong motivation for a framework that answers the research question as outlined in the introduction (§1). This chapter starts with a motivating example and a vision of what a framework for geospatial offloading should achieve (§3.1). Next, we will outline the factors necessary to enable a framework for geospatial shifting (§3.2), motivating further design and implementation specifics. Lastly, we outline the state of the art in frameworks for serverless workflow offloading for execution carbon reduction (§3.3) where especially the limitations (§3.3.3) additionally motivate the work of this thesis by identifying a gap for a new framework.

3.1 Motivational Example

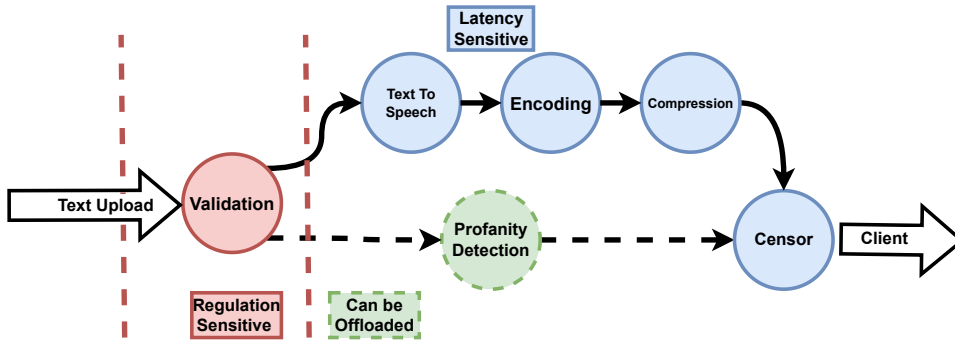


Figure 3.1: Motivational example application with both regulation-sensitive tasks and parallel, non-latency-sensitive tasks.

To motivate the research question and the framework built as part of

this thesis, we will commence with a motivational example workflow and then showcase what we envision a framework could achieve by geospatially offloading said workflow. Let us introduce an example workflow that turns text into speech and censors profanity. In Figure 3.1, we outline the workflow’s structure with all its logic blocks and data dependencies between them. When uploaded, the text first goes through a hate speech filter to remove any text passages that are illegal in the jurisdiction of the country of the uploader [83]. Since the workflow developer commits a crime if the workflow turns the text containing hate speech into actual speech, this step is sticky, and a solution is not allowed to offload this step to another region. Any subsequent steps only handle the cleaned text and can thus safely be offloaded to other countries, which offers an avenue of carbon optimizations. After this, two tracks of work happen in parallel:

1. The text is turned into speech using the Google text-to-speech API [89], then encoded into the `.wav` format, and finally compressed as a chain of tasks.
2. Profanity detection using a profanity detector library [42] that generates text indices that contain profanity.

A censoring step then gathers all the information from the previous parallel work tracks and censors the speech, returning the censored speech to the client.

This example highlights multiple concepts that motivate the framework built during this thesis:

- **Location-stickiness:** The validation step can not be moved outside of one jurisdiction and is thus location-sticky; this motivates a fine-grained deployment approach. A solution might miss many potential carbon emission optimizations if it only considers offloading the whole application.
- **Latency Sensitive:** The application has a latency-sensitive hot path and two parallel tasks that are relatively latency-non-sensitive. We need a per-task deployment to different regions considering the impact of end-to-end latency on decisions; otherwise, we might violate existing SLOs that the developer should be able to communicate through tolerances.

If we envision a scenario with three regions where we could deploy the Text2Speech Censoring workflow, the state-of-the-art would be to confine the whole workflow to one region, which may not even be the best region with regards to carbon intensity (Figure 3.2a). Our motivation stems from the need for a comprehensive solution that is adaptive to changing carbon intensities and application workloads and empowers us to shift regions of the tasks that can be offloaded as needed. Additionally, the solution needs to be able to fine-grainedly deploy tasks so that offloading does not interfere

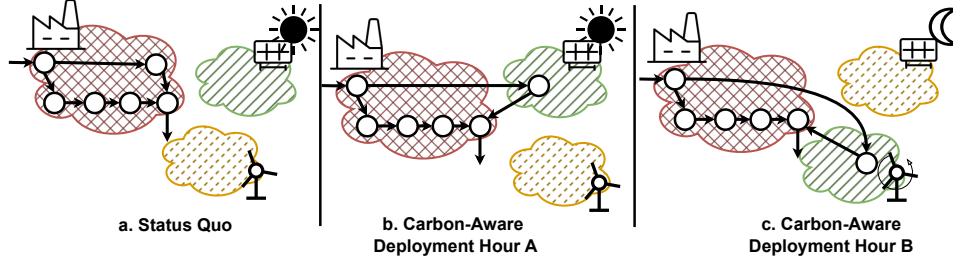


Figure 3.2: State-of-the-art is to deploy the Text2Speech Censoring workflow in one region. This thesis proposes a framework that can dynamically and carbon-awarely deploy the same workflow, taking advantage of the structure and carbon intensities in the available regions.

with tolerances or location constraints. Figure 3.2b & 3.2c shows how we envision our solution based on the motivation. In this vision, the fine-grained offloading decision for the profanity detection changes between two hours based on the carbon grid intensity due to changes in grid carbon intensities due to the renewable energy sources used.

3.2 Feasibility of Geospatial Shifting

Any solution that wants to provide an answer to the research question as outlined in the introduction (§1) by enabling geospatial shifting must take into account the following factors:

- **Compatibility:** The underlying hardware may vary from location to location. The execution environment of a workload must be guaranteed; otherwise, the outcome of a computation may vary significantly.
- **Cost:** As both execution and transmission of data vary from region to region and between providers, a viable solution enabling geospatial shifting must be aware of cost differences.
- **Latency:** Shifting from one region to another naturally causes a latency overhead, exaggerated by traffic sources/sinks and sticky data. Additionally, latency differences introduced by different hardware, software, and loads (relative pressure by co-tenants) in various regions will cause different results. Any framework that wants to offer real value to the end-user with low latency tolerances must be aware of these implications.
- **Carbon:** Carbon must be approached holistically. Both execution and transmission carbon must be considered, especially when considering the data size of transmissions. Depending on the workload, application structure, and data transmissions, offloading may cause more carbon overhead than it saves.

- **Compliance:** Enabling the developers to communicate restrictions on specific data/compute movements for compliance reasons is a necessary feature, especially when shifting geospatially.

These factors motivate a comprehensive framework that considers all these implications while being able to optimize for multiple objectives. However, since these factors vary between different applications, at various times, and for different users, a system needs to be able to adjust and adapt. A holistic solution that intricately balances the trade-offs between carbon efficiency, latency implications, cost overheads, data compliance, and the operational implications of complex cloud applications is needed to present a feasible answer to the research question.

3.3 Existing Solutions

The increased interest in reducing the carbon intensity of cloud applications has already produced a significant research output. The academic community has approached optimizing the carbon intensity of cloud workloads primarily from two directions: (1) *users and developers* that deploy workloads at the cloud providers and deeply understand the domain-specific challenges of their developed applications. While they might care about their application’s carbon intensity, they need more resources to investigate the best possible deployment or a deep understanding of the proprietary software systems that host their applications. (2) *cloud service providers* that run the workloads. Cloud service providers understand the architecture, deployment optimizations, resource usage patterns, and more. However, without the user explicitly telling them, they have very little knowledge about the tolerances the applications they host have, such as latency and regulatory requirements. Currently, clients cannot communicate such tolerances in the confines of the major cloud providers.

We categorize the state-of-the-art into two approaches: (1) *temporal* and (2) *geospatial* shifting.

3.3.1 Temporal Shifting

Temporal shifting means delaying or expediting a job in the temporal dimension. After seeing the daily carbon variances, shifting work to a beneficial time concerning the involved carbon intensity seems sensible. Previous research shows that moving a workload to a less carbon-intensive time slot can benefit the workflow’s overall carbon intensity. Cloud workflows that are not latency sensitive, such as CI/CD pipelines or nightly deployments, can be moved around in a given time window and are perfect candidates for this approach. Prior research such as “Let’s Wait a While” [214], among others [61, 134, 139, 191] attempted at such frameworks, while also highlight-

ing clear limitations. Their dependence on temporally shiftable workflows limits all of these frameworks. Workflows that can benefit from temporal shifting fall into the categories of *long* or *short running*, *scheduled*, and *pre-emptible* workflows. *Continuous* workflows are not shiftable by nature, while *ad-hoc* workflows also by nature expect an immediate execution, which rules out temporal shifting in most cases. *Non-preemptible* workflows are theoretically also temporally shiftable when the execution is relatively short or predictable.

3.3.2 Geospatial Shifting

The diagram of carbon intensities in different regions (Figure 2.2) motivates geospatial shifting. Frameworks such as "GreenCourier" [48] showed that implementing a geospatially aware scheduling policy in Kubernetes for scheduling serverless workflows can result in reducing the carbon intensity of serverless function invocations by an average of 13.25%. Cordingly et al. [56] introduces a proxy design where serverless function invocations are routed to the least carbon-intensive provider and region, showcasing substantial carbon intensity reductions. However, both studies ignore the carbon intensity of data transmission while also not offering an avenue to optimize larger-scale workflows since the studies focus on scheduling functions and ignoring dependencies or only being able to work with synchronous calls to single serverless functions. Research has primarily focused on the deployment problem from a cloud provider perspective and approached the problem solely as a resource scheduling problem, such as optimizing the workload in geographically distributed data centers [5], load balancing with renewables as part of the balancer [132, 133, 208], or exploiting otherwise wasted renewable energy by colocating data centers and connecting them over a dedicated network to get a "free lunch" [6]. The cloud provider perspective could bring the best results if feasible and implemented. However, due to a lack of tools for developers to communicate limitations such as region stickiness and end-to-end latency tolerances, more research into a global deployment framework is necessary to showcase the potential benefits of cross-regional migration.

While shifting simple workflows has been investigated in prior research, prior work has yet to adequately characterize the scope of more complex workflows required to answer the research question. Cloud workflows could benefit from geospatial shifting when looking back at the cloud workflows characterization (§2.3.1). Especially *short running* workflows with an *ad-hoc* time of execution, and that is *non-preemptible*. Other categories work as well, such as *long running*, *scheduled*, and *preemptible* workflows. However, geospatial shifting offers a unique opportunity to benefit from differences in grid intensities without waiting for the grid to improve but rather to move to less carbon-intensive regions.

Geospatial shifting is especially effective when the workflow is flexible concerning execution location. However, if these only apply to subparts of a workflow, then a fine-grained approach may still yield optimizations regarding carbon intensity. Shifting and workload migration also come at a cost that a framework attempting to provide a geospatial shifting solution must consider. While temporal shifting does not incur any additional cost by delaying or expediting a job, geospatial shifting incurs both a cost in latency between locations, monetary from egress to a different region, and, most importantly, carbon from the transmission. Every solution additionally creates overhead by the framework itself and by solving. Given the carbon swings, a static solution of finding the best deployment once and staying there is not feasible.

3.3.3 Limitations

While recent work in both temporal and geospatial shifting has highlighted the potential for reducing carbon intensity by shifting workflows to more favorable data center regions, there is still a gap for a comprehensive, cross-regional framework. A framework incorporating transmission and execution carbon while also being overhead-sensitive has yet to be built. The provider perspective offers many avenues for optimization; however, tolerances must be effectively communicated between developers and providers to overcome the current regional boundaries of optimization. We want to highlight the following limitations additionally that we encountered in the state of the art that we were aware of at the date of submission:

Ignoring Transmission Carbon: Currently, no solutions consider transmission carbon in calculating offloading decisions. Ignoring it is the wrong approach, even if it may look like the transmission carbon overhead is negligible. As shown in Section 2.6.3, the carbon intensity of transmissions exists and can not be ignored. Moreover, especially in geospatial shifting, taking carbon emissions from transmission into account is required.

Missing Comprehensive Solution for Cross-Regional Deployment: A framework that distributes workloads globally also needs to move application logic from one region to another. If the developer is responsible for moving the workflow, the framework will never be able to react automatically to new deployments. Additionally, deploying and migrating cross-regionally should be made as simple as possible so that the overhead caused by this functionality is kept as small as possible, both in the work required by the developer and in the computing necessary.

Non-Existing Framework for Cross-Regional Application Execution: The fine-grained geospatial distribution of complex workflows required to answer the research question adequately necessitates a solution that can handle cross-regional application execution, which has latency, interfaces, and reliability challenges. No other proposed framework currently

can handle this fine-granular execution, where most solutions propose a deployment and run in the most favorable region without being flexible enough to offload on a per-function level while maintaining defined dependencies.

Lack of accurate Carbon Forecasting: Forecasting carbon intensities in the relevant regions is paramount to the framework’s deployment decision quality. The existing solutions do not accurately account for carbon fluctuations and react passively to changes in carbon intensity, lacking the ability to forecast carbon intensities based on past data.

Single-Tolerance and Single-Objective Aware Deployment Algorithm: Any algorithm aiming at finding the best solution for the deployment problem of a cloud application needs to incorporate both tolerances and objectives and be able to handle multiple tolerances and objectives. Any solution that only focuses on one objective while ignoring tolerances or vice versa can not be called comprehensive and is most likely only usable for some developers’ use cases. While objectives usually attract significant interest, a multi-layered approach to tolerances, function, and application level is lacking. However, such a multi-layered approach is necessary for the complex applications deployed in today’s cloud environment.

Lacking Latency and Execution Predictions: Currently, no framework comprehensively incorporates existing research in latency or execution prediction into the scheduling algorithm. The fine line of offloading an application is between the gains of offloading the execution and the additional cost of further transmission. A solution that could model the end-to-end latency can claim to capture the complexities of applications.

Focusing on Singular Objective-Data Sources: None of the existing frameworks incorporate carbon, latency, and cost data in their deployment algorithm. Usually, the extensiveness is limited to one objective and its associated data. A comprehensive framework needs to be able to retrieve and model all three metrics that influence the optimal deployment of an application in a geospatial setting.

Not Minimizing Solution Overhead: Lastly, no existing solution considers the overhead incurred from running the algorithms and deployment when running the solution. A solution that does not consider its own cost might incur more carbon intensity than it saves. A dynamic and workflow-specific approach that considers workflow structure and workload is required.

Chapter 4

Framework Design

To answer the questions outlined in Chapter 1, while being aware of the factors required for a feasible solution (§3.2) and overcoming existing solutions limitations (§3.3.3), we developed the CARIBOU framework. CARIBOU is a features-rich framework that leverages carbon grid variations across regions for sustainable serverless workflow deployment and enables a large host of future research. We designed the framework to enable an initial answer to the subquestions, which would answer the main question, ”**Can we reduce the carbon emissions of complex cloud workflows by optimizing the geospatial deployment fine-granularly?**” while also being extensible for future improvements. We specifically decided to sketch out our solution as a framework rather than a system because a framework is defined as a set of tools designed to help developers build software applications while highlighting that the framework is extensible and changeable through its component-wise architecture. Following this paragraph, we will reiterate the relevant research questions and show where and how we answer them. Then, we will introduce the desired properties, provide a framework overview, and briefly highlight the component interactions. Lastly, we will delve deeper into the individual components.

The following research questions guided our work when designing a framework for carbon-aware geospatial shifting of serverless workflows:

- **Structure:** What application **structures** (§4.3) can be defined, and how does this structure relate to the geospatial deployment?
- **Metrics:** What **metrics** (§4.4) should a policy for geospatial workflow deployment rely on?
- **Policy: How** (§4.5) and **when** (§4.6) should the framework determine a new geospatial deployment that enables carbon reductions while remaining overhead and tolerance aware given the workflow constraints?
- **Enforcement:** How to materialize the above policy to geospatially **deploy** (§4.7) and to **invoke** (§4.8) the applications?

4.1 Desired Properties

A comprehensive framework should adopt the properties outlined in this section to effectively reduce the carbon intensity of cloud applications through geospatial deployment decisions. These properties outline our design approach to this framework and should act as solid guidelines without inhibiting further research by being too stringent.

- **Developer friendly** interface and framework usage for managed serverless applications that support the developer by providing a rich set of tools while minimizing the framework code footprint in the developer’s code.
- **Adaptive** to changing applications, workloads, and underlying data such as carbon. The main difference between carbon as an objective and the existing objectives of latency and cost is that while the prior two rely on application structure and workload, they are both static in their nature of underlying data. On the other hand, as we have shown, carbon fluctuates diurnally, regionally, and seasonally, introducing a new dimension of complexity. To be able to adapt to this added complexity becomes a major desired property of any comprehensive framework.
- **Modularity** of components. No single component should be non-replaceable; on the contrary, the components should interact over defined interfaces, and if any component becomes a bottleneck, replacing it should be relatively easy and fast.
- **Reliable** execution of the serverless application. The application execution should be the same even if the framework deployed the application geospatially over multiple regions.
- **Minimized overhead** of the carbon, cost, and runtime framework. Our framework needs to be lightweight and relatively cheap to run. It should also do a certain level of carbon accounting to self-adapt the strategy when running for a given application.

4.2 Framework Overview

In Figure 4.1, we present the overview of the framework we developed as the contribution to this thesis. CARIBOU as a framework contains three distinct spaces, where we define a space as a separate region of component interaction. Each space has a particular responsibility, two of which are "outside" facing and offer APIs to communicate with the framework. It is important to note that the components of the framework space themselves, which are more than an API, are actual entities invoked in a specific, to be effective low-carbon, region. We built the components and their interactions to be executed only when required, fully embracing the benefits of the serverless

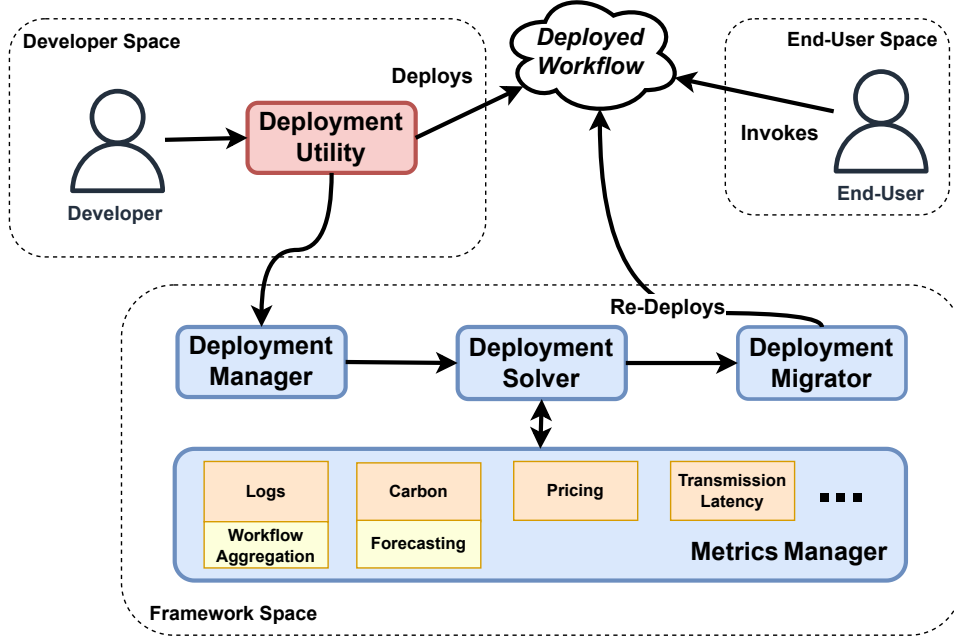


Figure 4.1: CARIBOU framework overview: We indicate the three spaces where the colors additionally highlight the difference between client-facing API (red) and the framework components (blue). Orange and yellow highlight data sources and data processing steps.

service paradigm to reduce the framework’s overhead. The three spaces and their respective responsibilities are:

- **Developer Space:** The interaction with application developers who must indicate tolerances and objectives and create the applications with all their dependencies. The interaction happens through the provided package to be imported and used as an API to define and deploy workflows.
- **End-User Space:** This is the end-user interface part of the provided Python package. We simplify the prerequisites for developers by offering endpoints to route end-user requests correctly.
- **Framework space:** The framework components that facilitate the tasks to optimize all workflows currently deployed to the framework geospatially. With its components in this space, the framework provides initial implementations that answer the research questions and provide the infrastructure for further research.

In the following sections, we will give more insights into the design of the different components while keeping the bigger picture in mind. Some sections do not directly have a representation in the framework overview

as they represent internal concepts such as the workflow model (§4.3) or represent actions such as workflow executions (§4.8). The components communicate asynchronously using distributed key-value stores that define clear interfaces. We decided on this mode of component interaction to decouple the individual components. Specific actions and triggers asynchronously execute all components when necessary or sensible. None of the tasks in the framework space is on the hot path of application execution. The framework optimizations should not impact the end-user when running in the background to determine a new optimal deployment of the workflow invoked. The components and respective sections are:

- Metrics Manager (§4.4)
- Deployment Solver (§4.5)
- Deployment Manager (§4.6)
- Deployment Utility (§4.7.1)
- Deployment Migrator (§4.7.2)
- Invocations (§4.8)

4.3 Workflow Model

As we previously introduced, complex workflows combine individual serverless functions. There is no inherent upper limit on the number of functions; the lower limit is one. We will abstract all these variants of workflows to enable one comprehensive term for both single functions and complex, multi-stage workflows. The *workflow model* defines how the framework internally represents the workflows as defined by the developers. It additionally defines the possible scope of deployable workflows using our framework. The developers implicitly indicate the workflow structure in their code by using the API we will introduce in Section 5.4.1. The *Deployment Utility* then generates the workflow by static code analysis (§5.5.4). We require the workflow model for both the *Deployment Solver* and *Metrics Manager*, as well as for invocations, but only on the framework side. The model reflects the internal representation, whereas the developers only implicitly define the structure of their workflows. We decided on this split to reduce the mental work required by the developer to annotate in code and define in configuration files and to allow developers to continue their work inside the same programming language without switching between the context of workflow definition and workflow development. The solver requires the model to generate new deployments, the *Metrics Manager* to return the metrics for a workflow deployment, and invocations to understand the current invocation location inside the DAG and where to route a subsequent call.

Existing application modeling solutions, some previously outlined (§2.5.1), either were too domain-specific as in the example of "Ray" [155], for static

deployment such as the model for "SkyPilot" or "TOSCA" [216, 220] or lacked support for adequately modeling conditional data transmission edges such as in "AFCL" [186, 187] for our purposes. To enable a robust set of workflows while incorporating concepts such as conditional invocations and dynamic changes to the metric data while remaining flexible for future changes, we introduce the *workflow model*. We define a *workflow* as a DAG $G = (N, E)$, where N is a set of nodes and E is a set of edges. The edge $e_{ij} \in E$ represents execution dependencies between nodes $n_i, n_j \in N$, inspired by the modeling of complex workflows in high-performance computing [41]. Each node $n_i \in N$ represents an execution stage in the application where computation happens and represents a serverless function invocation. Figure 4.2 outlines an example workflow DAG with five nodes and five edges.

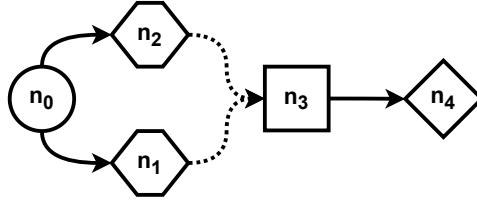


Figure 4.2: Example workflow DAG with nodes n_0, n_1, n_2, n_3, n_4 and corresponding edges. The edges e_{13} and e_{23} are conditional, indicated by dotted lines.

4.3.1 Source Code Functions

Each source code function (s_k) can be associated with multiple execution stages in the workflow. We represent each execution stage as a separate node (n_i) in the DAG to make the workflow representation acyclic. The model itself does not restrict the source code but will be used to determine the DAG; thus, Section 5.4.1 outlines how the developer uses the provided API functions to indicate the DAG structure in the source code. The corresponding source code functions for the DAG introduced in Figure 4.2 are outlined in Figure 4.3, showcasing that four source code functions can lead to a DAG with a different number of nodes.

4.3.2 Deployment

Definition 3. Deployment: Outlines the geospatial deployment of every execution stage of a workflow.

Each source code function $s_k \in S$ is potentially deployed in R regions, where R is defined by the cloud service providers currently supported by



Figure 4.3: Source code functions corresponding to the DAG introduced in the previous subsection. The source code represents no DAG structure but presents the basic underlying logic blocks. These blocks are source code from which the *Deployment Utility* generates the DAG structure through source code analysis.

the framework. The developer can restrict R at each source code and workflow level. Every node (n_i) is mapped to a source code function (s_k) where the mapping is many to one, giving the mapping $R_{allowed} : N \mapsto R$ that returns the allowed regions, given node n_i . Each node (n_i) has an associated deployment region $r \in R_{allowed}(n_i)$. Let $\psi : N \mapsto R_{allowed}(N)$ be the mapping of DAG nodes to regions. When invoking the workflow given the deployment indicated by ψ , the source code (s_k) associated with this node (n_i) is invoked in this region. This mapping is referred to as a *deployment* in this thesis and is always part of a larger construct called a *Deployment Plan (DP)* (§5.3). This deployment is required to solve twofold problems:

1. at the invocation of an application, each node (n_i) needs to know where (r) to route subsequent invocations to,
2. When the *Deployment Solver* generates a new deployment, the *Deployment Migrator* needs to know the change to adapt any source code deployment to potential new regions correctly.

4.3.3 Directed Edges

Every node (n_i) has a set of incoming and a set of outgoing edges, $E_{in}(n_i)$ and $E_{out}(n_i)$, respectively. Every edge in a DAG has a direction. If $E_{in}(n_i) = \emptyset$, the node is a *start node*. We only consider workflows with precisely one start node since this is the most common structure in workflow DAGs. Additionally, if $|E_{in}(n_i)| > 1$, the node is a *synchronization node*. Any edge e_{ij} is annotated with a boolean value that captures if the edge has been invoked by node n_i , giving us mapping $C : E \mapsto \{0, 1\}$.

In the example shown in Fig. 4.2, there are five directed edges, two conditional, indicated by the dotted line. At an invocation of this workflow, the values for C will be determined dynamically based on the actual variable values, where by default $C(e_{ij}) = \emptyset$ if an edge is not yet reached by node n_i , $C(e_{ij}) = 1$ if reached and conditionally invoked by n_i , and $C(e_{ij}) = 0$ if reached but conditionally not-invoked by n_i .

4.3.4 Paths

A DAG implicitly introduces the concept of a path. A path, $Path(n_i, n_j)$, is defined as two nodes n_i and n_j connected by a sequence of directed edges that allows traversal from n_i to n_j . The start node has a path to every node of the DAG. In Figure 4.4, we highlight a specific path from node n_2 to n_4 .

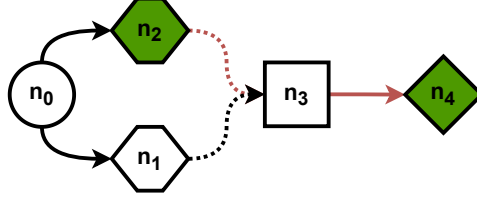


Figure 4.4: Highlighting the path from n_2 to n_4 .

4.3.5 Start Node

The start node serves two purposes: it is the sole entry point for invocations into the DAG and determines the initial incoming edge at an invocation since its location is trivially known. The implications of this will become apparent when executing a workflow where the current location of a successor node in the DAG is passed along.

4.3.6 Synchronization Node

When invoking an edge e_{ij} to a synchronization node n_j , the predecessor node n_i is required to atomically update an annotation associated with $C(e_{ij}) = 0$ if reached and not invoked or $C(e_{ij}) = 1$ if reached and invoked. After updating the annotation, n_i must check if the condition for executing n_j is True, and if so, execute n_j ; otherwise, do nothing. The condition for executing the synchronization node is:

$$(\forall e_{ij} \in E_{in}(n_j), C(e_{ij}) \neq \emptyset) \wedge (\exists e_{kj} \in E_{in}(n_j) : C(e_{kj}) = 1) \quad (4.1)$$

In Figure 4.2 the synchronization node is n_3 since $E_{in}(n_3) = \{n_1, n_2\}$. The node n_3 is thus dependent on both predecessors reaching the edges e_{13} and e_{23} where at least one of them needs to be invoked ($C(e_{i3}) = 1 \forall i \in 1, 2$).

4.3.7 Conditional Edge

The above semantics enable conditional branches needed to support conditional DAGs in serverless [136, 188]. Handling the case where all incoming edges to a node are unconditional, i.e., always taken ($\forall e_{ij} \in E_{in}(n_j), C(e_{ij}) = 1$), is straightforward: the last predecessor of n_j invokes the edge and thus executes the node when all other predecessors also marked their respective

edges as invoked. For a conditional branch (e_{ij}), a predecessor node (n_i) marks $C(e_{ij}) = 0$, meaning edge e_{ij} is non-invoked but reached when the trigger condition of the edge is not satisfied. In this case:

1. for all paths between n_j to any synchronization node n_s , set $C(e_{ts}) = 0$ if n_t is on $Path(n_j, n_s)$ and has edge e_{ts} ;
2. n_i checks if n_s now fulfills the condition, if yes execute n_s .

The above-discussed logic is required to make sure that conditionally not invoking a node does not block any successor synchronization node. To illustrate the abovementioned logic, we will introduce a slightly more complex workflow, including two synchronization nodes and chained conditional edges.

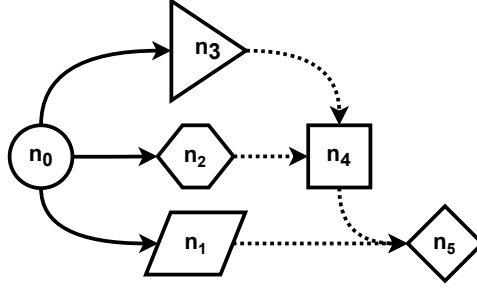


Figure 4.5: More complex example workflow DAG with nodes $n_0, n_1, n_2, n_3, n_4, n_5$ and corresponding edges. Nodes n_4 and n_5 are synchronization nodes. The edges e_{15}, e_{24}, e_{45} are conditional.

In Figure 4.5, we see two chained synchronization nodes with additional conditional edges $e_{15}, e_{24}, e_{34}, e_{45}$. The execution order is $n_0, n_1, n_3, n_2, n_4, n_5$ where $C(e_{15}) = 1, C(e_{24}) = 0, C(e_{34}) = 0, C(e_{45}) = 0$. Responsible for invoking both n_4 and n_5 is n_2 , since n_2 is the last to reach and annotate the edge e_{24} it needs to check the condition from Equation 4.1, which will result in false and n_2 does not execute n_4 . Additionally, since there is $Path(n_2, n_5)$, n_2 also needs to set $C(e_{45}) = 0$ which will lead n_5 to fulfill the condition of Equation 4.1, meaning n_2 will need to invoke n_5 .

4.4 Metrics

When given a DAG as well as date and time, the **Metrics Manager (MM)**, a component activated upon triggers for data collection and modeling, must be able to estimate the metrics of this specific DAG. The *Deployment Solver* uses the modeled metrics to determine an optimal deployment (§4.5) where it requests the metrics from the MM by passing a deployment for a workflow DAG as well as the hour of the day, which is required for more fine-grained solved deployments, which can be up to hourly granularity. This hourly

granularity is required to capture the diurnal differences in carbon intensities of electricity grids. Metrics modeling requires retrieving metrics values based on the collected empirical data for each node and edge in a graph for all necessary metrics. We kept the MM extensible, so adding additional metrics is a relatively low effort. Each metric added to the framework needs to be able to provide a function for execution (node) and transmission (edge) and may require any number of internal or external data sources to model said metrics according to the function. We introduce the specific data sources and aggregations in the implementation sections (§5.5.7). At the same time, we focus here on how the manager models the metrics workflow end-to-end latency, carbon, and cost.

4.4.1 End-to-End Modeling

Estimating the end-to-end latency, cost, and carbon emissions of complex conditional application DAG is a challenging task [38] warranting more research. To reduce the complexity, we decided to model the end-to-end metrics using Monte Carlo simulations [150]. Previous work has done this similarly to estimate the end-to-end latency of complex DAGs [35, 71, 128, 129].

The MM bases the Monte Carlo simulation on past workflow invocations. These provide the data for per-node execution runtimes for the regions where the workflow’s source code functions have been deployed and invoked. Additionally, the past workflow invocation data provides per-edge conditional probabilities and latencies between nodes and between regions. The data is only available for the regions where the corresponding source code functions were deployed to and invoked, which we group by transmission size where the relative weights of sizes are maintained. In addition to edges between all nodes, the simulation also considers the transmission latency between the end user and the start node, which is maintained as an incoming edge in the DAG with a corresponding list of transmission sizes and corresponding latencies. Each simulation iteration generates a version of the DAG, where the edges are activated or deactivated based on random sampling of the conditional edge probabilities. The MM models the metrics of this DAG version for both the nodes and edges:

- **Nodes:** In the simulations of the metrics of node executions, all metrics are related to the execution runtime. When simulating, the manager thus samples a random execution runtime, which allows it to calculate the corresponding cost and carbon of this node’s execution according to the later introduced functions.
- **Edges:** The edges metrics depend primarily on the data size of the transfer since this also indicates the latency. The MM samples a transfer size to retrieve transmission latency, cost, and carbon according to the functions when simulating the edges.

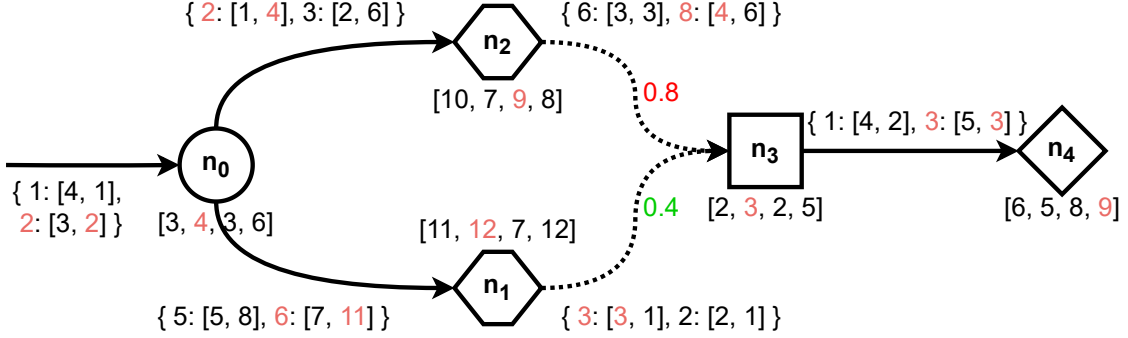


Figure 4.6: Example of a Monte Carlo simulation iteration where MM samples the red highlighted numbers in this iteration. Additionally, one conditional edge was activated (green) and one not (red). The resulting DAG has an end-to-end runtime of 47, and the MM uses the values of all executed nodes and edges for carbon and cost modeling.

In Figure 4.6, we give an example of a Monte Carlo simulation iteration where the data shown is from previous workflow runs. A different deployment might mean different data on edges or nodes. When simulating the DAG for one Monte Carlo iteration, the MM sums up the individual cost and carbon as the end-to-end carbon and cost. Similarly, the manager determines the end-to-end latency by identifying the critical path of the current DAG version.

The Monte Carlo simulations are conducted in batches of 1,000, continuing until the confidence interval of 95% has a relative width of less than 5% for end-to-end latency, cost, and carbon footprint or until a maximum of 20,000 samples is reached. For these distributions, the 95th percentile is the "tail case" used to determine tolerance violations for satisfying the QoS (§4.5.1), and the mean represents the "average case" used for deployment objective ordering (§4.5.2).

4.4.2 Carbon

Since carbon reduction is the primary objective of our framework, we require a comprehensive data model for carbon that goes further than existing solutions. As we previously introduced, many solutions exist for execution carbon (§2.3.2); however, transmission carbon is underdeveloped (§2.6.3). We will introduce our data modeling regarding both, which will entail concretizing the formulas introduced in the background on modeling the carbon emissions of cloud application executions (§2.3.2) and the background on modeling data transmission carbon emissions (§2.6.3). We base the carbon intensity data on the ElectricityMaps API [73], retrieving the carbon data

for the past seven days and forecasting for the next 24 hours in an hourly granularity, where the hour used for modeling is determined by the hour for that the solver is requesting the metrics. Forecasting is required since carbon data does not remain static. Predicting the carbon data for the next 24 hours is essential for good metric estimation.

Execution Carbon

We model execution carbon with Equation 4.2, including compute and memory, and we omit the contribution of storage as it has shown to be negligible [124]. For this initial version of the framework, we considered IO's carbon intensity to be part of the transmission carbon. How and where to correctly model this remains an open research question. The power usage effectiveness (PUE) we decided on is 1.11, the average of the 1.07-1.15 range reported for AWS datacenters [7].

$$Carbon_{ex} = I_{grid} \times (E_{comp} + E_{mem}) \times PUE \quad (4.2)$$

We calculate the energy usage of memory with Equation 4.3. For memory energy usage, we estimate P_{mem} to be 3.725e-4 kW/GB, which is in line with previous research [114, 124].

$$E_{mem} = P_{mem} \times mem \times \frac{t}{3600} \quad (4.3)$$

Subsequently, the energy usage of computing (CPU) can be estimated using Equation 4.4. The energy usage of compute can be estimated based on the number of CPU, or vCPU in the specific case of serverless functions, which at AWS, for example, is a function of requested memory ($n_{vCPU} = \frac{mem}{1769}$) [24].

$$E_{comp} = P_{vCPU} \times n_{vCPU} \times \frac{t}{3600} \quad (4.4)$$

We model the energy usage of IO (E_{IO}) similarly to transmission carbon (§2.6.3).

In our model, we calculate the power used by the CPU with Equation 4.5. Previous research has either used linear processor utilization-based power model [65, 75] to model the power usage of a CPU or estimated the power usage by thermal design power (TDP) [123, 124, 217]. We decided to use the linear processor utilization-based model because it provides the most accurate energy usage estimation based on the public data we can access. The average vCPU utilization [51, 212], which varies between each workflow stage, can be obtained using the extended logging functionality AWS Insights. This logging functionality reports the average CPU time. The power draw per core in AWS datacenters required for the linear utilization-based power model is estimated to be 7.5e-4 kW when idle (P_{min}) and 3.5e-3 kW

when fully utilized (P_{max}) [207]. We summarize the variables used throughout this section in Table 4.1.

$$P_{\text{vCPU}} = P_{\text{min}} + \frac{\text{cpu_total_time}}{t \times n_{\text{vCPU}}} \times (P_{\text{max}} - P_{\text{min}}) \quad (4.5)$$

Parameter	Description	Units
$Carbon_{\text{ex}}$	Carbon Footprint of Function Execution	g CO_{2e}
I_{grid}	Carbon intensity of the Power Grid	g CO_{2e} per kWh
E_{comp}	Energy Consumption from Compute	kWh
E_{mem}	Energy Consumption from Memory	kWh
E_{IO}	Energy Consumption from IO	kWh
PUE	Power Usage Effectiveness	Ratio
P_{mem}	Power consumption per GB of memory	kW per GB
mem	Amount of Requested Memory of Function	GB
t	Function Execution Duration	Seconds
P_{vCPU}	Power consumption per vCPU	kW per vCPU
n_{vCPU}	Number of CPUs	vCPU
P_{max}	Average Power per vCPU when server is Idle	kW
P_{min}	Average Power per vCPU when server is fully utilized	kW
cpu_total_time	Time spent using CPU	Seconds

Table 4.1: Units and Descriptions for modeling cloud application carbon emissions.

Transmission Carbon

We present a transmission carbon model, as seen in Equation 4.6, based on recent research [36, 78, 194].

$$Carbon_{\text{trans}} = I_{\text{route}} \times EF_{\text{trans}} \times S \quad (4.6)$$

As previously outlined, the estimates for EF_{trans} vary greatly over past conducted studies and years as outlined in Table 2.4. Additionally, the energy efficiency of data transfer approximately doubles every two years [30, 36]. Therefore, we conservatively extrapolate EF_{trans} based on past studies to be in the 0.001 to 0.005 kWh/GB range. To provide a confident model based on research, it makes sense to consider both the best and worst cases regarding the transmission carbon overhead of offloading. The best case is 0.001 kWh/GB, which applies when the data transfer is within a region or cross-regional. Thus, staying in the home region incurs a similar transmission carbon overhead to moving somewhere else, depending on I_{route} . On the other hand, the worst case is 0.005 kWh/GB and incurring no carbon overhead when staying in the same region, leading to the most minor incentive for offloading due to expensive transmissions when targeting carbon as

an objective for offloading. We calculate I_{route} based on a segment approach where the route between two data centers is split into smaller segments, and the carbon intensity is averaged over the segments. Table 4.2 additionally lists the variables used in the function.

Definition 4. The **Home Region** is defined as the developer-defined default location that precedes the framework’s dynamic deployments and acts as a fallback and a baseline.

Parameter	Description	Units
$Carbon_{\text{trans}}$	Transmission Carbon footprint	g CO_{2e}
S	Size of Data Transfer	GB
I_{route}	Carbon intensity of Route	g CO_{2e} per kWh
EF_{trans}	Energy consumption of the transfer	kWh per GB

Table 4.2: Units and Descriptions for modeling data transmission carbon emissions.

4.4.3 Cost

Function invocations cause costs both at execution and transmission. We outline the sources and corresponding functions below. Many providers offer so-called free-tier offerings, a certain number of resource usages that do not incur costs. We omit the AWS free tier [17] in our calculation as our framework currently does not support the fine-granular accounting for deployed workflows required to model this dimension accurately.

Execution Cost

The execution cost is modeled based on execution time (t), configured memory size (mem) in megabytes, and a fixed per-invocation fee [18]. Additionally, a workflow incurs cost at the start of each invocation through one additional DynamoDB access introduced to facilitate geospatial shifting through retrieving the current deployment, which costs depending on where the initial function is located [11] and is a function of the deployment size. We omit this additional function from the Equation 4.7 for brevity.

$$Cost_{\text{ex}} = c_{\text{invocation}} + (t * mem * c_{\text{lambda}}) \quad (4.7)$$

Transmission Cost

We calculate the transmission cost from the associated outbound data transfer (egress), the same as data transfer from AWS EC2 to the internet [12], differing from region to region. We additionally consider the transmission costs incurred from cross-regional SNS messaging for both stage invocation and synchronization nodes [14], again varying by region where the data originated. Both values are 0 if the transmission is within a region from one AWS service to another. This gives us the following Equation 4.8:

$$Cost_{\text{tran}} = (c_{\text{egress}} + c_{\text{sns}}) \times S \quad (4.8)$$

4.5 Determining Optimal Deployments

When given a workflow DAG and the per node allowed regions as well as the interface to the *Metrics Manager* to retrieve values of the metrics of any deployment, our framework component, the **Deployment Solver (DS)**, needs to be able to generate a deployment, i.e., where to deploy each workflow node. The DS bases the optimal deployment determination on the framework’s *Metrics Manager* based on workflow information, namely structure, objectives, and tolerances, as well as on the metrics calculated on a workflow level. We decided on this clear split of metrics (data model) and determining the deployment since one should maintain the desired property of independent, changeable components. For a workflow with execution stages N and available regions R , the search space is $|R|^{|N|}$. The search space grows exponentially with increasing number of nodes or regions. Node or workflow level tolerances can narrow this search space, but only to a certain extent. Raw computing is not a viable solution for this problem since one of our desired properties is that the framework components, of which the DS certainly is one, have a small carbon footprint. We attempted to keep the computational requirements of the determination itself to a minimum. A simple approach to tame the search space is to limit the deployment of all DAG nodes to the same region, reducing the solver complexity to $O(|R|)$. This approach was outlined in multiple previous works [48, 144] and renders reasonable results. However, this approach can be globally suboptimal for not (1) deploying nodes off the critical path to remote, low-carbon regions and (2) deploying nodes without data compliance requirements to foreign, low-carbon regions, effectively limiting the options by not being able to deploy fine-grained.

Implementing the solver using a breadth-first search (BFS) strategy proved intractable and resource-inefficient. To both keep our initial solution simple but powerful, we decided on a *Heuristic-biased Stochastic Sampling (HBSS)* algorithm [47, 100], outlined in detail in Section 5.5.3. It employs heuristics to explore new deployments and tests for improvement, leveraging

the information obtained as a region bias. The heuristic orients itself after the primary objective of the specific workflow.

The DS thus returns a "best" deployment when triggered, where best is indicated by the developer while observing the tolerances set by the developer.

4.5.1 Tolerance Violations

The implemented deployment algorithm may generate many deployments for the DAG. A generated deployment solution that violates a tolerance according to the modeled "tail case" (95th percentile of Monte Carlo sampled distribution) is removed and not considered. Tolerances can be relative or absolute, and the DS compares relative tolerances against the "tail case" of the home region deployment. An absolute tolerance can not invalidate a home region deployment; the home region deployment must always be valid regarding region constraints. Otherwise, the framework prohibits the deployment of that workflow. Unrealistic absolute tolerances lead to only deploying the workflow in the home region. To model this "tail case" empowers the DS and, thus, the framework to support varying tolerances for all metrics without theoretically violating the objectives. However, the tolerance violations are only based on modeling and are not actively monitored once the DS selects a deployment. If violations occur due to relative pressure or component failures at the cloud service provider, we currently have no way of reacting ad hoc. A selected deployment that violates a deployment will be replaced at the next deployment since the newly collected data will make the modeling more accurate and reflect the reality of the situation.

4.5.2 Selecting the Optimal Deployment

The selection of the optimal deployment depends on the information the developer provides regarding their objectives. The developer can indicate a priority ordering for the current carbon, latency, and cost-supported objectives. This priority order will lexicographically order the resulting deployments. The "best," according to the DS, will be selected and automatically deployed by the *Deployment Migrator*. The newly selected deployment will be stored as part of the workflow deployment plan (DP) in the DP staging area, used to communicate with the *Deployment Migrator* (§4.7.2) to inform of new deployment potentially requiring a migration of the workflow.

4.6 Dynamic Triggering of Policy Determination

Now that we know how to determine a policy, the next question is when to trigger such a determination. Since carbon data is available in the granularity of an hour forecasted based on the past seven days, a naive approach

of triggering every hour would be wasteful of resources. On the other hand, triggering with just enough frequency might avoid missing metrics and workflow invocation load shifts, such as transmission sizes and execution times. CARIBOU has to offer a dynamic approach to fulfill the desired properties that consider the underlying data while acknowledging workflow information, such as the number of invocations. We implemented this dynamic triggering in the **Deployment Manager (DM)**, the only component fully aware of all deployed workflows and with a per-workflow state it manages.

4.6.1 Framework Overhead

CARIBOU incurs overheads primarily from deployment generation (§4.5, compute heavy), metrics collection and deployment migration (§4.4 and §4.7.2, data transmission heavy). The framework’s deployment region and the frequency of generating deployments influence these overheads. The carbon savings gains must be more significant than these carbon overhead factors to offer net gains.

4.6.2 Framework Gains

As outlined, the carbon differential between the home and potential offload regions and the workflow traffic volume affect the potential carbon savings. A workflow placed in a cluster of viable regions with significant carbon intensity differences can save more than a workflow without viable offloading locations. Similarly, an application requires a certain number of invocations to warrant offloading, and the DM can not ignore this break-even point; otherwise, the overhead per determination and migration might offset the gains.

4.6.3 Dynamic Control Mechanism

A dynamic control mechanism to trigger the deployment determination is required to balance the two effects of gains and overhead. To balance generating new, potentially more optimal deployments and the framework overhead, CARIBOU uses a per workflow token bucket algorithm to self-regulate the deployment generation frequency. The bucket’s inflow is a function of workflow invocations since the last check and potential offload gains. Each workflow will undergo regular token checks. The DM adapts the frequency of the token checks to the bucket’s underflow or overflow in the past period. Suppose a token check has passed, but the DM did not instruct the *Deployment Solver* to determine a new deployment. In that case, the token check time will be adjusted later, depending on invocations in the past period. If the token inflow rate increases, the time will be sooner, if the rate decreases later. We opted for a conservative approach to not cause excess overhead in the case of, for example, an infrequently invoked workflow

where a metric changed significantly. Tokens represent the carbon budget for framework overhead, and the framework consumes them when the bucket contains enough tokens for a deployment solution. The cost, or outflow of tokens, of a deployment generation is determined based on workflow structure and the framework’s region carbon intensity.

4.6.4 Deployment Retirement

A deployment maintains validity for a specific time based on confidence in the metrics data used to determine that deployment. We designed that every framework-determined deployment will eventually be retired to prevent missing out on significant metric shifts. At that point, the invocations will all be routed back to the home region deployment. The active retirement of deployments is a conservative design approach. In uncertainty, we default to the home region deployment as if our framework were not in the loop. Since a deployment might cause tolerance violations based on bad modeling, the DM will likely replace that faulty deployment at the next token check. In a future iteration, we envision the framework monitoring the executions more actively and retiring violating workflows proactively.

4.7 Application Deployment and Re-Deployment

Any application that wants to benefit from geospatially distributed requires the functionality to be deployed and re-deployed cross-regionally. CARIBOU needs to dynamically adjust the deployment according to the carbon intensities without requiring manual input to deploy functions to every region. It must provide a utility for deployment and a component to re-deploy functions to new regions without additional developer effort. In the following subsection, we will highlight the design decisions in both deployment and re-deployment.

4.7.1 Application Deployment

Application deployment should happen transparently to enable auditing but abstracted away from the developer to reduce the strain from deploying complex workflows with all their dependencies and ensure the workflow packages have the expected structure for other parts of the framework. Additionally, deployment should be flexible regarding dependencies and enable the developer to have the same, if not an enhanced, experience compared to the vanilla experience provided by a cloud service provider. We decided to facilitate application deployment by creating the **Deployment Utility (DU)**, part of the framework package imported into a project like any other third-party dependency offering a CLI interface. Our design is targeted towards our language of choice, Python, since the Python ecosystem

is large and both AWS and other providers for serverless functions support the language. We will indicate the Python package as `caribou` to clarify the difference between the developer-facing package and framework. Extending the framework to additional languages requires relatively little effort. A future maintainer must only port the Python-specific API implementation to enable additional languages.

A deployed workflow is assigned a unique ID based on the workflow name and version as defined by the developer. If an ID is already deployed to the framework, the developer is prohibited from overwriting it and must either remove the duplicate or change the name or version. We introduced this versioning to allow for the possibility of deploying multiple versions of the same workflow while allowing for a transparent versioning system. Updating a workflow is not possible. The only option to update a workflow is to remove the old version and deploy the new version with the same name and version.

Definition 5. The **workflow ID** assigned is unique per framework and a combination of name and version.

The deployment of a specific workflow consists of the following steps:

1. A workflow is created by the developer with the required interface (§5.4.1) where the developer also indicates tolerances, objectives, home region, and additional configurations,
2. The workflow developer invokes the DU,
3. The utility packages up the client’s code into a package and pushes it to the home region
4. The DU creates the function and instantiates all auxiliary services required in the home region,
5. The utility stores the home region deployment in the workflow’s deployment information entry in the key-value store.

The home region deployment is always maintained. After these steps, an application is officially deployed and invokable through the client (§5.5.6), and the DU informs the *Deployment Manager* about this new workflow by writing into the respective state table at the distributed key-value store.

4.7.2 Application Re-Deployment

Automating migrations to new regions is necessary to maximize carbon emission reductions without burdening developers. The **Deployment Migrator (DMi)** solves this technical challenge by regularly checking whether the migration of a function to a new region based on the latest deployment is required. The component is invoked and runs as a serverless function regularly. The migrator checks if a new deployment is required according

to the *Deployment Solver*. Once our framework has determined a new deployment, the currently deployed regions for all source code functions might not equal the newly determined ones. In this case, the DMi will determine the difference and migrate the corresponding source code functions. If a re-deployment is warranted, the migrator replays the process outlined in **Application Deployment**, except step (3). In the third step, the migrator copies the source code package instead of repackaging it. A new deployment is activated once the migration has succeeded for all node deployment regions. Activated means that it is retrievable upon an invocation of a workflow from the corresponding distributed key-value store. If any function re-deployment fails, the framework defaults to the home region deployment, preventing invocations from routing through an invalid deployment. This process catches any potential issue with deployment, including region unavailability due to increased traffic. The DMi will then retrieve and retry the non-activated deployment later or, when a new deployment is solved without the unavailable region, replace it with a newly solved deployment initiated by the *Deployment Manager*.

4.8 Cross-Regional Workflow Execution

CARIBOU must be able to handle and enforce cross-regional workflow execution and traffic routing. Executing a geospatially distributed application should have the same interface and result as executing an application deployed in the same region. No possible deployment should necessitate changing the source code. To our knowledge, no other public framework enables task-level, cross-regional workflow execution. Additionally, the challenge of control flow invocations and synchronization nodes, introduced in Section 4.3, must be solved to enable a well-rounded framework. For this purpose, we made two fundamental design decisions:

1. The `caribou` provided function and workflow wrapper hides the complexity of the cross-regional execution
2. Application invocations are glued together using pub/sub messaging.

Following, we will outline the process that happens when invoking a workflow (§4.8.1), what our process means for the possible message payload (§4.8.2), how the invocation is routed throughout the workflow (§4.8.3), what particular logic the synchronization nodes require during an invocation (§4.8.4) and lastly how pub/sub glues together the invocations (§4.8.5).

4.8.1 Invoking a Workflow

As will be explained in Section 5.4.1, every source code deployed in the CARIBOU framework has to import the `caribou` package and wrap the workflow as well as every function in a wrapper. This wrapper hides away the logis-

tics of function invocations and routing. End-user workflow invocations can be routed directly to the home-region deployed source code function of the start node or through a proxy provided by the developer that calls the `caribou`-provided CLI (§5.4.1) functionality that uses the **Invocation Client (IC)** (§5.5.6) to kick off a workflow invocation. In any case, the wrapper routes 10% of the workflow invocations to the home region for performance benchmarking and continuous metric collection. The *Metrics Manager* collects metrics in every region. However, maintaining up-to-date information on the home region is critical for good deployment decisions since the home region represents the baseline for defined tolerances. The initial node invocation or the IC fetches the deployment plan (DP) containing the current deployment from the distributed key-value store on a workflow invocation. The retrieval of the deployment by the start node or IC is indicated on the left of Figure 4.7. The wrapper defaults to choosing the home region deployment unless a non-retired framework-determined deployment exists.

4.8.2 Message Payload

The payload size of an invocation is limited due to the pub/sub model utilized for invoking successors, which is SNS in our case that limits the payload to 256 KB [15]. Additionally, since the wrapper stores this data for a synchronization node in the distributed key-value store, this adds another total size limit. We opted for DynamoDB, whose item size limit is 400KB [27]. Recent studies have shown that the invocation of serverless DAGs usually does not pass raw data but so-called intermediate data. Intermediate data is small-scale information passed between DAG nodes to communicate data through remote storage [142, 143]. The wrapper then uses the intermediate data to retrieve the actual data from blob storage, such as S3 in AWS or other storage solutions.

4.8.3 Traffic Routing

The retrieved DP contains the deployment, which determines the current workflow traffic routing by mapping each node to the deployment region. Each node determines the deployment information of all successors, the workflow DAG structure, and its location in the DAG based on the DP. The value fetched from the distributed key-value store provides all this information except the current location. Determining the current DAG location during an invocation is a challenge that needs to be solved. As previously discussed, the start node is the only node able to determine its location intuitively from the DP without any additional logic since the location noted in the DP will be the start node by default. The function wrapper invoking the successor must pass on the location information, which can be calculated based on its location and the successor index. From the start

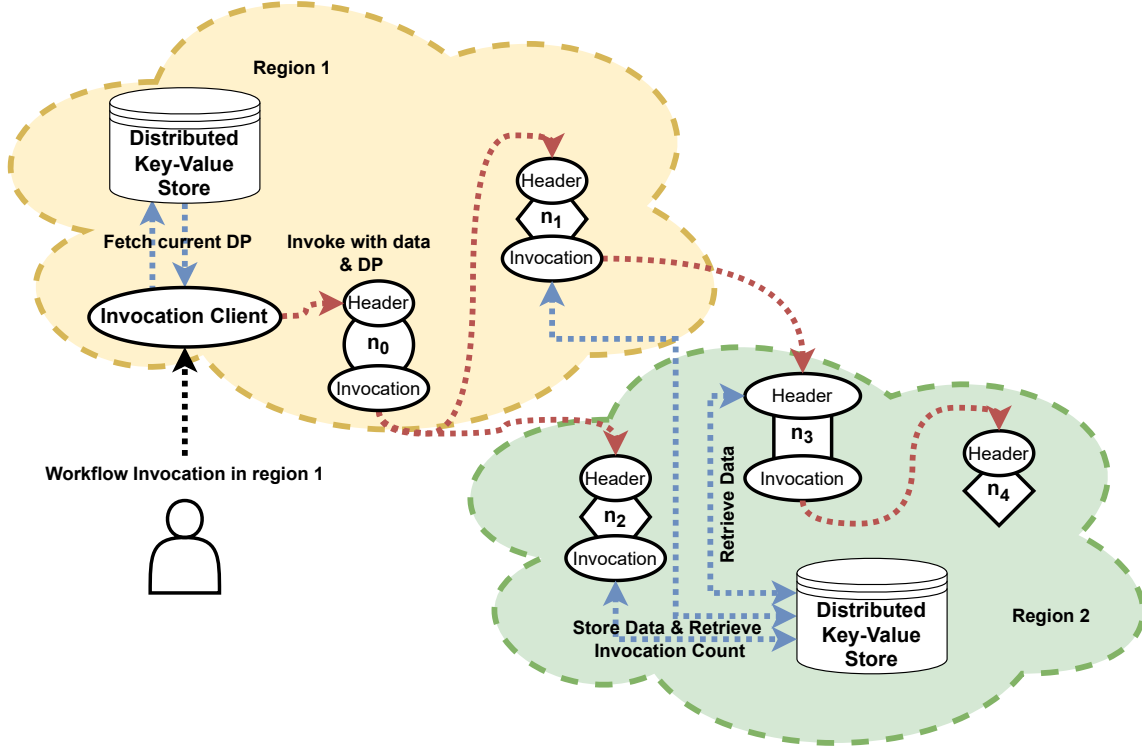


Figure 4.7: Workflow invocation: The IC retrieves the DP from the distributed key-value store and then invokes the first node in the corresponding region (region 1). Subsequent calls are routed by the corresponding workflow and function wrappers accordingly, where one part of the workflow is deployed in region 2. The synchronization node logic is facilitated by its predecessors, which synchronize using the distributed key-value store and atomic updates.

node onwards, any subsequent node location is determined inductively by passing that information to its successor. When invoking a successor, the decorator provided in the wrapper copies the DP and notes the location of the successor, piggybacking it on the invocation's intermediate data sent to the successor. The successor handler decorator retrieves the DP, and the data is then passed to the defined function, ensuring it receives the data as forwarded by its predecessor. This process is visualized by an example in Figure 4.7, where the function wrapper at the start node retrieves the deployment mapping and then invokes the successor in the correct regions, passing on information such as the successor DAG location.

4.8.4 Synchronization Nodes

For synchronization nodes, the logic outlined in Section 4.3.6 needs to materialize during a workflow invocation. All predecessors of a synchronization node store first the intermediate data and then the annotation, which marks whether a specific predecessor conditionally invoked the synchronization node, to a table each in a distributed key-value store in the synchronization nodes region. The tables are located in the synchronization nodes region to reduce the latency of retrieving the information during the synchronization nodes invocation. When updating the annotation table, the predecessor will receive the current state of the annotations. The annotation write update happens atomically [26], so the invocation count retrieved reflects the number of invocations at this node's update. The current annotation state is then used by the predecessor using the DAG information to decide if all the required annotations are present and, if so, invoke the synchronization node. The synchronization node then retrieves the intermediate data from the distributed key-value store. This can be seen in Figure 4.7 between n_1 , n_2 , and n_3 where the distributed key-value store holding the intermediate data and annotations is in region 2 because the synchronization node is in region 2.

4.8.5 Pub/sub for invoking a successor

The wrapper invokes function calls by posting a message to the respective function's publisher/subscriber (pub/sub) messaging topic. By posting the message to the correct region and topic that is part of the information retrieved from the DP, the wrapper routes the invocation to the correct successor deployment without requiring any code changes. We selected pub/sub as a geospatial offloading glue due to its availability at all major cloud service providers (e.g., AWS SNS, Azure Service Bus, and Google Pub/Sub) and its ability to support many programming languages, allowing for future portability. Additionally, pub/sub services seamlessly integrate with serverless functions as invocation triggers, ensuring dependable message delivery as function execution triggers. Furthermore, the services provide a level of reliability by requiring subscriber acknowledgment. If no acknowledgment is received, the pub/sub service automatically retries to deliver the messages.

Chapter 5

Framework Implementation

Following the design outline in the previous Chapter 4, we will now spend more time on specific implementation details, highlighting specific challenges we solved.

5.1 Overview

We implemented CARIBOU, both the developer API package as well as the actual infrastructure components, in the Python programming language¹. A future maintainer can switch out any component if the interface functionalities are maintained. We decided to actively practice the separation of concerns pattern, which allows faster development, easier component-wise testing, and interaction over clear interfaces. We also applied the template pattern, where applicable, to simplify the extension for additional providers. For an initial version of the framework, Python, one of the most popular serverless languages [63, 112], provided the most robust third-party support and was a low-overhead language regarding the onboarding of the team members. The code base of CARIBOU contains 16K source lines of code developed over approximately four months. The framework will be published as an open-source Python package called `caribou`, containing the source code for the API package usable by workflow developers and the code for the infrastructure components. It is important to note that we will only use the Python package name when referring to the API package. We implemented the framework following standard Python testing and linting practices. The framework code adheres to the practices set by the formatters `black` [180], `isort` [59], `pylint` [135], and `mypy` [158]. The source code contains unit and integration tests, boasting a test coverage of > 90%. Table D.1 highlights the core component files and indicates the corresponding component they implement.

¹The framework source code can be found at <https://github.com/ubc-cirrus-lab/caribou>

5.1.1 Framework Architecture

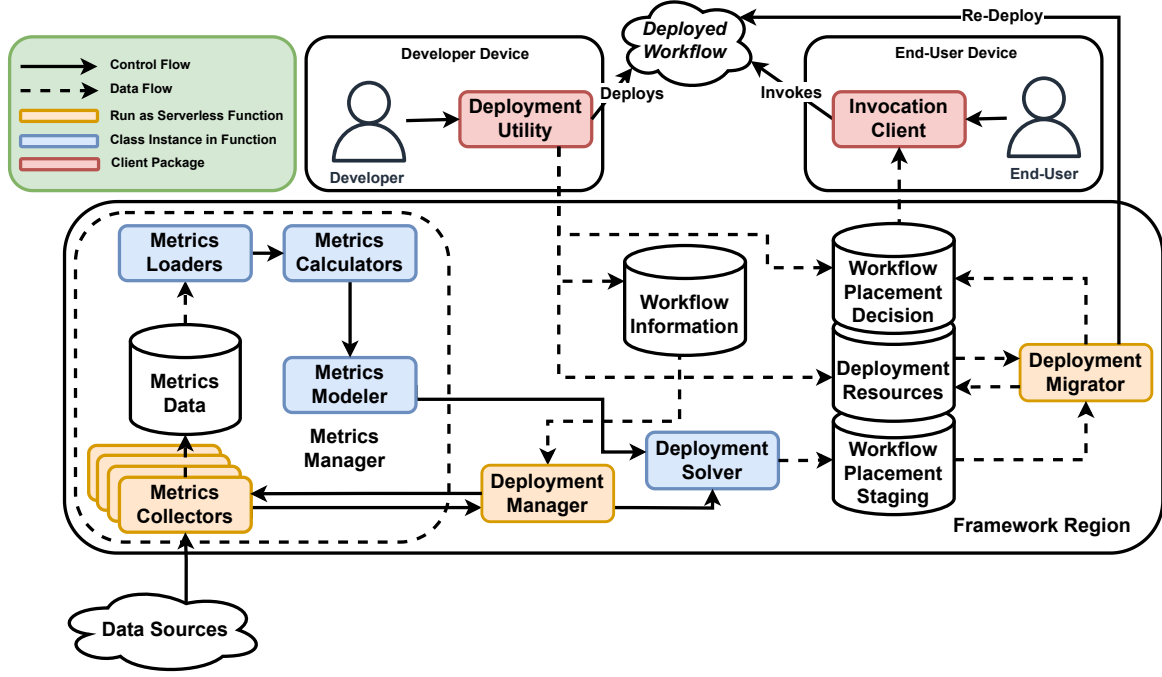


Figure 5.1: The framework is deployed in one region, whereas the developer and end-user might be in different regions. When required, the Metrics Collectors, Deployment Manager, and Deployment Migrator are run as serverless functions. The communication between components happens asynchronously using tables in distributed key-value stores, indicated as databases.

The framework architecture outlined in Figure 5.1 is an extension from the overview presented in Section 4.6.1 with more implementation details added. The three components run as timed serverless functions can be deployed independently and triggered based on timers. These three functions optimally require at least the permissions outlined in Listing C.1. Currently, the framework and the workflow developers must be in the same AWS organization or at least have permission to access the same resources. A future iteration would improve this permission model to disaggregate the two, potentially with a per-developer account to calculate the usage on a per-account basis.

Framework Component	Triggered by	Triggered when
Deployment Utility	Developer	Deploying workflow
Invocation Client	End-User	Invoking workflow
Deployment Manager	Timer	Every 24h
Deployment Solver	Deployment Manager	Solve required
Deployment Migrator	Deployment Solver	Migration required
Metrics Manager Carbon	Timer	Every 24h
Metrics Manager Logs	Deployment Manager	Token check required
Metrics Manager Pricing	Timer	Once every two weeks
Metrics Manager Performance	Timer	Every 24h

Table 5.1: Component trigger timings.

5.2 Component Triggers

We implemented all components independently to be invoked as serverless functions when triggered. Each component can run asynchronously and independently, communicating over tables in a distributed key-value store (§5.4.3). Each component’s invocation times (when) and triggers (by what) vary depending on its function. Table 5.1 outlines each component’s trigger.

5.3 Deployment Plan

The deployment is the primary data required by many components to interact with each other and is especially important for communicating the workflow deployment according to the framework’s optimizations. Like all other data used to communicate information within the framework, a deployment plan (DP) is a JSON file representing all the relevant data for deployment migration and routing invocations to the correct regions. The workflow deployment plans are stored in the `workflow_placement_decision_table`. An example can be seen in Listing B.1. The DP is structured with the following top-level keys:

- **instances:** Representing DAG information such as nodes and edges.
- **current_instance_name:** Containing the information on the current node in the DAG when reading the JSON used for invocation routing.
- **workflow_placement:** Containing the home region deployment and any framework-generated placements.

We decided to additionally store subcomponents of this information that do not go out of sync, such as the DAG information, in additional tables at the distributed key-value store. This replication in subtables is a measure to reduce the costs of table accesses when only subparts of the information are required. Since the DAG information does not change, no data synchronization is needed.

5.3.1 DAG Information

The DP contains information on the DAG, where the following information is maintained for all nodes:

- Node name
- Node Region constraints
- Successor nodes
- Predecessor nodes
- Dependent Sync Successors

The dependent sync successors contain information on the incoming edges of any synchronization node where a path from this node to the synchronization exists. The workflow wrapper (§5.5.6) requires this path information for the conditional invocation check for a synchronization node (§4.3.7). This information is maintained statically to reduce DAG operations, such as retrieving paths on the hot path of a workflow execution.

5.3.2 Current Node Information

Additionally, the DP contains information on the current node. This information is trivially the start node in the version stored in the distributed key-value store but will be adapted when the DP is passed from node to node at invocation (§4.8.3).

5.3.3 Deployment Information

Lastly, the DP contains the workflow deployment information, including the home region deployment and potentially framework-solved deployments. A deployment includes all the information necessary for traffic routing to the correct message publication channel (messaging queue) to which the deployed source code function is subscribed, which is the messaging queue identifier and region of all nodes of the DAG. The framework-generated deployments also have a retirement date determined by the *Deployment Manager* (§4.6.4). A framework-solved deployment can be granular from a whole day to hourly.

5.4 Interactions

The components and users of the framework interact asynchronously. Thus, listing these interactions and elaborating the details in the following listing is beneficial. There are three types of interactions in the CARIBOU framework:

- **Developer interactions** (§5.4.1): Developers require a robust interface to develop complex serverless workflows with the introduced concepts of conditional invocations and synchronization points, as well as

being able to communicate constraints, tolerances, and additional dependencies. Additionally, developers need to be able to manage their respective workflows over a CLI that retrieves and updates values in the corresponding distributed key-value tables.

- **End-user interactions** (§5.4.2): End-users require endpoints to invoke the geospatially deployed serverless workflows. These endpoints must alleviate the overhead of manually fetching the current deployment, expecting the result of an invocation to be the same no matter where the workflow was executed.
- **Framework component interactions** (§5.4.3): The framework needs to communicate asynchronously to communicate the deployment decisions as well as to update the source code deployments accordingly.

5.4.1 Developer Interface

Workflow developers who want to benefit from our framework’s adaptive geospatial deployment functionalities must be able to communicate the respective workflow structure to our framework and manage deployed workflows. The following subsection introduces how a workflow developer can annotate their source code functions to represent a deployable workflow using our framework (§5.4.1). Following this, we introduce the provided CLI functionalities enabling workflow developers to manage the deployed workflows (§5.4.1).

Workflow Declaration

A workflow developer must declare a workflow in their respective source code to benefit from our framework’s capabilities for self-adaptive geospatial workflow deployment. Source code annotations enable our framework to infer the workflow’s DAG structure and enable invocation routing. Meta-information is required to guide deployment generation concerning tolerances, objectives, and constraints. The developer declares a workflow by defining two parts for each workflow:

1. Declare the workflow in the source code through the API provided by the `caribou` Python package.
2. Declare the workflow deployment manifest, where the developer can provide further deployment details such as tolerances, objectives, and additional configurations.

We outline the expected project setup for a workflow in Listing 5.1.

Source Code API: To declare a workflow, a developer uses the lightweight API functions provided by the `caribou` Python package. This declaration


```

1 workflow/
2   .caribou/
3     config.yml
4     iam_policy.json
5   src/
6   app.py
7   requirements.txt

```

Listing 5.1: Directory structure of a workflow.

has to happen in the `app.py` file. We show an example of a workflow declaration in source code in Listing 5.2. Additionally, developers can add any source code files to the `src` directory that implement additional application logic, which can then be imported into the `app.py` file and used there. The workflow instance declared (line 2) provides three methods to define the DAG and represents the workflow wrapper (§5.5.6).

```

1 from caribou import Workflow
2
3 workflow = Workflow(name="example", version="0.0.1")

```

Listing 5.2: Workflow declaration.

To declare a source code function part of this workflow, the developer registers the function handler using a Python decorator as outlined in Listing 5.3. This decorator will wrap the source code function (§5.5.6). Inside the decorator declaration, the developer can indicate whether the function is a start node (line 3) of the workflow. The framework enforces one start node per workflow when deploying the workflow. The developer can specify function-level configurations, such as region constraints (allow/disallow), to enforce function-level data compliance (lines 5-18). If any regions are allowed then only these regions will be viable candidates for deployment. If the workflow developer disallows any regions, all regions except these are viable candidates. A region can never be allowed and disallowed, meaning allowing regions is more restrictive than disallowing regions. In the decorator, the developer can also configure the deployed serverless function on a provider-specific level concerning specific parameters. We add the provider specificity for future extensibility to additional providers apart from AWS. In AWS, memory and timeout are possible options (lines 19-26). The function parameter (`event`, line 29) holds the value passed in the invocation where the value is either the workflow invocation parameter or passed from a previous source code function invoking this function. The workflow developer is usually left to verify the parameter data structure, a practice common in dynamically typed languages such as Python.

Using two API calls, the developer can implicitly create the DAG struc-

```

1 @workflow.serverless_function(
2     name = "Example-Function",
3     entry_point=True,
4     regions_and_providers = {
5         "allowed_regions":
6             [
7                 {
8                     "provider": "aws",
9                     "region": "region\_1"
10                }
11            ],
12     },
13     "disallowed_regions": [
14         {
15             "provider": "aws",
16             "region": "region\_2",
17         }
18     ],
19     "providers": {
20         "aws": {
21             "config": {
22                 "timeout": 120,
23                 "memory": 512,
24             },
25         },
26     },
27     environment_variables=[{"key": "example_key", "value": "
example_value"}],
28 )
29 def example_function(event):
30     results = workflow.get_predecessor_data()
31     workflow.invoke_serverless_function(
32         next_function, intermediate_data, conditional
33     )

```

Listing 5.3: Source code function declaration.

ture inside a source code function by indicating edges between source code functions, flattened to a DAG at the static code analysis happening in the *Deployment Utility*. A call to `invoke_serverless_function` (lines 31-33) corresponds to an outgoing DAG edge. When calling, the handler of the other function and intermediate data are passed as arguments to the function. The developer can optionally pass a boolean variable to indicate a conditional invocation. When the source code function is executed as part of an invocation, the workflow wrapper dynamically evaluates this boolean at the invocation of the `invoke_serverless_function` function.

The developer can indicate a synchronization node in the source code by calling the `get_predecessor_data` method (line 30). Since the developer can not indicate what predecessor a synchronization is supposed to wait on,

any source code function annotated as a synchronization node will act as a barrier, waiting on all predecessors. Logically, it also requires that the mapping between the node and source code functions be one-to-one. This API call will, at execution, retrieve the predecessor intermediate data from the distributed table and return them as a list of values, one value per predecessor.

Deployment Manifest: The developer must additionally provide a deployment manifest that consists of the `config.yml`, the `iam_policy.json`, and an optional `requirements.txt` file. In the requirements file, the developer can indicate Python third-party library dependencies. In the identity access management (IAM) file, the developer can give workflow-specific permissions policies concerning services at the cloud service provider [25]. These policies are associated with every deployed source code function linked to the deployed serverless workflow.

The configuration file allows developers to define workflow-level objectives, tolerances, and more. We outline an example configuration in Listing A.1 and a minimal example of the IAM file in Listing A.2. The developer also specifies the "home region"—the initial deployment region of the workflow. Additionally, the developer can define the tolerances on end-to-end latency, carbon emission, and cost per invocation in this file. The *Deployment Solver* enforces these at deployment generation.

Lastly, developers can specify regions or providers eligible or prohibited for deployment to enforce regulatory compliance on a workflow level, where function-level configurations supersede workflow-level ones. The framework defaults to considering all potential regions if no regions are explicitly allowed or disallowed.

Framework Command Line Interface

After declaring a workflow, the developer can deploy said workflow. For all the following actions, the developer requires a user with a specific set of permissions in AWS to deploy and manage the workflows. We outlined these permissions in Listing C.1. The workflow developer can initiate a deployment using the framework's command-line interface (CLI). The CLI offers the following commands to interact with the framework:

- **deploy:** When either the current work directory in the command line corresponds to the workflow directory or the path of the workflow directory is provided using the `--project-dir` flag, the command deploys a workflow to the framework. Running the `deploy` command will kick off the process outlined in §5.5.4.
- **list:** List the currently deployed workflows in the framework.
- **new_workflow WORKFLOW_NAME:** Initiate a new workflow from a tem-

plate with the passed workflow name at the current working directory. This CLI functionality is provided to simplify the correct workflow directory structure setup and will generate a default version of the structure outlined in Listing 5.1.

- **remove WORKFLOW_ID**: Removes a workflow with all its deployments from the framework and all deployed regions. This action is destructive, and the workflow can no longer be invoked. The framework will remove the workflow from all regions where any source code function is currently deployed, and any workflow trace will be removed from all communication tables.
- **run WORKFLOW_ID**: Invoke a specific workflow using the *Invocation Client* (§5.5.6). Additionally, the `--argument` flag can be used to pass a parameter to the invocation.

5.4.2 End-user invocation

After we outlined how workflow developers interact with our framework, we want to introduce how end-users interact with and use the workflows managed by the framework. End-user workflow invocations can be routed directly to the source code function representing the start node at the home region or through a proxy provided by the developer that calls the `caribou`-provided CLI utility `run` command outlined. We envision adding support for some REST endpoints that provide load-balancing functionality.

5.4.3 Framework Component Interaction

Next, we will examine the framework’s internals and outline how its components interact to manage the workflows. The framework components interact using distributed key-value stores. We chose DynamoDB because its table functionalities are relatively simple to set up and relatively cheap, and similar services are available at all major cloud service providers. We decided to use the AWS-specific offering dynamoDB since it integrates easily with the rest of the framework. Most importantly, no framework component interaction requires real-time reactions from other components. Instead, all interactions happen asynchronously, and specific components, such as the *Metrics Manager*, might only update the underlying metrics data very infrequently. Handling every request for data synchronously by starting up the corresponding components or, even worse, by having the framework up and ready to accept calls all the time would go against the framework’s overhead reduction design property. Thus, running the components only when needed and storing the data required to communicate between the components in a distributed key-value store, such as DynamoDB in AWS, is sufficient.

We abstracted the interactions of the components with the tables using the `RemoteClient` class. This class is used by all components, providing

the necessary API abstractions where required. For AWS, the remote client class is a wrapper for the `boto3` package [22], itself a wrapper of AWS API calls. This double abstraction is necessary to make the framework flexible to add additional providers and reduces the maintenance effort of `boto3` within the code if `boto3` changes any interface function. To interact with the distributed key-value stores, the `RemoteClient` class provides create, read, update, and delete (CRUD) functions. The communication tables all have the same layout, with a `key` and a `value` where the key must be unique, while the value is usually a string representation of the JSON data.

Table D.2 outlines all tables required for system interactions, the respective keys and values, and which component uses them to communicate. This simple interface underutilizes the strengths of querying in, for example, DynamoDB. However, this simple interface is strong enough since most interactions involve iterating over keys or requiring specific information without sub-information. Additionally, by not tying ourselves too closely to dynamoDB-specific features, we enable an easy extension to a more suitable storage solution in the future.

5.5 Framework Components

In the following subsections, we will introduce all the components that make up the framework behind the scenes, focusing on relevant implementation details, highlighting interfaces with other components, and explaining the implementation challenges. The components are:

- Workflow Configuration (§5.5.1)
- Deployment Manager (§5.5.2)
- Deployment Solver (§5.5.3)
- Deployment Utility (§5.5.4)
- Deployment Migrator (§5.5.5)
- Invocation Client (§5.5.6)
- Workflow & function wrapper (§5.5.6)
- Metrics Manager (§5.5.7)

5.5.1 Workflow Configuration

One general component that is used by multiple components is the `WorkflowConfiguration` class. We wrap the developer-defined configuration YAML in this class for access and communication about the configuration. This class also contains the workflow DAG structure obtained from the deployment plan or another table. The workflow DAG structure is, for example, required by the *Deployment Manager* that needs to ascertain the complexity of the DAG for the token inflow.

5.5.2 Deployment Manager

The **Deployment Manager (DM)**, as first introduced in Section 4.6, is responsible for the adaptive and overhead aware triggering deployment solves. The DM is the only component aware of all workflows in one place and manages workflow deployments by tracking the state of each workflow in the framework. It additionally maintains the state of each workflow with a token bucket, where the carbon reduction potential determines the inflow of tokens, and the cost of a deployment solution determines the outflow. The bucket is limited to the invocation data of the last 30 days, reflecting the data collection period. The *Deployment Utility* informs the DM of a new workflow by adding a corresponding key-value pair to the `deployment_resources_table` (§5.5.4).

The DM then iterates over all workflows in this table and checks the process outline below. The DM keeps a state of each workflow in the `deployment_manager_workflow_info_table`, where it stores per workflow the time of the last deployment solve, the token overflow from the previous token check, and the time of the next token check. Figure 5.2 illustrates the self-adaptive process orchestrated by the DM. When iterating over the workflows, the DM goes through the following process:

1. If the workflow is new, meaning the DM has never checked it, it initiates a metrics collection from the *Metrics Manager*.
2. If the workflow is not new, the DM checks whether the token check has passed; if yes, initiates a metrics collection from the *Metrics Manager*; if no, continues with the following workflow at step 1.
3. After having collected the workflow-related metrics, the manager checks how many invocations this workflow received in the past period since the last check. A workflow must have been invoked at least ten times in the past period. If this is not the case, the manager continues with the following workflow at step 1.
4. If the minimal invocation count is present, the manager calculates the positive inflow tokens based on invocations, workflow end-to-end runtime, and standard deviation of carbon intensities of all regions, representing a crude measure of potential offloading improvement.
5. The manager invalidates old tokens from the bucket (older than 30 days).
6. It then dynamically determines which deployment run is affordable given the inflow plus leftover tokens from past periods. Eight deployment runs are possible, each with a different hourly granularity of the corresponding deployment (one plan for the whole day up to a plan for every hour). Increased hourly granularity increases token costs since the overhead is proportionally higher.
7. If any deployment run is affordable, the DM kicks off a solve deploy-

ment from the *Deployment Solver*, passing the deployment retirement date and time calculated in step 8; if not, it directly continues to determine the next token check time.

8. The next token check time is determined based on the delta between the token generation rate and current bucket content, smoothed by a sigmoid function, to ensure that the next check is appropriate based on the invocation rate of the past period. This smoothing and adjustment means that if the rate has changed from high to low, the next check will be later; conversely, if the rate increases, the token check will be later. Generally, the system attempts to stabilize new deployments; if the rate has not changed significantly, the next check will be later than the previous. If a deployment run has happened, the retirement date and time of the deployment are the same as the next token check time. The minimal next token check is 24 hours since no new carbon data will have been collected by the *Metrics Manager* below this threshold. The current maximum is seven days as the quality of carbon predictions deteriorates afterward.

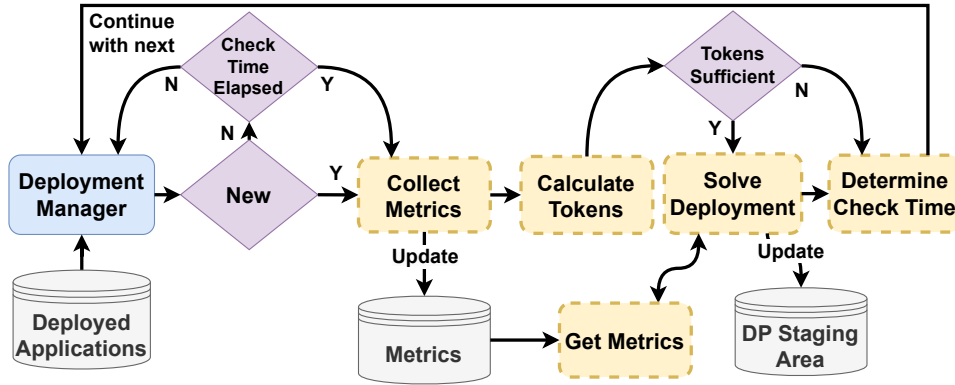


Figure 5.2: CARIBOU dynamically determines the deployment generation trigger frequency with a token bucket algorithm.

5.5.3 Deployment Solver

The **Deployment Solver (DS)**, initiated by the *Deployment Manager* (§5.5.2), will, given a DAG and information about tolerances and constraints, generate the requested deployments. The component uses an interface to the *Metrics Manager* to retrieve the required end-to-end estimates (§4.4.1) for any in the process generated deployment. The interface consists of an instance of the `DeploymentMetricsCalculator` class, which accesses the tables filled and maintained by the *Metrics Manager* containing the information required to calculate the metrics. More on this will follow in the

corresponding Subsection 5.5.7. The calculator class then runs the Monte Carlo simulation based on the DS-generated deployment.

We implemented the DS using a Heuristic-biased Stochastic Sampling (HBSS) algorithm, which has proven to be a good compromise between brute force and a coarse-grained solver. We will explain this further in the following subsection. Based on the generated deployments, the component will then select the best deployment based on the developer-indicated priority order (§4.5.2).

Heuristic-biased Stochastic Sampling

The DS initializes the HBSS algorithm with a simple approach: the entire workflow is deployed to all allowed regions, where the regions need to be permitted for all workflow nodes. If the DS is run without location constraints, all available regions are retrieved from the `available_regions_table`. The constraints can never be so strict that no region is possible for all nodes. It must be possible to deploy the whole workflow to at least the home region, the developer’s defined default deployment region. Then, the heuristic search begins. In every iteration, the algorithm generates new deployments, retrieves the corresponding metrics values of this deployment from the `DeploymentMetricsCalculator` instance, tests them for any tolerance violation, and checks for improvement. The search terminates after exploring the search space or after β iterations, where β depends on DAG size and region count. The parameters α, γ, δ represent the learning rate, temperature, and bias probability. All parameters such as $\alpha, \beta, \gamma, \delta$ are open for further optimization. Based on a small-scale parameter search, we decided on the values outlined in Algorithm 1.

Staging a Deployment

When the DS lexicographically selects the best deployment according to the developer-defined objectives, the DS stages this deployment in the `workflow_placement_solver_staging_area_table` for migration by the *Deployment Migrator*.

5.5.4 Deployment Utility

We implemented the **Deployment Utility (DU)** to reduce the manual tasks required to deploy a complex serverless workflow. The workflow developer invokes the DU when the developer runs the `deploy` CLI command. The utility executes on the workflow developers’ local machine. When given the path to a source code directory, the DU statically analyzes the application structure, identifies dependencies, packages up the function as a Docker image, and creates the function and all necessary dependencies. The DU is designed to automate the creation and deployment of a serverless workflow

Algorithm 1 HBSS algorithm used for finding optimal deployments.

```

1: function HBSS(Deployments)           ▷ Deployments will always be of size > 0
2:    $\alpha \leftarrow \lceil 0.2 \times |N| \rceil$            ▷ At least 1
3:    $\beta \leftarrow |N| \times |R| \times 6$            ▷ Learning rate
4:    $\gamma \leftarrow 1.0$            ▷ Temperature
5:    $\delta \leftarrow 0.2$            ▷ Bias probability
6:    $i \leftarrow 0$ 
7:   CurrentDeployment  $\leftarrow$  Deployments[0] ▷ Initialise with home deployment
8:   while  $i < \beta$  do
9:     NewDeployment  $\leftarrow$  GENNEWDEPLWBIAS(CurrentDeployment)
10:    if TOLERANCEVIOLATED(NewDeployment) then
11:      continue
12:    end if
13:    if NewDeployment.metric < CurrentDeployment.metric or
      STOCHASTICMUTATION( $\gamma$ , CurrentDeployment, NewDeployment) then
14:      CurrentDeployment  $\leftarrow$  NewDeployment
15:       $\gamma \leftarrow \gamma \times 0.99$            ▷ Cool down
16:      Deployments.add(NewDeployment)
17:      if len(Deployments) ==  $|R|^{|N|}$  then           ▷ No more deployments
18:        break
19:      end if
20:       $i++$ 
21:    end if
22:  end while
23: end function
24: function GENNEWDEPLWBIAS(CurrentDeployment)
25:   NewDeployment  $\leftarrow$  CurrentDeployment
26:   InstancesToMove  $\leftarrow$  RANDOMSELECTINSTANCES( $\alpha$ )
27:   for instance  $\in$  InstancesToMove do
28:     ARegions  $\leftarrow$  GETALLOWEDREGIONS(instance)
29:     if Random <  $\beta \wedge \exists \text{BiasRegion}$  then
30:       NewDeployment[instance]  $\leftarrow$  SELECTBIASEDREGION(ARegions)
31:     else
32:       NewDeployment[instance]  $\leftarrow$  RANDOMCHOICE(ARegions)
33:     end if
34:   end for
35: end function
36: function STOCHASTICMUTATION( $\gamma$ , CurrentDeployment, NewDeployment)
37:    $\Delta \leftarrow \forall |CD.metric - ND.metric|$ 
38:   return Random <  $e^{-\frac{\Delta}{\gamma}}$ 
39: end function

```

with multiple functions. It eliminates the need for manual bootstrapping of dependencies and steps, allowing developers to deploy their workflow quickly and effortlessly. The actual deployment process happens inside the **Deployer** class used by both the DU and *Deployment Migrator* since the processes for both are very similar. At the end of a deployment, the DU will upload the

metadata to both the `deployment_resources_table`, which maintains all deployed function information, as well as the home region, deployed image locations, maintained in `caribou_workflow_images_table` required by the *Deployment Migrator*. Lastly, the DU stores the home region deployment and all the other information in the `workflow_placement_decision_table` under the workflow ID.

Static Code Analysis

The DU statically analyzes the source code inside the passed directory, starting from the `app.py` file, while including any additional functions imported from the `src` directory in the analysis. Starting by adding the start node of the workflow to a queue, the static analysis algorithm removes elements from the front of the queue until the queue is empty. All outgoing edges are added to the DAG for each node removed from the queue, and all edge destinations and other source code functions are added as nodes to the queue as long as there does not already exist a node with the same name in the queue.

When adding a node to the DAG, the node will be assigned a unique name that serves two purposes:

1. Identify the node uniquely in the DAG.
2. Enable the identification of a successor node based on the current node for the inductive process as outlined in the invocation design section (§4.8.3).

The initial node will be named the following, where `index_in_dag` is trivially 0:

```
<function_name>:entry_point:<index_in_dag>
```

All synchronization nodes are unique, and since the interface does not specify which predecessor the sync node waits on, it must wait on all predecessor nodes, requiring it to be unique to prevent cycles. The naming of synchronization nodes is as follows:

```
<function_name>:sync:
```

Lastly, all other nodes are named as outlined, where `successor_of_predecessor_index` is defined by the number of outgoing edges of the predecessor:

```
<function_name>:<predecessor_function_name>_<
predecessor_index_in_dag>_<successor_of_predecessor_index
>:<index_in_dag>
```

At the end of the static code analysis, the DU will have a JSON representation of the DAG, where each node's predecessors, successors, and any dependent sync successors are stored. The workflow wrapper (§5.5.6) requires the dependent sync successors for the conditional edge path check,

where a node needs to update the annotation for all dependent synchronization nodes in the case of a conditional non-invocation. At this point, the DU also performs a cycle and compliance check to prevent the deployment of impossible graphs according to our current workflow model. If the workflow ID defined by the developer has already been deployed, the DU aborts any further deployment.

Function Package

Any source code function part of a workflow, as deployed by the DU, needs to be packaged up into a function package. If required, the `DeploymentPacker` class packages the source code function instantiated in the `Deployer` class. In our implementation, a function package is a Docker image containing all Python dependencies, the source code, the required framework source code for both the function and workflow wrapper to route invocations correctly, and any runtime environment dependencies, such as additional software. We decided on function packages to ensure that the execution environment is the same and independent of the region and underlying software architecture. It additionally allows the developer to define further framework-level dependencies such as `ffmpeg` [179] for audio and video recording, conversion, and streaming not provided in all runtime environments by default. It also enables future maintainers to effortlessly extend to different programming languages since it decouples this part of the framework implementation from a specific language. Thus, the function package could still be deployed and migrated with the same framework code, even if the source code function code is different. Lastly, Docker images have the added benefit of being efficiently and reliably migratable cross-regionally.

Function Creation and Dependencies

A function deployed with our framework will consist of a function package, a handler to the corresponding source code function required by the serverless function service, a role associated with the permissions of this particular function, and the messaging queue representing the publication vessel as well as the corresponding subscription from the function to the queue. The DU facilitates all these actions using the `RemoteClient` offering these functions. The DU must set up all these components at the cloud service provider to enable a seamless function execution. Furthermore, each region where this function is deployed will have its registered function, role, messaging queue, and a subscription from the function to the messaging queue. It is important to note that all the components mentioned, except the role, are region-specific concepts. That means they can not be migrated and must be registered anew when a source code function is migrated to a new region. At this point, the DU also uploads the workflow corresponding information to

the `deployment_resources_table` and `caribou_workflow_images_table`.

5.5.5 Deployment Migrator

When the framework has chosen a new region for a specific application function, it must be able to migrate the function package and all the necessary dependencies to this new region. The **Deployment Migrator (DMi)** facilitates this migration by comparing the deployed workflows, maintained in the `deployment_resources_table` with the newly required staged deployments according to the `workflow_placement_solver_staging_area_table`. The former table is also updated for each migrated function, ensuring an accurate state of deployed regions per function persists. The deployment migrator utilizes the same **Deployer** function to facilitate this migration since many of the required steps are the same, with some simplifications as not requiring rebuilding the Docker image. Thanks to how serverless functions are packaged, this process is relatively straightforward and does not require any change to the deployed code. This packaging also means no framework component adapts the source code after being packaged on the developer's local device. When migrating a function from one region to another, the framework needs to relocate the corresponding Docker image to the new region and register the function and additional components in that new region. Building a Docker image anew every time we migrate to a new region would be counterintuitive and computationally expensive. It would be against our design property of low overhead. Thus, we decided to migrate the images by copying them from the region where we know this application will always be deployed (home region) to any new region. Copying a Docker image is not a functionality available from the cloud service providers themselves. Functionalities like multi-region replication exist, for example, in AWS. However, they can not be retroactively applied. To further reduce the DMi's footprint, we decided to look for a solution that does not require a Docker daemon for migration. Since we designed the DMi to run as a serverless function, this would be impossible in AWS.

Given these constraints, the best solution is an open-source package maintained by Google, and part of the Go language container registry called **crane** [151]. It introduces a dependency to the Go programming language required to run it; however, this dependency is much lighter and natively supported on more machines than a full Docker daemon. The package **crane** offers a simple command line interface for interacting with remote container images and can also be used to migrate images from one container registry to another. The DMi retrieves the name and location of the image from the `caribou_workflow_images_table`. We use this functionality to migrate the source code image to the new region. Once the DMi migrated the deployment, the DMi removes the newly deployed deployment from the staging area table and updates the corresponding framework deployment of the DP

in the `workflow_placement_decision_table`.

5.5.6 Execution

When executing a workflow, the framework needs to correctly route the call to either the start node of the current deployment if there is a framework deployment or to the home region. We opted for an *Invocation Client* to facilitate this initial routing, which can be invoked using the CLI `run` command (§5.4.1). The client will then call the start node in the region, as defined by either the home region or framework deployment. Each workflow invocation will receive a unique run ID at the client to prevent conflicts concerning synchronization nodes and uniquely identify an invocation for debugging. Each node's source code function is wrapped once as part of the workflow to provide workflow-specific information and, secondly, as a function providing the logic required for cross-regional traffic routing (§4.8.3).

Invocation Client

The **Invocation Client (IC)** retrieves the current deployment plan (DP) from the distributed key-value store table `workflow_placement_decision_table`, determines if there is a framework deployment, and, if yes, retrieves the deployment for the specific hour of invocation, depending on the framework deployment granularity, the home region deployment naturally only has one granularity. It then invokes the start node at the correct region with the messaging topic by posting a message to the proper topic. In any case, the client routes 10% of the workflow invocations to the home region for performance benchmarking and metric collection.

Workflow Wrapper

All source code functions of a workflow are wrapped in the `CaribouWorkflow` class, which provides methods needed by other components, such as the *Deployment Utility* to interact with workflows and provides the API to the workflow developer to declare a workflow. When a source code function is deployed, this wrapper will be a specific instance for this function, providing helper functions to the function executions for routing. In the context of serverless functions, where the same execution environment is reused for multiple successive invocations, this wrapper instance is also reused by AWS for multiple node invocations. When accessing data that is only valid for one run by the run ID, we must differentiate between runs. Following that, we will provide more information about what happens behind the scenes of these API functions. The three important API functions provided by the workflow are:

- **register_function**: The workflow developer can use this function to register a source code function used for static code analysis at the

Deployment Utility. The workflow wrapper stores each function as an instance of the `CaribouFunction` class that captures the function-specific data such as callable, name, region constraints, and other data as specified in subsection 5.4.1. The list of source code functions registered is used by the *Deployment Client* for the static code analysis.

- **invoke_serverless_function:** The workflow developer can use this function to indicate an outgoing edge from one source code function to another where the target function to be called is passed together with an optional conditional variable. Inside this function, the successor node name is determined and used to determine the successor's region and messaging queue according to the deployment used. The workflow wrapper copies the DP and copies the current node name in the plan to the successor node's name. The wrapper piggybacks the copied DP on the intermediate data. Additionally, any logic required when conditionally not invoking a successor is executed. The DP includes informing dependent successor synchronization nodes, which can be retrieved from the DAG structure, of non-execution and potentially invoking dependent synchronization nodes if required. If the successor is a synchronization node, the workflow wrapper uploads the intermediate data to `sync_messages_table` and the annotation to the `sync_predecessor_counter_table` in the region of the synchronization node. Otherwise, the intermediate data will be directly forwarded to the successor.
- **get_predecessor_data:** The workflow developer can use this function to register a synchronization node that awaits the call from all predecessor nodes. This function retrieves the intermediate data from the `sync_messages_table`, where all predecessors stored their respective intermediate data.

Node location determination: As previously outlined, determining the node location when executing a source code function deployed in a region is non-trivial. We had to resort to retrieving the successor by keeping track of the outgoing edge index and knowing the current node and successor's names. Given our naming scheme for nodes, this information is sufficient to uniquely determine the name of any successor node, which then gets forwarded to this node. We call this procedure inductive because, by knowing the start node location and its outgoing edges, we can inductively determine the node locations of the whole DAG.

Data transmissison: Since we did not want to restrict the user's ability to send data but needed to wrap the intermediate data with the DP, we had to implement a custom JSON encoder and decoder to handle any data handed as payload.

Synchronization nodes: The annotation table, `sync_predecessor_counter_table`, records which predecessors have called and, if called, have conditionally invoked. Recording this information is done by storing a map for each synchronization node where the predecessors store their name as a key and the conditional as a value. When updating, the last updated value is directly returned. Since AWS DynamoDB offers strong consistency for writing, this ensures that the value returned will conform to the transaction ordering. This table's key is a concatenation of the workflow ID, the synchronization node name, and the run ID of this invocation. On the other hand, the synchronization intermediate data table (`sync_messages_table`) stores the data for each synchronization node with the same key. Each predecessor also updates this data with the intermediate data before updating the annotation, ensuring transaction ordering and then read by the synchronization node using a `ConsistentRead` parameter to ensure that the complete data from all predecessors is read [26].

Function Wrapper

The source code function is wrapped in a decorator that runs before and after the function invocation. This wrapper is hidden away through the decorator that the workflow developer uses to register a source code function (§5.4.1). Before the function, the wrapper retrieves and stores the current DP from the payload retrieved from the predecessor to be used at a successor invocation of this node. Additionally, it retrieves the current node location required to determine any successor location. Lastly, this wrapper logs information the *Metrics Manager* requires to collect all the necessary information on a workflow.

5.5.7 Metrics Manager

The **Metrics Manager (MM)** collects, transforms, and aggregates the data required to calculate the metrics. In Figure 5.3, we outline the architecture of the MM in more detail. The MM consists of two subcomponents:

1. **End-to-End Modeling:** The modeling is used to enable the *Deployment Solver* to access the necessary end-to-end modeled metrics to make decisions about deployments.
2. **Data Collection:** Data must be collected from the required sources, transformed, and then correctly aggregated in the tables required for the end-to-end modeling.

End-to-End Modeling

The End-to-End modeling enables the *Deployment Solver* to make guided decisions about potential deployments by returning the framework-supported

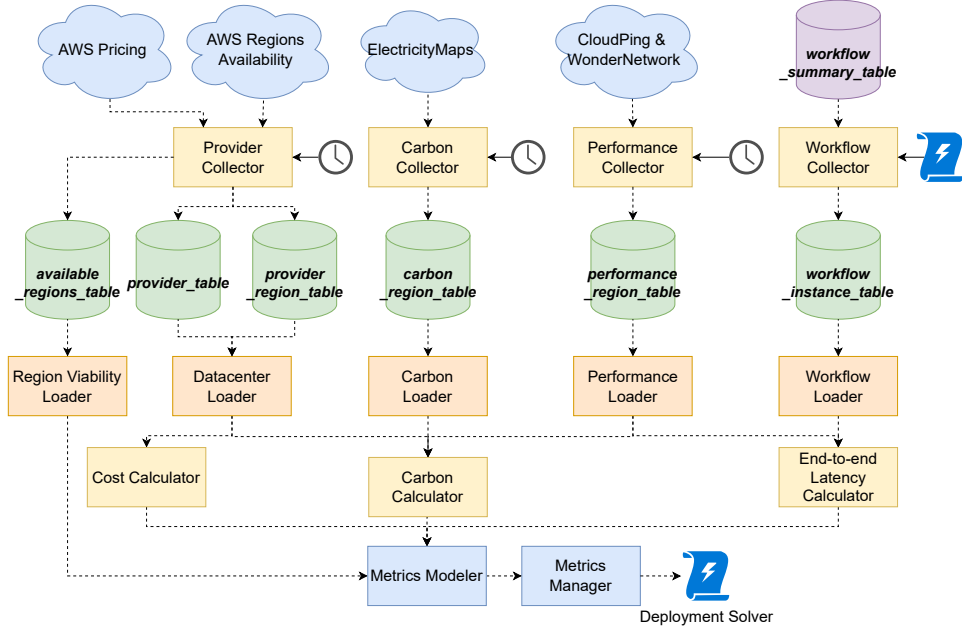


Figure 5.3: CARIBOU MM component overview. Lines indicate data flow. Yellow boxes indicate classes that retrieve and aggregate data, and orange boxes indicate classes that load the data. Green tables are aggregated data, and violet tables are intermediate data tables.

metrics when needed. The *Deployment Solver* passes a DAG, a deployment, and a time of day to retrieve the corresponding metrics. The MM facilitates the end-to-end modeling of all metrics in the **MetricsModeler** class using a Monte Carlo simulation, as previously outlined (§4.4.1), which involves sampling execution runtimes, data transfer sizes and latencies, and conditional edges based on past workflow invocations. This sampling is done by the MM using the corresponding access methods of the **WorkflowLoader**, giving access to the past invocation samples in the required granularity (per node-region deployment).

When a transmission latency between two regions is missing but required for sampling latencies, we extrapolate the latency data between functions based on existing measurements in the home region and then calculate the delta between measured home region transmission latency and externally retrieved data transmission latency. We then use this delta to extrapolate the transmission time to other regions by multiplying it with the external source value. For missing execution runtimes and transfer data sizes, the home region data is used since we assume that the distribution remains more or less homogenous for these data points.

When the MM through the **MetricsModeler** simulates a deployment at a specific hour of the day, it calculates each metric’s corresponding node

and edge values. The abstract class a metric needs to implement to provide the calculation methods for both edges and values is the `MetricCalculator` class. Metric-specific instances are then used in the `MetricsModeler` to run the Monte Carlo simulation. Each calculator class has a corresponding loader class instance, which accesses the data tables provided by the data collection components.

The required data by each calculator is:

- **End-to-end Latency Calculator:** Requires workflow-specific data on execution runtimes, transmission data sizes, latencies, and conditional invocations. Additionally, we need additional information to help fill this gap for all region-to-region latencies where the workflow still needs to be deployed. We split this data because if there is workflow data on a region-to-region latency, this data always takes precedence over the simulated data.
- **Carbon Calculator:** Requires the carbon intensity forecast for the next 24 hours to model the carbon intensity for the required hourly granularity. Additionally, the calculator requires the sampled execution runtimes and transmission sizes, which the `MetricsModeler` passes as parameters based on the sampled data from the latency calculation. Lastly, the carbon calculation must include the average CPU utilization and node-specific IO usage.
- **Cost Calculator:** Requires the per-provider and region-specific cost data as well as the sampled execution runtime and transmission sizes.

Data Collection

For all metrics that the `MetricsModeler` can model end-to-end, a subcomponent must first retrieve the raw data required for this modeling. Following, we will outline the four required `DataCollector` implementations to fill the required data in the tables that the corresponding calculator retrieves for end-to-end modeling.

Workflow Invocation Data: The information on a workflow level, including CPU utilization, execution runtimes, transmission sizes and latencies, and invocation probabilities, are collected utilizing extensive logging of past invocations facilitated by the function wrapper and obtained through AWS CloudWatch logs [10]. When the execution runtime for a new region is unavailable, the corresponding calculator uses the home region’s execution time samples. The data is collected when triggered by a request from the DM. The `WorkflowCollector` extracts the data by retrieving all logs since the last synchronization iteratively. The raw logs are transformed into invocation samples that contain all the information on one invocation, maintaining all the information on one invocation in one place to maintain

information such as deployment, invocation probability of edges, end-to-end latency, and more. This granularity of keeping the logs requires a per-log way of keeping or forgetting past information. Since we know the invocation date for every log and maintain the total number of logs, a fine-grained forgetting strategy could be enabled. The MM maintains a list of every workflow’s invocations in distributed storage for the last thirty days and, at most, for the 5,000 latest workflow executions. Cold start data is filtered out as tainted and not part of the invocations. If more than 5,000 invocations are stored, it starts selectively forgetting the oldest ones. That means only invocations representing DAG information (e.g., region-to-region latency) not present in new data are maintained; the collector removes all others. The collector aggregates the logs per workflow information on every node and edge level, and execution or transmission regions are maintained. We maintain the transfer size distribution as a list of sizes since this is relevant for transmission carbon. This aggregation step provides the data to retrieve a per-node and edge list of latencies run, times, and input sizes, which the MM later retrieves for modeling the metrics.

Region to Region external Latency Data: Without historical data, the framework defaults to using *CloudPing* [148], offering AWS-specific ping data to estimate transmission latency and *WonderNetwork* [185] for missing transmission latency data in CloudPing. Data may be missing from CloudPing if the data center is new or between regions that AWS turns off by default, a cost-saving measure by AWS to prevent unwanted costs for the customer. The **PerformanceCollector** stores 100 samples for each transmission between two regions. The data is collected weekly to catch grave performance swings in region-to-region latency.

Carbon Data: The **CarbonCollector** retrieves the carbon data for the past seven days from the API provided by ElectricityMaps [73]. We maintain the data in an hourly granularity. Since past data alone cannot make good future predictions, we have to apply carbon forecasting. Since the *Deployment Solver* does not generate a new solution every hour for every workflow, we need some window into the future. The collector needs to forecast carbon as the diurnal patterns exhibited by several regions indicate that the best deployment for carbon objectives might shift throughout the day but have slow seasonal shifts. In that case, to ensure that infrequently solved deployments may still benefit from the CARIBOU framework, carbon forecasting is used for reasonable future data prediction. The collector accomplished this by utilizing Holt-Winters Forecasting Exponential Smoothing [113] from Statsmodels [193] once every day using the hourly carbon intensities of the last seven days of all regions for forecasting. The data is collected every 24 hours at midnight for the next day to maintain

the most up-to-date carbon data for any solution occurring in the next 24 hours.

Provider Data: The `ProviderCollector` retrieves the data from the AWS price list interface [28] that returns the invocation, execution, SNS, and egress costs per region. Additionally, this collector maintains the list of available regions in the `available_regions_table`. The data is collected irregularly since the pricing strategy needs to be updated more frequently to warrant more regular updates.

Chapter 6

Framework Evaluation

We established a complex framework with many components in the design and implementation section. Since this thesis aims not to propose the most efficient solution but to act as a proof of concept, we will focus on showing the efficacy of a framework for reducing carbon by geospatial offloading to answer our research question and our attempts to reduce the framework overhead simultaneously. Future evaluations could go more fine-grained and evaluated in detail, as well as component-wise efficiency and performance. For this evaluation, we will keep a birds-eye perspective while considering the uncertainties in modeling data transmission carbon (§2.6.3).

6.1 Experimental Setup

We undertake two kinds of evaluations: The first focuses on the effectiveness of our framework in exploiting the regional carbon intensity differences between regions. The second focuses on the framework’s overhead, which we aimed to reduce to a required minimum.

Each experiment consists of two phases: The *first phase* is **data gathering**, meaning we deploy the workflow to an initial home region. Unless stated otherwise, we choose `eu-central-1` and run approximately 200 invocations for each benchmark in that region. We then run the workflow collector sub-component of the metrics manager to collect and aggregate the workflow invocation logs. We also expose the metrics manager’s carbon collector to subsets of carbon data to generate corresponding carbon forecasts for the experimentation days. The experimentation period is from October 15th to 21st, 2023, since we consider this a representative period concerning carbon differences between the regions and outline the carbon intensities of that period in Figure 2.2. Concerning the function execution IO carbon intensity, we statically deploy the data access to the corresponding home region and calculate the data transmission accordingly as part of the transmission of carbon emissions. The *second phase* is **running the solver** to generate

deployments. The solver is, depending on the specific plot configuration, able to offload up to five regions, `eu-central-1`, `eu-south-1`, `eu-west-3`, `eu-north-1`, and `eu-central-2` unless stated otherwise. In a second set of experiments, we choose `us-east-1` as home region and `us-west-1`, `us-west-2`, and depending on experiment `ca-central-1` as offloading regions to gather additional insights and prove the viability of our framework in a different setting. Then, we expose the deployment solver to the day-specific metrics and run the solver to generate a deployment for that specific day for every hour, generating the most fine-grained deployment for every hour of the day. The solved deployments, the actual carbon data, the runtimes for each node, and the latency data between the regions allow us to calculate the carbon data reported in the plots. We refrain from showing actual carbon numbers since the uncertainty in carbon modeling would not allow for definitive statements, but we provide arguments for comparative evaluations.

We additionally made the following decisions to ensure a fair evaluation:

1. We limit the evaluation period from the 15th to the 21st of October for a fair cross-evaluation comparison.
2. We choose random subsamples of the indicated data sources in both a "big" and "small" size, showing two extremes concerning input sizes.
3. Requests originate from a fixed source (`eu-central-1` or `us-east-1`). Fixing the source traffic location is critical to present meaningful results since end users and trigger sources do not move around to different regions when we offload a workflow in the real world. Geospatial offloading can have significant transmission latency, cost, and carbon implications, which are especially highlighted when the invocation source is fixed.
4. When comparing the frameworks against alternative solutions, we ensure that all solutions receive the same configuration and that the solution-specific implementation overhead is minimal.
5. We control for the CPU architecture (Intel Xeon Processor @2.5GHz) since this can vary depending on the AWS-assigned function execution environment and lead to very different results with regards to execution runtime that would make comparable statements hard to make [57].
6. Additionally, to control for performance fluctuations within regions at different times of the day, we collect data in multiple regions for each execution node. We use the whole set of data for the shown plots. While this does not eliminate any cross-regional variations, it is a best-effort attempt to control for such differences.
7. Wherever helpful, the experiments show the best and worst case model regarding transmission carbon (§4.4.2).

6.1.1 Experiment Data

We outline the data for the experiments per workflow in the workflow-specific subsections. The input data used for all benchmarks exemplify realistic workloads. We attempt to showcase the framework’s behavior in different input cases by evaluating both a large and small input size. In reality, data input size may vary significantly and undergo distribution shifts during the lifetime of a workflow. The carbon database provides the ElectricityMaps data with historical carbon data [73]. We collected the home region executions between April 19th and April 26th, 2024. We use a uniform invocation pattern to evaluate trade-offs and high-level aspects. Larger scale, continuous evaluations were based on the 2021 Azure Functions invocation trace [31, 226].

6.2 Benchmark Workflows

We implemented five benchmark workflows, ranging from simple, single-function workflows to complex, conditional workflows inspired by previous research. These workflows exemplify serverless function use cases that may exist in the real world.

6.2.1 DNA Visualization



Figure 6.1: DNA Visualization workflow structure.

DNA visualization [55] represents a simple, single-function workflow. The workflow, when given a GenBank input file (`.gb`), visualizes the corresponding DNA features using the *DNA Features Viewer* Python library [70]. The input DNA sequences are *Borrelia burgdorferi B31 plasmid cp32-6* (69KB) and *Nanoarchaeum equitans Kin4-M* (1.1MB), respectively, from the genetic sequence database provided by the *National Library of Medicine* [162].

6.2.2 Image Processing

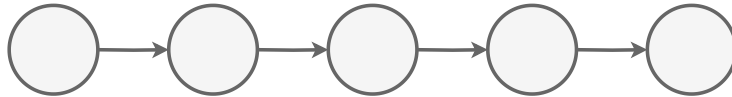


Figure 6.2: Image Processing workflow structure.

Image Processing [116] represents a pipeline of actions applied on an input image using the Python imaging library fork of Pillow [53]. The steps

are 1) flip, 2) rotate, 3) filter, 4) greyscale, and 5) resize. The input image is an image of a painting by Daniel Long [174] downsampled (222KB) and in full resolution (2.4MB).

6.2.3 Text2Speech Censoring

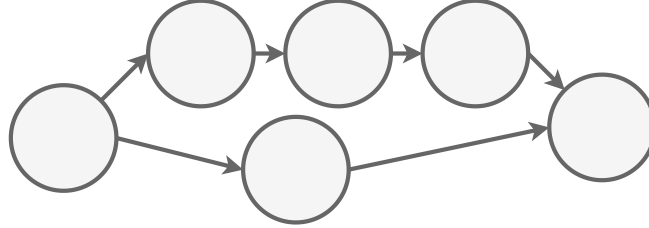


Figure 6.3: Text2Speech Censoring workflow structure.

Text2Speech Censoring [71] implements the motivational example outlined in section 3.1. The input is two random subsamples of the Yelp Reviews dataset [221], specifically including reviews containing expletives of size 1 KB and 12 KB, respectively.

6.2.4 Video Analytics

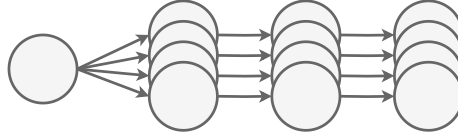


Figure 6.4: Video Analytics workflow structure.

Video Analytics [211] performs object recognition on video stream frames. The workflow first splits the input video into four video sequences. The workflow performs resizing, decoding, and object recognition parallel to the four sequences. The object recognition task is done using SqueezeNet [103] through `torchvision` [181]. The experiment video is based on the INO video analytics dataset [107] with an example video of 206KB and 2.4MB, respectively.

6.2.5 MapReduce

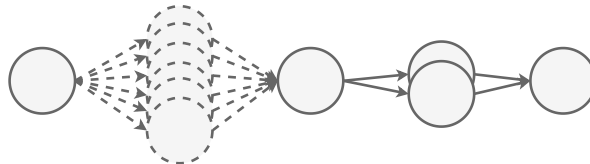


Figure 6.5: MapReduce workflow structure.

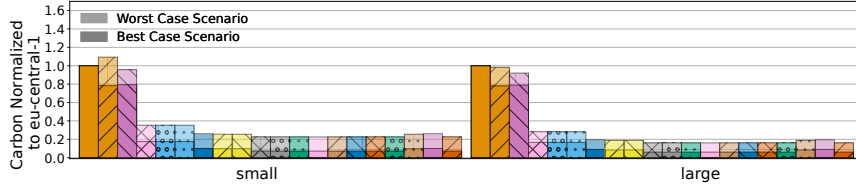
MapReduce [85] implements the famous paradigm for analyzing large-scale datasets. We shard the input dataset into smaller subsets. Based on the input size, the start node determines the required number of workers and schedules a workload of at least 12.8MB for each mapper. If fewer than the six maximum workers are required, the others are conditionally not invoked and thus do not generate costs or overheads. The workers are then assigned subsets of the shared dataset that they download, and each applies a word count function. The data is then forwarded to a synchronization node called a "shuffler." This node assigns a subset to two reducers, which group and count the words accordingly. The reducers forward the respective results to an output processor that combines and stores the results. The input files are two representative subsamples of the Yelp reviews dataset [221] of size 25.6MB, resulting in two mappers, and 256MB, resulting in six mappers, respectively. As previously outlined, we had to manually adjust the IO carbon emissions since our framework cannot collect that information from the logs. This adjustment was especially relevant for MapReduce, where we had to adjust the IO for the worker nodes that load significant data from S3. As we will see in the figures, this results in significant carbon overheads from transmission.

6.3 Carbon Related Evaluation

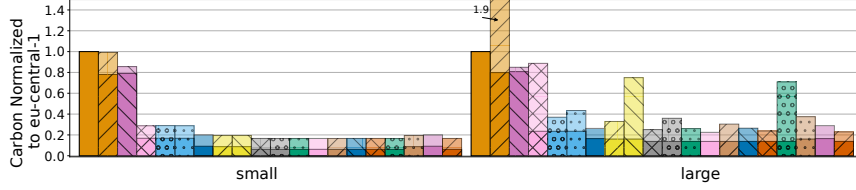
6.3.1 Effectiveness of Geospatial Shifting

Figure 6.6 shows the carbon savings compared with running everything in `eu-central-1` using manual static deployment and CARIBOU with different sets of regions. For this specific experiment, we did not consider the dynamic aspect of the system of solving only every couple of days based on the number of invocations. Instead, we consider the carbon differences if the framework would solve for a new deployment each time at midnight and use this deployment for the next 24 hours. We can draw the following insights from this overview:

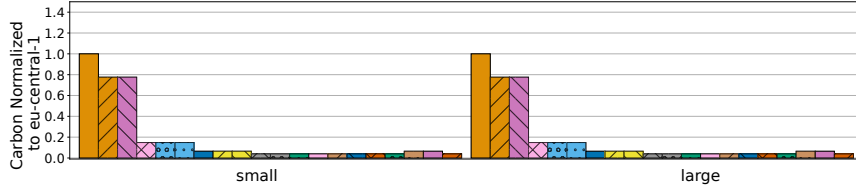
- **Insight 1: Static deployment to regions with lower carbon intensity does not necessarily reduce a workflow’s carbon emissions.** All regions have lower average carbon intensity compared to `eu-central-1` in the experiment period. Naively deploying workflows in either region without considering application-specific data transmission carbon overhead can lead to significant excess carbon emissions depending on the input size and the transmission carbon scenario. The cost of naively deploying is exemplified in Figure 6.6a where naively deploying to `eu-south-1` would lead to excess carbon in the worst case model but being adaptive between `eu-south-1` and `eu-central-1` can lead to carbon savings. Additionally, any static deployment misses out



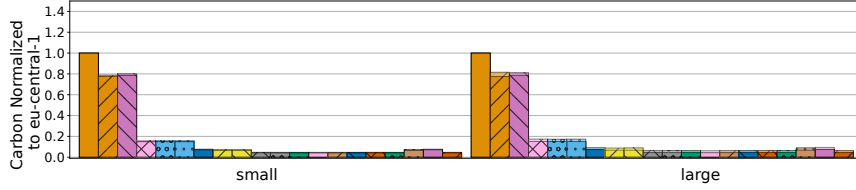
(a) DNA Visualization



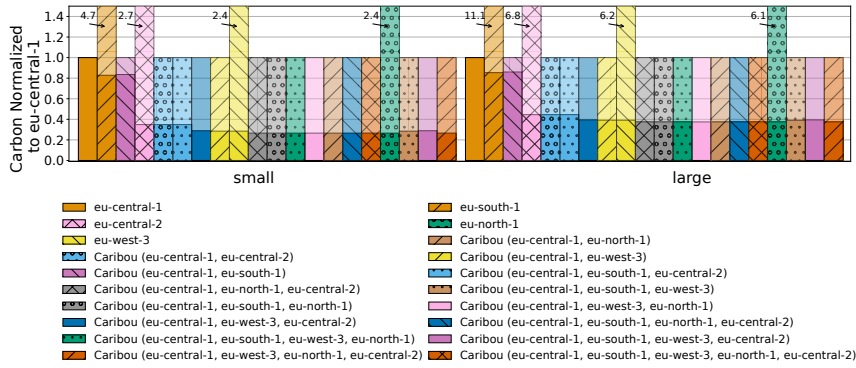
(b) Image Processing



(c) Text2Speech Censoring



(d) Video Analytics



(e) MapReduce

Figure 6.6: Normalized relative carbon to deploying the workflow in eu-central-1 with the two sets of input sizes (small and large) and the two data transmission carbon scenarios (translucent being energy factor 0.005 kWh/GB, no within region carbon and fully colored being 0.001 kWh/GB and same within region).

on leveraging cross-regional carbon intensity variations (§2.1.3).

- **Insight 2: An adaptive framework is needed to control geo-shifting across workflows or even for different inputs of the same workflow.** CARIBOU tames the spikes in excess carbon emissions through not offloading data transmission heavy applications such as MapReduce in the worst case, where a non-adaptive framework can cause significant overhead. This example additionally shows that adaptiveness is needed not only within an application but also depending on the input size given to that application. For example, Image Processing as seen in Figure 6.6b under the worst-case model given a small input could benefit from being singularly offloaded to `eu-south-1`, however, given the large input sizes that strategy suddenly generates more carbon than simply staying at home.
- **Insight 3: Benefits of geospatial shifting depend on the mix of available regions.** The benefits of the mix are due to the different and varying carbon intensity patterns in various regions (§2.1.3). Suppose we zoom into the three two-region combinations for the Image Processing benchmark. In that case, depending on the combination, we observe that the reductions vary but are all greater than those of any individual involved region. We see additional significant gains if we extend the two-region combinations with a third, low-carbon region such as `eu-north-1`. This offloading would not have happened with compliance constraints.
- **Insight 4: Holistic geospatial shifting offers substantial gains.** CARIBOU considers many novel aspects in determining carbon-optimal deployments (§4.5). The significant carbon reduction offered by CARIBOU – up to 90.2% (geometric mean) from the experiments conducted for Figure 6.6 – warrants serious consideration of geospatial shifting. More research is needed to unlock even more savings for serverless workflows and, more importantly, pave the way for extending techniques presented in this work to new workloads.

Additionally, we run the same experiments on the North American continent to retrieve additional insights from different sources. There the difference between the three US American regions, `us-east-1`, `us-west-1`, and `us-west-2` is much more minor with similar carbon intensity patterns such as the UK or Spain due to a strong focus on Solar and Wind for renewables (`us-east-2` is in the same electrical grid as `us-east-1` thus not explicitly mentioned). The Canadian region, `ca-central-1`, has a pattern similar to Sweden, thanks to a large amount of hydropower. The results, as seen in Figure 6.7, from the same experiment as outlined in this section with `us-east-1` as the home region in the period from 15th to the 21st of October 2023 show similar results as in Europe, strengthening the insights gained. Here, the geometric mean of maximal carbon reduction from this experi-

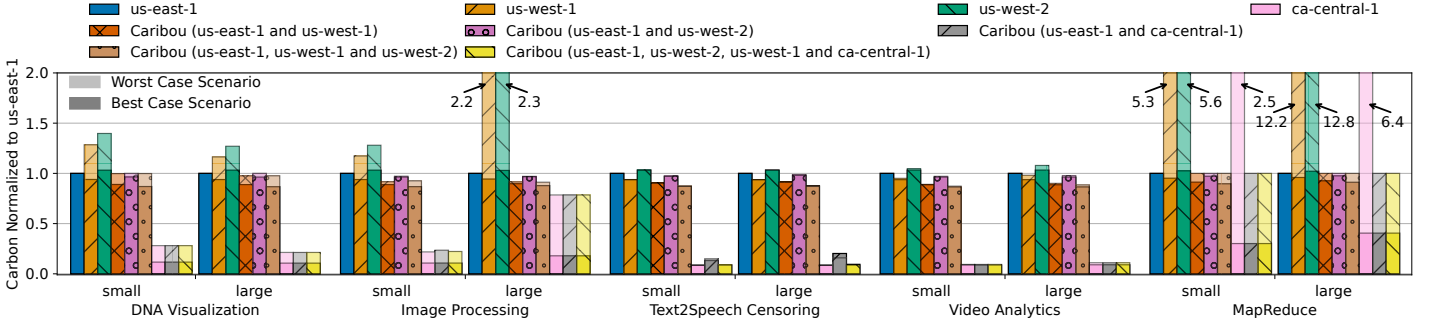


Figure 6.7: Normalized relative carbon to deploying the workflow in `us-east-1` with the two sets of input sizes (small and large) and the two data transmission carbon scenarios (translucent being energy factor 0.005 kWh/GB, no within region carbon and fully colored being 0.001 kWh/GB and same within region).

ment is 87.9% due to slightly less stark carbon intensity differences between the chosen home region and the best region.

6.3.2 Application Structure and Carbon Savings Potential

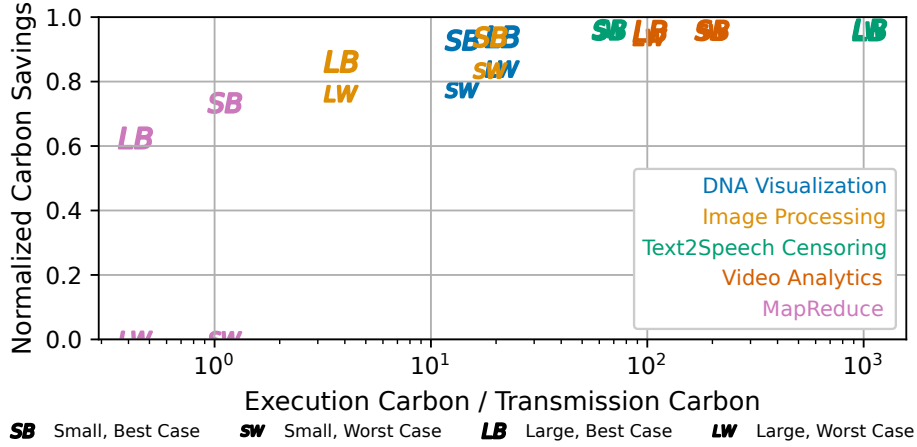


Figure 6.8: Execution / Transmission ratio compared with potential carbon savings when shifting geospatially.

The effectiveness of geospatial shifting depends on the application's compute-to-transmission ratio. A more transmission-heavy workflow will benefit less from geospatial shifting than a compute-heavy one. Figure 6.8 additionally showcases this relationship, where we compare the normalized carbon savings over the execution to transmission ratio. For this experiment, the

solver could offload to the European regions. We calculate the ratio using our modeled energy usage data based on collected workflow execution data.

6.3.3 Carbon Efficiency and Latency Tolerances

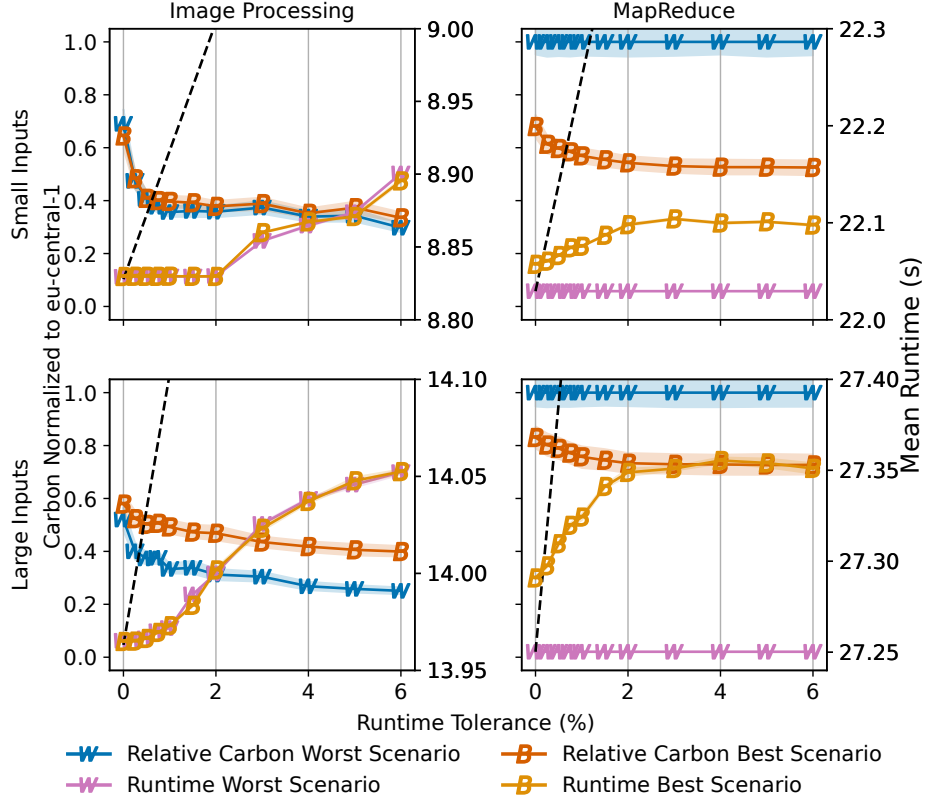


Figure 6.9: Carbon emissions under different latency tolerances when using CARIBOU to determine optimal deployments by enabling the solver to potential offload to all outlined regions (§6.1). The black dotted line represents the QoS.

Tolerances are a feature of our framework that enables more developer flexibility and is a tool to enforce QoS promises. Incorporating tolerances on deployment generation, such as workflow end-to-end latency (§5.5.1) on the generated deployments, enables developers to meet Quality of Service (QoS) requirements while still benefitting from carbon reductions. In Figure 6.9, we evaluate how different workflow end-to-end latency tolerances influence the deployment decisions made by the deployment solver given complete flexibility to offload to the European regions, compared to a single region deployment in `eu-central-1`. We showcase the effects of the Image Processing and MapReduce benchmarks since they showcase inter-

esting workflow structure characteristics. Image Processing is a chain of functions with no parallel sections, while MapReduce has multiple parallelized steps (both map and reduce). As the latency tolerance of workflow execution increases (from 0% to 6% over the deployment in the home region `eu-central-1`), the freedom of the framework to offload parts or the whole application to regions with lower carbon intensity also increases. The framework generally observes the QoS tolerances, with minimal violations in the MapReduce best-case scenario for large input sizes (less than 0.2The decisions the *Deployment Solver* makes reflect a conservative end-to-end latency modeling. Interestingly, even though the tolerances are very low, the algorithm still offloads even in the Image Processing case where there is no parallelization. We explain this behavior by the relative closeness of the region where the algorithm offloads to `eu-central-2`. In Subsection 2.6.1, we outlined that the ping between `eu-central-1` and `eu-central-2` is all between 3.1ms and 4.9ms.

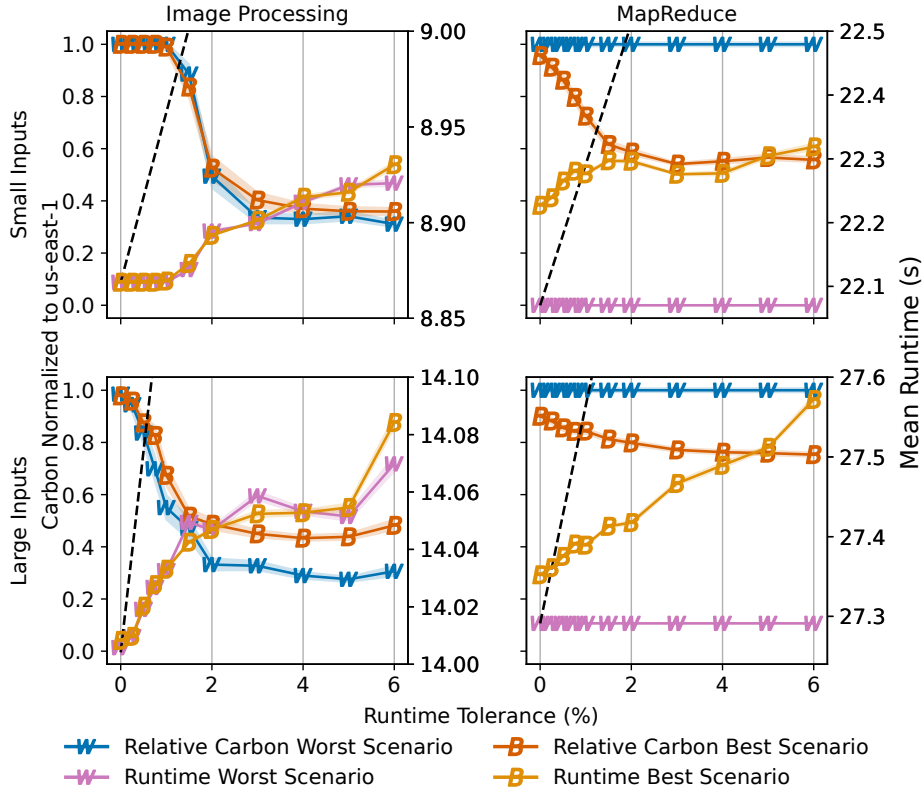


Figure 6.10: Carbon emissions under different latency tolerances when using CARIBOU to determine optimal deployments for `us-east-1`, `us-west-1`, `us-west-2`, and `ca-central-1`. The black dotted line represents the QoS.

The evaluation is additionally supported if we consider the North Amer-

ican regions previously introduced where the latencies are more significant due to the longer distances (mean 17.7ms from `ca-central-1` to `us-east-1` according to CloudPing on May 4th, 2023 compared to 6.6ms within `ca-central-1`). In Figure 6.10, we evaluate the resulting deployments from increasing the latency constraints on the North American continent. Here, the algorithm generally does not offload for small latency tolerances for Image Processing since this benchmark does not offer any offloading of stages off the critical path. Conversely, MapReduce offers offloading even at 0% tolerance since it consists of parallel workflows. However, the large input size makes it more sensitive to the data transmission scenario. While the violations incurred are more significant (0.9%), they mostly show that MapReduce’s complexity has not been modeled accurately enough compared to the experimental runs.

6.3.4 Efficiency of Self-Adaptive Re-Deployment

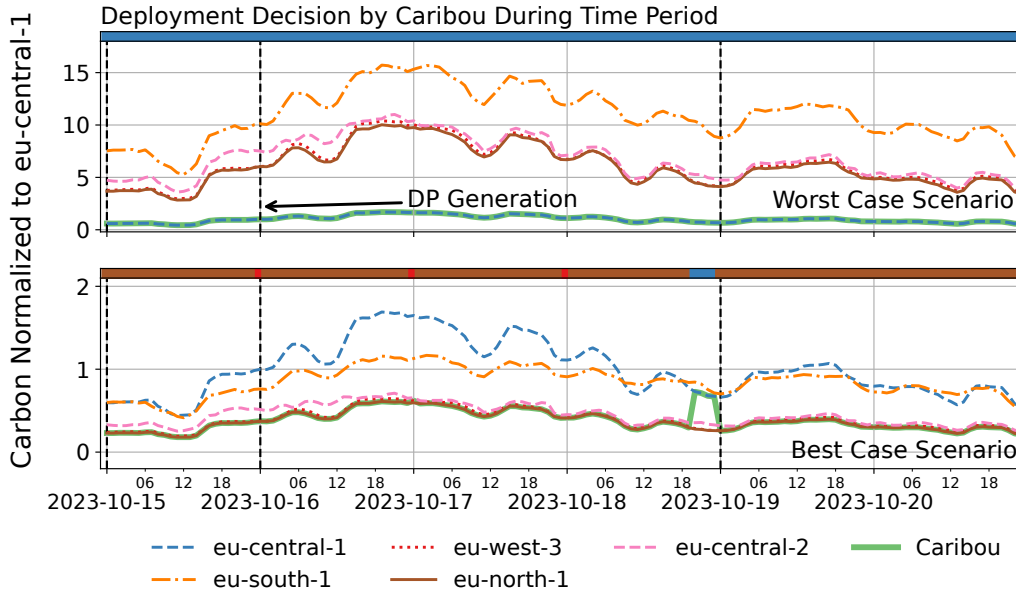


Figure 6.11: Showcasing CARIBOU’s deployment decisions for MapReduce using the large input size, normalized to mean carbon emission of execution in `eu-central-1`. The line on top of each plot indicates our framework’s deployment decisions.

Figure 6.11 captures the week-long operation and visualizes the decisions made by the deployment solver over time. In this experiment, we simulate the invocations based on a specific invocation pattern for one week. At points determined by the metrics manager, we initiate a re-solve that generates new deployments for the next period. These deployments are then

used to calculate the effective relative carbon emissions, as we report in the figure, similar to the previous figures. The invocation pattern is a representative application of the Azure traces dataset [31] with approximately the 5th percentile invocation count reported from Azure characterizations [143]. We showcase only this here since MapReduce offered the most variability between the two transmission scenarios. However, all other benchmarks are listed in Appendix E.1. We mark new deployment generation points with vertical lines as part of a deployment plan (DP). Initially, the framework enters a learning phase, optimizing deployment regions daily and transitioning to a lower frequency schedule, showcasing that a new solution would likely bring slight improvement and that the system stabilizes. Generally, CARIBOU generates deployments that offload the workflow to the lowest-carbon region for that time, both in the best-case scenario where all three regions are viable offloading options and under the worst-case scenario where, due to the high transmission carbon overhead of the significant input, keeping the workflow in `eu-central-1` presents the sole feasible solution. The framework deployment expired on the 18th after 6 pm, causing it to default to the home region.

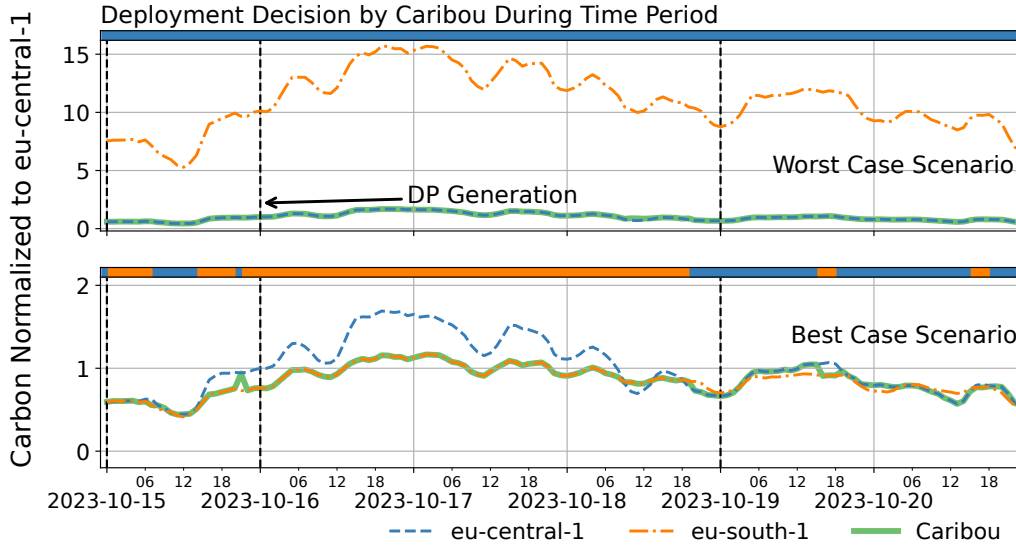


Figure 6.12: Showcasing CARIBOU’s deployment decisions for MapReduce when the region selection is limited to `eu-central-1` and `eu-south-1`.

Similarly, to investigate the system’s feasibility when adapting between regions with similar carbon intensity patterns, we limited another experiment to `eu-central-1` and `eu-south-1`. The evaluation shows that the framework can go for the most feasible region at most times. Notably, the defaulting to `eu-central-1` on the 18th after 6 pm leads to a better result since this region has the lower carbon intensity during that time frame.

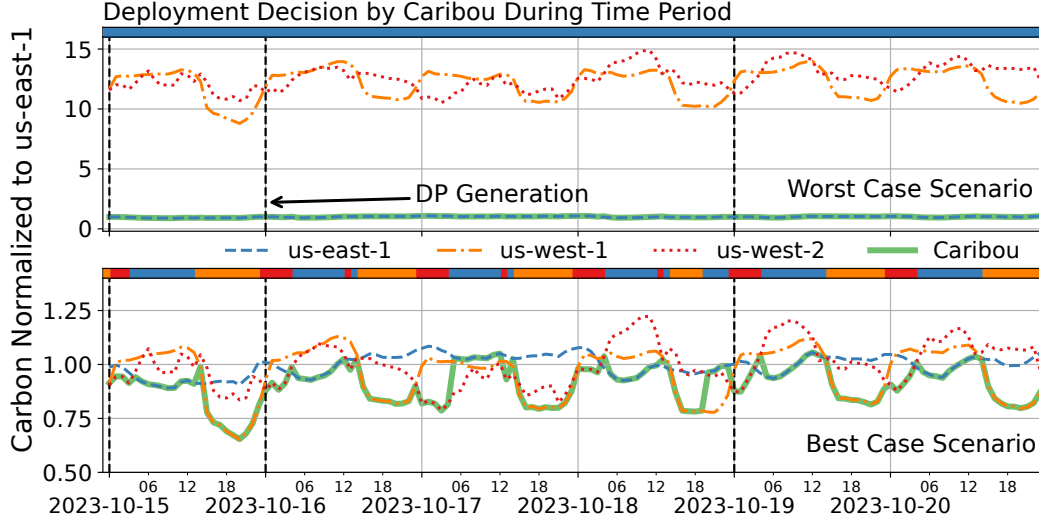


Figure 6.13: Showcasing CARIBOU’s deployment decisions for MapReduce when the region selection is limited to `us-east-1`, `us-west-1`, `us-west-2`, and `ca-central-1`.

If we compare the results with the North American framework deployment decisions, where the carbon intensities overlap, we get the result as seen in Figure 6.13. Here, we can observe a notable limitation in carbon emission forecasting on October 17th between 6 am and midday in the best-case scenario, attributed to a carry-over from the previous day’s inaccurate prediction. Additionally, identical to the previous evaluations, the deployment expired on the 18th after 6 pm, causing the framework to default to the home region, `us-east-1`.

6.3.5 Feasibility of Dynamic Deployment Solve

To evaluate the feasibility of our dynamic deployment solutions, we first have to show that the carbon predictions using Holt-Winters Forecasting stay the same if reused over multiple days. For this, we experiment during the same period as all other experiments. We take the carbon forecast for a day and calculate the mean absolute percentage error between the prediction and the actual data for one to seven days. We observe in Figure 6.14 that while the error does increase slightly for some regions, namely `eu-central-1` and `eu-central-2`, it has a slow upward trend where the prediction is the best for the first 24 hours, as expected and then slowly becomes worse, up to 60% MAPE. At the same time, the prediction for `eu-south-1` and `eu-north-1` do not follow a clear trend. These findings support our implementation, where deployments can be reused over multiple days, but they also support our conservative approach, which is to retire a deployment after a while.

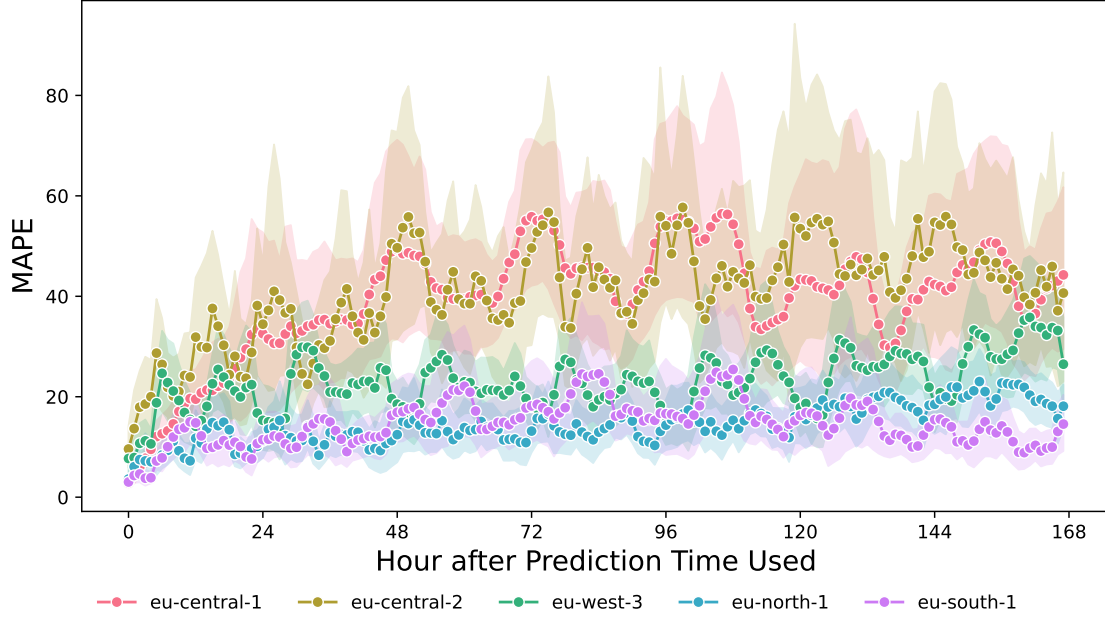


Figure 6.14: Evaluating the mean absolute percentage error of the carbon forecast when reused for up to seven days.

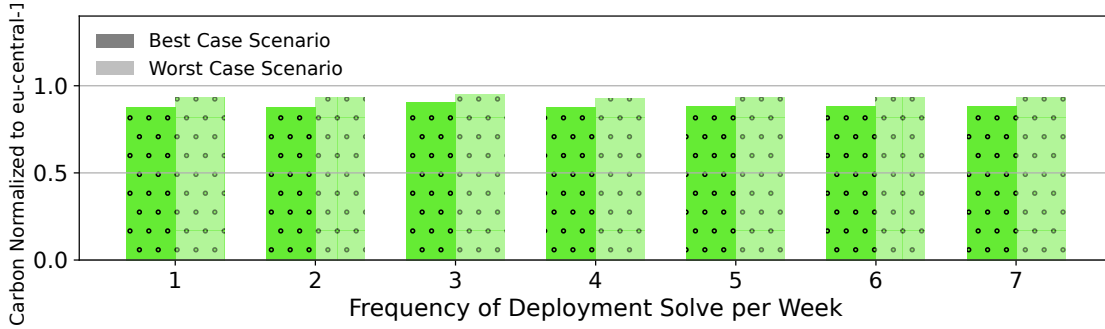


Figure 6.15: We normalized relative carbon to deploying the Image Processing workflow in `us-east-1` with different solving frequencies per week when offering `us-east-1`, `us-west-1`, and `us-west-2` as potential offloading locations. The solving frequency determines how many days a deployment is reused before being replaced.

While we observed that the prediction quality does not significantly worsen over time, it is essential to evaluate how many days a deployment can be reused and whether this reuse influences a workflow’s normalized relative carbon emissions. In Figure 6.15, we are doing just that by changing the frequency of deployment solve per week between one and seven, where one means we reuse the same deployment for a whole week, whereas seven

means the deployment solver generates a new deployment every day. We focus on the Image Processing benchmark; the results are similar for all other benchmarks. We choose the regions `us-east-1`, `us-west-1`, and `us-west-2` as offloading locations since these three vary overlappingly over a week, as we can observe in Figure 6.13. We observe no clear trend of better or worse results when solving more per week. This insight supports the idea that having an adaptive approach influenced by the number of invocations makes sense since solving less will not result in significantly worse results.

6.4 Other Objectives

While reducing the carbon emissions from cloud applications through geospatial deployment was the primary goal of this thesis and necessary to answer the research question, a stated side objective was to provide a comprehensive framework that can help developers reach their objectives. Besides carbon, we support end-to-end latency and cost as objectives. We have already shown that the tolerances the developers can define work (§6.3.3). However, we also want to explain briefly whether the other objectives work. The main difference between carbon and the other objectives is that the other two are relatively more static. The cost of invocations and executions change over time, but very slowly, and AWS rarely uproots its pricing strategy overnight. Similarly, execution times and transmission latencies might change due to relative pressure at a data center or transmission cable, but unless these changes follow a clear pattern that we can capture with our past data approach, we currently have no way to model them; doing so would go beyond the scope of this thesis. Additionally, since we do no explorative deployment, meaning collecting data from locations we have no data in, if there is no clear indication from our obtained latency, cost, or carbon data that this region is beneficial, we cannot capture this new data. In this section, we attempt to convey that our fundamental strategy still works by showcasing the very static case of the cost objective. Let us take cost as the number one objective. In this case, the deployment algorithm shifts the deployment decisions to all remain in the home region of `eu-central-1` since this region, same as `eu-north-1`, `eu-west-1`, `eu-west-2`, and `eu-west-3`, has the lowest cost in Europe and additionally causes no egress fees. We test and verify the behavior in tests, and the solver can correctly make this decision. Similarly, if end-to-end latency is the primary objective, the work will remain in `eu-central-1` since this does not incur any transmission overhead. However, suppose the end-user calls to shift to a different region. In that case, the framework adapts wherever incoming calls can be handled the fastest, incurring the lowest end-user-to-framework latency. Thus, we prove that the framework can also handle the more static objectives.

6.5 Framework Overhead

When evaluating the overhead of our proposed framework, we must prove that the framework itself does not generate excessive cost, carbon, and end-to-end latency overhead. Otherwise, running a workflow with the framework will never be viable. All the following overheads are calculated by running the evaluation in `ca-west-1` to control for the hardware more strictly across implementation variants. The processor in that region is always an Intel(R) Xeon(R) Processor @ 2.90GHz.

6.5.1 Invocation End-to-end Latency Overhead

We evaluate the framework’s latency impact by comparing the benchmarks implemented with the AWS-specific workflow orchestration tool, AWS Step Functions, and a simple SNS-powered solution for invoking successor functions. The latter prohibits us from implementing any workflows with synchronization nodes since implementing that logic would again propose a specific solution and most likely need more insight. However, evaluation against it in the non-framework benchmark allows us to quantify the overhead of our function and workflow wrappers compared to a vanilla SNS implementation. AWS Step Functions is a first-party service by AWS that is explicitly for serverless workflow orchestration. It offers a rich toolset without the option to offload to different regions other than deploying the whole workflow to another region.

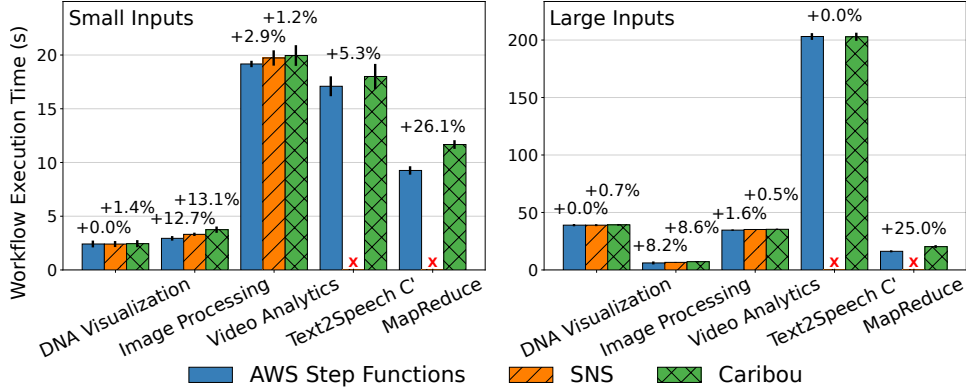


Figure 6.16: Comparison of workflow execution time between AWS Step Functions, Amazon SNS, and CARIBOU. SNS is the function-to-function messaging channel in CARIBOU but does not support synchronization.

Figure 6.16 shows that AWS Step Functions offers the lowest workflow execution times, geometric mean 4.97% and 3.21% faster than the SNS-only implementation for small and large input sizes, respectively. We argue that this difference comes from AWS’s ability to cluster the function executions

closer together since they know the order of execution that needs to be indicated when creating a step function. Access to the whole stack allows AWS to perform optimizations on the provider level that are unavailable for clients. CARIBOU conversely introduces geometric mean 5.09% and 3.20% overhead over the SNS-only implementation, showcasing the additional logging and logic the function and workflow wrapper introduces. Additionally, the start node and any synchronization node need to make additional calls to the distributed key-value store dynamo DB. This overhead can be hidden away by long execution times, as seen when switching from small to large input sizes. Additionally, as expected from the additional computation, the relative overhead is more significant for complex workflows such as MapReduce with two synchronization nodes and conditional invocations than DNA visualization.

6.5.2 Invocation Cost Overhead

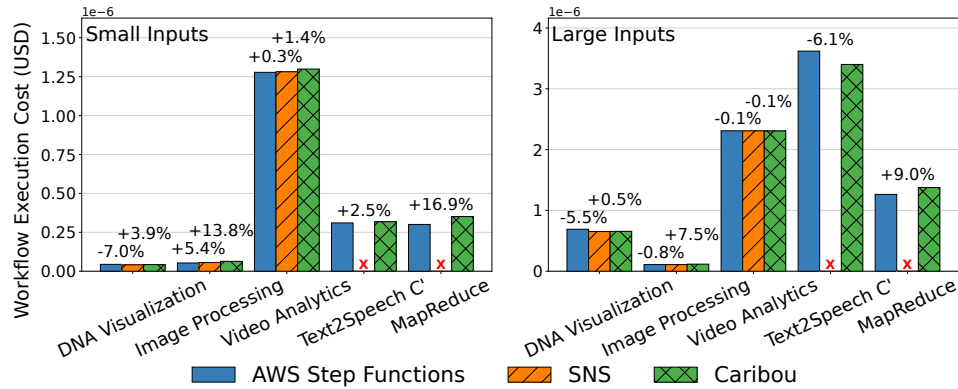


Figure 6.17: Comparison of workflow cost per invocation between AWS Step Functions, Amazon SNS, and CARIBOU.

Similarly to the end-to-end latency overhead, we compare the framework’s impact on cost by comparing the five benchmarks with AWS Step Functions and a vanilla SNS implementation.

Figure 6.17 shows the cost overhead of running the workflows with our framework. AWS Step Functions is not always the cheapest solution, with a geometric mean cost of 0.59% and 2.17% more than the SNS-only implementation for small and large input sizes, respectively. This difference is mainly due to the additional duration cost of using step functions. AWS Step Functions incur additional charges (on top of the AWS Lambda charges) where the execution time is rounded to 100ms compared to the AWS Lambda 1ms as well as charging for 64-MB chunks of memory instead of the more fine-grained approach is solely pure Lambda [21]. While CARIBOU adds geometric mean 6.23% and 2.58% overhead for small and large input sizes over

the vanilla SNS solution by firstly being slower and additionally introducing overheads for fetching the deployment plan as well as additional overhead for the synchronization points in Text2Speech Censoring and MapReduce. The overhead is relatively small, especially for long-running functions with few synchronization points. In contrast, the overhead difference between CARIBOU and AWS Step Function is less significant than the runtime overhead thanks to the more fine-grained billing and the low-cost overheads of the additional components introduced. We also have to consider the cost of running the solver, which is amortized over the number of invocations.

6.5.3 Invocation Carbon Overhead

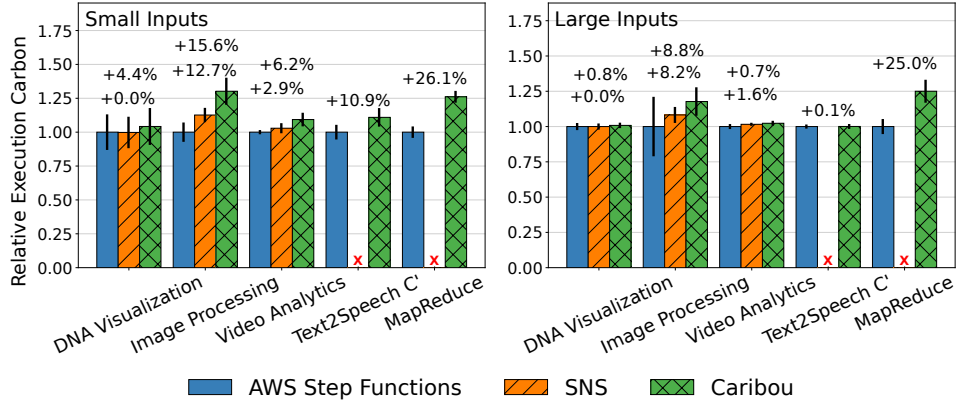


Figure 6.18: Comparison of workflow relative carbon per invocation between AWS Step Functions, Amazon SNS, and CARIBOU.

When evaluating the carbon overhead of invoking a workflow deployed in the framework, we must compare the added data transmissions and execution runtimes, similar to the previous overhead experiments. Given the lack of information about how AWS Step Functions implements synchronization points, we make a best-case assumption. In this scenario, we assume no additional carbon overhead is added, which could potentially lead to a significant underestimation of the actual carbon overhead. We assume that data transmission for all technologies adds the same overhead. We chose the best-case scenario for offloading since all deployed source code functions are in the same region. Thus, choosing the worst case would result in no carbon overhead from any data transmission. Approximately the same data size is transmitted from one function to another, with a slight overhead for our framework, which we account for in our evaluation since we are forwarding the deployment plan (DP). The synchronization nodes for CARIBOU add another overhead of storing and fetching annotation and intermediate data. However, the added carbon emissions are negligible since the data sizes are

tiny. We assume that the overhead of SNS to the transmission technology used for Step Functions is similar since we have no way of modeling that overhead at this point. The evaluation gives a geometric mean of 4.97% and 3.21% for small and large input sizes, respectively, for more carbon emissions from the SNS implementation compared to Step Functions and a geometric mean of 18.27% and 11.85% for small and large input sizes when comparing Step Functions to CARIBOU. We also assume data locality, where all the data remains in the home region, and we assume the best case for offloading with the transmission coefficient at 0.001. In Figure 6.18, we compare the relative execution carbon to AWS Step Functions. The main differences compared to the execution time plot are the additional overhead of transmitting data as part of either synchronization nodes (small overhead due to tiny data sizes) or the overhead of the DP transmission (relative impact depending on workflow executions and transmissions). Overall, the overhead is most significantly visible for the Video Analytics workflow due to the number of transitions where the DP has to be forwarded and the large size of the DP for the same reason. The DP forwarding adds relatively less overhead for the significant inputs since the carbon emissions of execution overshadow any transmission.

6.5.4 Framework Solve and Migration Overhead

To evaluate the feasibility of solving and migrating in the current framework, we deploy all benchmarks to one region and run ten solves to obtain an average time to solve for a 24-hour deployment (different deployment for each hour of the day) when given the five European regions to offload to. Then, we run a migration each where the whole workflow is redeployed to a new region, one per source code function. We extract the transmission size (image size) and solving/migration time from these experiments. We run the solver and migrator as an AWS Lambda function with 1024MB of memory. We assume the solver to run in `eu-central-2` since it offers stable and low-carbon energy and the workloads are migrated from `eu-central-1` to `eu-north-1`. As we assume every invocation to this function will incur a cold start, we will need to bill an additional approximate 650ms. Transferring data into the container registry and using the image within the same region for AWS services is free; migrating an image from one region to another cost \$0.09 per GB [13]. The storage cost for the registry is \$0.10 per GB per month and thus comparably negligible. Using this, we calculated the cost of such a cycle and the carbon emissions. The latter and the deployment resulting in the lowest carbon emissions were used to calculate the break-even (iterations after which we saved more carbon by offloading compared to the solve and migration). The location where the framework components are run impacts the number of invocations required to break even due to the carbon intensity used to calculate the framework overhead. We only

consider the best case for offloading in this estimation. It becomes apparent that for workflows such as MapReduce, even solving once will not be feasible if the worst case is assumed since no carbon savings can be achieved due to the large input size.

Benchmark	Time Solve & Migration (seconds)	Image Size (MB)	Cost Solve & Migration (USD)	Carbon Solve & Migration ($gCO_2\text{-}eq$)	Mean Carbon Reduction (Large / Small)	Nr. Invoca- tions to Break Even (Large / Small)
DNA Visualization	9 + 42	309.4	0.0086 + 0.0278	0.3289 + 0.1017	93.89% / 92.82%	48 / 1126
Image Processing	281 + 160	244.9	0.0074 + 0.0220	2.8126 + 0.0803	86.22% / 93.81%	817 / 1619
Text2Speech Censoring	334 + 285	462.2	0.0103 + 0.0416	3.9462 + 0.1520	96.06% / 96.06%	161 / 3074
Video Analytics	1481 + 744	3184.4	0.0371 + 0.2866	14.1740 + 1.04075	95.60% / 95.91%	150 / 546
MapReduce	805 + 193	241.0	0.0166 + 0.0217	6.3599 + 0.0793	62.50% / 73.39%	71 / 405

Table 6.1: Benchmark Solve and Migration overhead numbers.

We list the results from these experiments in Table 6.1. For a reference point, driving a petrol car (6.5 l/100km) a kilometer emits around 260 $gCO_2\text{-}eq$ [157]. We observe that:

- **Structure** of the DAG is important. Simpler DAGs result in fast solutions and migrations since fewer nodes and functions are involved. For example, DNA Visualization is almost 98% faster in solving and migration than Video Analytics.
- **Image** size, an indication of the complexity of the involved dependencies, plays a role. Video Analytics with `torchvision` as a dependency causes significantly more overhead from migration compared to other benchmarks.
- **Invocation Data Size** directly impacts the number of invocations required to break even. Large input sizes generally break even faster since more carbon is saved from offloading, while the carbon saved for solving and migrating remains the same. This difference in input sizes to carbon savings indicates that adding this to the adaptive approach might also be valuable.

Generally, this evaluation shows once again that a dynamic approach is required that considers the workflow’s complexity, potential cost, cost of solver, and potential carbon savings. This insight strengthens our argument for a dynamic solver trigger and opens avenues for further improvements.

Chapter 7

Discussion

In this chapter, we aim to outline the broader impact of the results of this thesis, as summarized in Chapter 6. Additionally, we want to highlight specific limitations we have encountered and discuss particular aspects of the framework that have yet to receive attention.

7.1 Results

When we compare our results to static deployments in the home region, representing the status quo, and to single-region improvements, we show that a dynamic framework is required. In the best case, any other attempt would show similar improvements to our framework. In the worst case, it would cause significantly more overhead, specifically from transmission data carbon. Thus, a viable framework needs to be comprehensive and adjust dynamically to the realities of changing carbon emissions. Our framework can account for transmission carbon, making the correct deployments depending on the transmission overhead and changing application load and structure. Additionally, we have shown that our framework can adjust to changes in circumstance, be it a different carbon model, hourly carbon data, or different region selection, by remaining adaptive and fine-granular in its deployments. Furthermore, by supporting complex constraints and tolerances and being objective aware, our proposed framework enables workflow developers to communicate significant information instead of simply deploying a workflow. Our results showed that while our modeling is imperfect, resulting in minor tolerance violations and not actively reacting to tolerance violations, our framework still offers significant benefits that offset these shortcomings.

At the same time, building any framework creates overheads that we must address. In our evaluation, we quantify these overheads and how they can be significant if wholly ignored. Thus, we showcased that our framework does cause overheads while also minimizing their effects by considering when

and how to solve a new deployment. By evaluating the specific overhead per invocation, we also showcased what overhead our framework has to counteract to be viable. When implementing this overhead-aware decision-making, we showed that we know the framework’s shortfalls and have tried our best to mitigate them. We intentionally did not spend much effort on efficiently deploying and bootstrapping the framework; instead, we relied on being able to deploy the framework itself as serverless functions for a low-resource framework that only incurs costs when required.

Our evaluations are thorough and simulate as many eventualities as possible. While we ran all benchmarks and collected run data, it’s important to note that the comprehensive results maintain a somewhat synthetic nature due to the absence of a whole week or longer run of the framework. The deployment solver generated the deployments based on actual data, and the calculated carbon is based on real runs. However, the carbon emissions of an invocation were calculated from the collected data in the home region, such as execution runtimes, transmission size, and the carbon intensity of the deployment regions. A full deployment to all regions for all deployments would have been possible, but it may not have rendered much better or more insightful results. We refrained from running the benchmarks with all the generated deployments since the number of possible deployments and corresponding runs would have led to immense experiments, which would have been infeasible in the given period. Such a long-term study of the effect of our framework is outstanding. The evaluations shown as part of this thesis are sufficient to prove that offloading is feasible. However, a fully-fledged real-time study of a complex application over a more extended period might showcase results pointing to additional potential improvements or limitations of the current framework.

While we could have made many more evaluations regarding the independent components, we specifically wanted to highlight the framework as a whole; we leave individual improvements up to future iterations of the existing framework implementation. We provided specific insights from our evaluations along the way. Our results showcase the benefits of our framework and provide the environment for many more research questions.

Our implementation and the corresponding evaluation have provided the foundation to answer the thesis question, **“Can we reduce the carbon emissions of complex cloud workflows by optimizing the geospatial deployment fine-granularly?”**, with a yes with caveats. Such a system is realistic and possible, and we present the first version of it in this thesis. Our results matter since we have shown that overcoming the limitations posed by existing solutions is possible and feasible. Additionally, extending existing solutions for cross-provider deployments could benefit from our efforts in formalizing serverless workflows, the end-to-end modeling of complex workflows for the three relevant metrics, our solution for deploying, re-deploying, and executing complex workflows cross-regionally, as well as from our insights

into both carbon emissions of execution as well as transmission. Lastly, existing solutions could benefit from considering the overhead posed by their solutions themselves, which we showed to be possibly quite significant.

7.2 Limitations

While our framework is a comprehensive proof-of-concept, we must highlight the following limitations.

7.2.1 Data Transmission from Access

One major limitation of the framework’s current design is that we do not model data access within a function. Data transmission, even if within the same region, adds latency, carbon, and potential cost overheads that we currently do not track outside of added latency to the function execution. For our evaluation, we manually added an execution’s IO based on information from our experimental setup. We currently only account for input data for the transmission data, while we do not account for any data access within the function. While this is sufficient to answer our research question, there are better options for a fully functioning framework. Since we can not assume that the data accessed within a function is replicated in all potential regions, this would generate additional carbon overhead from down- and uploading the data from and to the distributed data stores such as S3 or DynamoDB. We argue that extending the API and the workflow model would not require significant changes to the current implementation. In the interest of time, this was not pursued as part of this thesis and is left open for future work.

7.2.2 Electrical Grid Influences

Based on our discussions with power and sustainable computing experts, we know one additional limitation of our proposed geospatial shifting vision. Our work assumes that the carbon intensity of electrical grids is independent of workload shifting. If our framework becomes overly popular and migrates significant workloads to specific regions, we assume this does not influence that region’s carbon intensity. This assumption only holds if the power consumption of the shifted workload is small enough compared to the overall scale of computing in that region. However, if the shifted workloads cause unexpected traffic peaks and additional, mostly fossil, energy sources must be used to mitigate those spikes, the carbon reduction effects might be inverse. While this dynamic can (and should) be modeled, unfortunately, there is currently no way to associate the marginal carbon intensity of grids with a specific consumer. Additionally, since we have no internal data on the power usage of regions or specific data centers, we need more data to

model this effect accurately. Connecting compute usage with specific energy usage spikes remains an open research question for energy and computing experts.

7.2.3 Performance Variations

During our development and evaluations, we have experienced shifts in performance, both within regions and cross-regionally. While these shifts are well-known and researched, a deployment-oriented framework cannot accurately predict these performance variations. Additionally, suppose the framework becomes overly popular, and shifts work to a few specific regions. These regions might become unavailable for new deployments or cause performance slowdowns through relative co-tenant pressure on the data center. Being able to both model and consider these effects is an open research question for data center and performance experts.

Chapter 8

Related Work

8.1 Serverless Workflow Deployment

Table 8.1 compares CARIBOU with other frameworks for serverless workflow deployment. These frameworks can be broadly categorized into the following categories:

- Provider-specific, proprietary solutions for workflow deployment such as *AWS Step Functions* [20], *GCP Workflows* [96], or *Azure LogicApplications* [32], offering a rich toolset for complex applications with additional concepts such as fan-outs. However, these solutions are naturally vendor-locked and do not support any objectives, and deployments can only happen on a per workflow basis to one region.
- Deployment and provisioning of workflow choreographies such as *xAFCL* [187, 224] and topologies such as *OpenTOSCA* [216], offering rich environments for modeling of complex cloud workflows. These works most closely resemble our workflow model and provide many similar concepts, such as multi-stage applications with control flows and synchronization points, while also adding support for additional modeling. Specifically, xAFCL also allows the fine-grained deployment of a choreography based on a solved, statically optimal plan regarding latency or cost. However, they lack the automated migration approach, requiring manual deployment, and their application modeling is explicit using either a modeling language or a graphical user interface. In contrast, developers define the workflow structures implicitly in our approach.
- Proof-of-concept ideas and initial implementation for multi-cloud serverless deployment [227] promising a framework that can be deployed fine-grained without adding any complex workflow support.
- Solutions that consider carbon for workflow deployment, such as *Green-Courier* [48] or *Carbon Aware GSLB* [144], but lack the support for the more fine-grained, comprehensive approach since they focus on

carbon aware function scheduling from a serverless function provider perspective. They focus on offloading the execution of singular serverless functions geospatially, do not consider transmission carbon, and do not support DAGs.

Framework	Objectives	Deployment Granularity	Automated Workflow Migration	Geo-spatial	Multi-Stage	Control Flow	Sync Nodes	Transmission Overhead	Supported Providers
AWS Step Functions [20]	✗	Coarse	✗	✗	✓	✓	✓	✗	AWS
GCP Workflows [96]	✗	Coarse	✗	✗	✓	✓	✓	✗	Google
Azure Logic Apps [32]	✗	Coarse	✗	✗	✓	✓	✓	✗	Azure
Serverless Multicloud [227]	Latency Cost	Fine	✗	✗	✓	✗	✗	✗	AWS, Google, Alibaba
BPMN4FO [224]	✗	Coarse	✗	✗	✗	✓	✗	✗	AWS, Azure, IBM
xAFL [187]	Latency Cost	Fine	✗	✓	✓	✓	✗	✗	AWS, Azure, IBM, Google, Alibaba
OpenTOSCA [216]	✗	Coarse	✗	✗	✓	✓	✓	✗	AWS, Azure, IBM, Google,...
Carbon Aware GSLB [144]	Carbon 🌿	Coarse	✗	✓	✗	✗	✗	✗	Azure
GreenCourier [48]	Carbon 🌿	Fine	✗	✓	✗	✗	✗	✗	Google
CARIBOU	Carbon 🌿 Latency Cost	Fine	✓	✓	✓	✓	✓	✓	AWS

Table 8.1: Overview of different capabilities of frameworks for serverless cloud workflow deployment.

8.2 Sky Computing

Sky Computing [115, 200], where compute resources are seen as decoupled from specific providers and become a utility with a homogenous service layer, proposed as a middleware system [173, 220] that connects workloads with providers, offered inspirations for this work. By abstracting the specific service offerings and being able to send work where it is cheapest [153] or most carbon neutral [56], Sky Computing will lead the way to a more customer-driven pricing and carbon strategy of the providers. The concept has already been evaluated for serverless as well [33], which is, as we have shown, a great candidate for these workloads. We hope to have contributed to this discussion by building a framework that shows that the basic idea of not being tied to cloud providers' limitations is worthwhile and that could be easily extended or integrated into a more Sky Computing-like architecture.

8.3 Provider Perspective

While solutions such as the one proposed by this thesis are limited to the information accessible from the outside, frameworks that could access insider information into the provider’s scheduling and resource allocation could further improve carbon gains. While the unsolved question of communicating constraints and tolerances currently prohibits a full cross-regional implementation, intra-regional workload shifting has already shown promise. Works that have gone intra-regional scheduling and allocation have shown additional gains [48]. Additionally, work approaching the problem from a provisioning angle rather than a deployment side shows further promises [49]. More specifically, colocating centers with greener energies and scheduling virtual machines according to renewable energy production, showcasing the adaptability to changes in power grids, to exploit a ”free lunch” [6] or to load-balance with also taking carbon emissions from energy into account [133] shows that similar approaches can be made even more efficient when having access to provider internal data.

8.4 Edge Computing

There are additional optimization avenues to reduce carbon emissions when considering geospatial deployments. Much computing is happening on edge devices [50, 190], especially with the advance of federated learning [203, 215]. Expanding and accounting for carbon emissions when pushing computation to the edge [110, 117, 125] introduces another complex dimension, where embodied carbon [175] through the heterogeneity of involved devices, data locality [46, 131, 183], data privacy and security [225], become additional important factor for deployment determination [118]. Very recent research [126] has specifically combined the question of resource allocation and serverless edge computing with intermediate energy harvesting servers, modeling the energy usage of mobile users as a Stackelberg game and providing further interesting ideas for extending the current work. Another work has explored the deployment of both serverless functions to edge devices as well as the corresponding key-value state in geographically distributed environments, indicating a direction this research could take to tackle the data locality problem [160].

Chapter 9

Conclusion

When we commenced our work on this thesis in November 2023, we started with the provided map overview of different grid carbon intensities. The differences were quite stark and visible. We asked ourselves, "What if we move the execution of cloud workloads to the greenest regions?". Leading with this question, specifically aiming to increase the sustainability of cloud applications, we did our first back-of-the-envelope calculations, which showed that significant carbon reductions were theoretically possible. However, we have to account for more than just the carbon emitted from the execution of cloud applications. Depending on what research we take as the ground truth, data transmissions will also more or less significantly impact the carbon emissions of cloud applications. These calculations and our knowledge of modern cloud applications' complexity, interconnectedness, and complex requirements sparked our interest. While shifting workloads to greener regions was not necessarily new, current approaches still had significant limitations. A comprehensive review of the state of the art revealed that no prior research has attempted to build a comprehensive framework that would move work to the greenest region in a fine-granular, transmission-aware manner while being multi-objective and tolerance-aware. We considered these paramount to offering a comprehensive solution to maximize the number of applications that could benefit from geospatial offloading. This comprehensive solution would enable workflow developers to tell us their requirements on where and how their workflows should be deployed. Thus, we formulated the following research question: "**Can we reduce the carbon emissions of complex cloud workflows by optimizing the geospatial deployment fine-granularly?**". To answer the research question, we built CARIBOU that enabled us to answer the question while providing us with many engineering and research challenges. We specifically focused on serverless workflows as great candidates for cloud workloads. These offer many features that enable us to answer the research question within the given period. By solving these challenges, such as how to model a complex workflow end-to-end or how

a workflow developer could communicate their complex workflow structure with the framework, we presented a comprehensive system that optimizes the geospatial serverless deployment of complex workflows regarding carbon emissions.

The research conducted in this thesis started by solving the fundamental problems of efficiently allowing developers to declare a workflow structure in source code, deploying and migrating workflows cross-regionally, and executing them across regional boundaries. All of this required us to formalize the possible structures we wanted to be able to handle as part of the framework as DAGs. When we solved these questions, we needed a way to generate deployments that, while being objective aware, also considered the developer requirements. To build a solver that can generate such deployments without being blind to the direction of what to solve for, we first needed to model the relevant metrics end-to-end for a given DAG and potential deployment. This process of guiding the solver started with modeling carbon for the workflow stages and transitions. While we expected much research on the carbon emissions of transmitting data, we quickly learned that domain experts still need to solve this question. This research gap led us to a two-scenario solution that showcases a best-case and worst-case scenario. As with the rest of the proposed framework, we do not aim to answer and solve all domain-specific problems. Instead, we provide the scaffolding to run experiments and gather initial results as a proof of concept. While solutions exist for modeling the end-to-end latency or cost of a complex serverless workflow, adding this to our framework to keep the concerns separate proved challenging. The Monte Carlo simulation approach we implemented for end-to-end modeling allows a solver to efficiently generate and select deployments for fine-grained geospatial serverless deployment while remaining tolerance- and objective-aware by providing specific metric values for each deployment. These simulated metrics allow the solver to have a heuristic to solve for, thus guiding the deployment search for our heuristic-biased stochastic sampling algorithm. This component, which generates deployments, is managed by an overhead-aware super-component that tracks the framework state and initiates new deployment-solving runs when reasonable from an overhead perspective.

When evaluating our framework to answer our research question, we showed that we can reduce the carbon emissions of complex cloud workflows. More specifically, we observed that for five specific benchmark workflows, our framework could reduce carbon emissions by handling the different complexities of a single-function workflow to a complex, conditional MapReduce implementation. Furthermore, our framework can handle the complexity fine-granularly while remaining overhead-aware from a framework perspective and not causing significant invocation overheads compared to existing solutions. While our framework has limitations since it does cause overheads, we are aware of these shortcomings. We can mitigate them in future

iterations, where we provide some ideas for future work in the following subsection. Ultimately, our most significant contribution was to build a comprehensive overhead-aware framework that can deploy, manage, migrate, and invoke complex serverless workflows geospatially to reduce the environmental footprint of cloud workloads.

9.1 Future Work

Building the framework also allows us to glance to the future with many more open questions remaining that can be solved within the confines of our framework since we have laid the foundation. The future work remaining can be categorized into two areas:

1. Addressing limitations of the current solution:

- While the workflow model is a comprehensive approach to complex workflows, it currently needs more tools to model for data sinks and sticky external data. Especially with the increasing interest in analytics applications in the context of serverless workflows, data locality, and corresponding performance overheads become evermore important [119, 142]. In a future iteration, the framework’s maintainers should extend the workflow model by adding data nodes that act as representations. The necessary changes are straightforward and can be implemented without disrupting the framework workflow. While optional for answering the fundamental question, this extension can significantly enhance the model’s capabilities. The extension could easily be integrated into the developer API.
- We implemented the framework to answer the research question of this thesis foremost in mind. We wanted to build the framework as a proof-of-concept, where efficiency was considered secondary for now. Even the selected language for the framework, Python, could be more energy-efficient [170, 171]. While we developed the framework with efficiency in mind, removing many of the low-hanging efficiency overheads should be one of the following steps to improve the efficiency component-wise.
- We additionally identified scaling issues with the current Monte Carlo Simulation approach where future work needs to look into caching, required iterations, and utilization of more efficient programming languages.
- Our current implementation does not react to any active violations for developer-defined tolerances. The ad-hoc reaction would be required to provide better guarantees to developers and react to the changing pressures on the cloud infrastructure.

2. **Extending** the framework to answer additional questions:

- We are approaching the problem of the cost of a function invocation conservatively and globally, where we assume all developers exist within the same AWS account and all free-tier of this account is used up. To extend the architecture into an effective middleware architecture, an additional account layer must be introduced where the framework has access to account information and specific privileges; however, the source code and AWS billing remain on the developer or their organization account while the middleware charges separately per developed framework.
- Currently, all roles that interact with the framework, including the framework itself, must be within the same AWS organization or at least have access to the same resources. This model is sufficient for a proof-of-concept. However, to transform the proposed framework into a fully-fledged middleware solution, more work will be required on accounts, usages, and billing.
- While AWS is a good candidate for an initial proof-of-concept, offering the most diverse set of regions globally, extending the framework to consider additional providers might allow for further research into carbon efficiency differences between providers and eventually increase the competitiveness between providers regarding their carbon efficiency.
- Extending the framework to optimize in another dimension of temporal shifting together with geospatial shifting and allowing developers to communicate their tolerance to temporal shifting efficiently might allow for additional research questions and optimizations for carbon reduction.

Bibliography

- [1] ACM Technology Council. Computing and climate change. <https://dl.acm.org/doi/pdf/10.1145/3483410>, 2021. Accessed: 2024-05-17.
- [2] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Aditya Sundarrajan, Kiwan Maeng, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon dependencies in datacenter design and management. *ACM SIGENERGY Energy Informatics Review*, 3(3):21–26, 2023.
- [3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.
- [4] Swedish Energy Agency. Energy in sweden 2022. https://energimyndigheten.a-w2m.se/FolderContents.mvc/Download%3FResourceId%3D208766&ved=2ahUKEwjQ2f_wxaaFAxW40DQIHS1kDooQFnoECDQQAQ&usg=AOvVaw2515MriW3CiEpeLNcti_qC, 2023. Accessed: 2024-05-17.
- [5] Iftikhar Ahmad, Muhammad Imran Khan Khalil, and Syed Adeel Ali Shah. Optimization-based workload distribution in geographically distributed data centers: A survey. *International Journal of Communication Systems*, 33(12):e4453, 2020.
- [6] Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W Moore, and Andy Hopper. Free lunch: Exploiting renewable energy for computing. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, 2011.
- [7] Amazon Web Services. Four trends driving global utility digitization. <https://aws.amazon.com/blogs/industries/four-trends-driving-global-utility-digitization/>, 2020. Accessed: 2024-05-15.

- [8] Amazon Web Services. Renewable energy methodology. <https://sustainability.aboutamazon.com/renewable-energy-methodology.pdf>, 2023. Accessed: 2024-05-08.
- [9] Amazon Web Services. Understanding your AWS carbon footprint estimation. <https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/ccft-estimation.html>, 2023. Accessed: 2024-05-17.
- [10] Amazon Web Services. Amazon cloudwatch. <https://aws.amazon.com/cloudwatch/>, 2024. Accessed: 2024-05-10.
- [11] Amazon Web Services. Amazon DynamoDB pricing. <https://aws.amazon.com/dynamodb/pricing/>, 2024. Accessed: 2024-05-08.
- [12] Amazon Web Services. Amazon EC2 on-demand pricing. <https://aws.amazon.com/ec2/pricing/on-demand/>, 2024. Accessed: 2024-04-17.
- [13] Amazon Web Services. Amazon elastic container registry pricing. <https://aws.amazon.com/ecr/pricing/>, 2024. Accessed: 2024-05-06.
- [14] Amazon Web Services. Amazon SNS pricing. <https://aws.amazon.com/sns/pricing/>, 2024. Accessed: 2024-05-08.
- [15] Amazon Web Services. Amazon SNS publish. https://docs.aws.amazon.com/sns/latest/api/API_Publish.html#:~:text=Constraints%3A,contain%20up%20to%20140%20characters., 2024. Accessed: 2024-05-14.
- [16] Amazon Web Services. AWS DynamoDB. <https://aws.amazon.com/dynamodb/>, 2024. Accessed: 2024-05-08.
- [17] Amazon Web Services. AWS free tier. <https://aws.amazon.com/free/>, 2024. Accessed: 2024-05-08.
- [18] Amazon Web Services. AWS Lambda pricing. <https://aws.amazon.com/lambda/pricing/>, 2024. Accessed: 2024-05-17.
- [19] Amazon Web Services. AWS S3. <https://aws.amazon.com/s3/>, 2024. Accessed: 2024-05-08.
- [20] Amazon Web Services. AWS step functions. <https://aws.amazon.com/step-functions/>, 2024. Accessed: 2024-05-17.
- [21] Amazon Web Services. Aws step functions pricing. <https://aws.amazon.com/step-functions/pricing/>, 2024. Accessed: 2024-05-06.

- [22] Amazon Web Services. Boto3. <https://github.com/boto/boto3>, 2024. Accessed: 2024-04-17.
- [23] Amazon Web Services. The cloud - amazon sustainability. <https://sustainability.aboutamazon.com/products-services/the-cloud>, 2024. Accessed: 2024-05-08.
- [24] Amazon Web Services. Configure lambda function memory. <https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html>, 2024. Accessed: 2024-05-15.
- [25] Amazon Web Services. Policies and permissions in iam. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html, 2024. Accessed: 2024-05-10.
- [26] Amazon Web Services. Read consistency - amazon dynamodb. <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html>, 2024. Accessed: 2024-05-10.
- [27] Amazon Web Services. Service, account and table quotas in Amazon DynamoDB. [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ServiceQuotas.html#:~:text=The%20maximum%20item%20size%20in,lengths%20\(again%20binary%20length\).](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ServiceQuotas.html#:~:text=The%20maximum%20item%20size%20in,lengths%20(again%20binary%20length).), 2024. Accessed: 2024-05-14.
- [28] Amazon Web Services. What is AWS price list? <https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/price-changes.html>, 2024. Accessed: 2024-05-16.
- [29] IPCC Annex III. Technology-specific cost and performance parameters. *IPCC. Climate Change*, 2014.
- [30] Joshua Aslan, Kieren Mayers, Jonathan G Koomey, and Chris France. Electricity intensity of internet data transmission: Untangling the estimates. *Journal of industrial ecology*, 22(4):785–798, 2018.
- [31] Azure. Azure Functions invocation trace 2021. <https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md>, 2021. Accessed: 2024-05-17.
- [32] Azure. Azure logic apps. <https://learn.microsoft.com/en-us/azure/logic-apps/>, 2024. Accessed: 2024-05-17.
- [33] Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahradd. On merits and viability of multi-cloud serverless. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 600–608. ACM, 2021.

- [34] Jayant Baliga, Robert WA Ayre, Kerry Hinton, and Rodney S Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99(1):149–167, 2010.
- [35] Liang Bao, Chase Wu, Xiaoxuan Bu, Nana Ren, and Mengqing Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2114–2129, 2019.
- [36] Jonathan Barnsley, Jhénelle A Williams, Simon Chin-Yee, Anthony Costello, Mark Maslin, Jacqueline McGlade, Richard Taylor, Matthew Winning, and Priti Parikh. Location location location: a carbon footprint calculator for transparent travel to the UN climate conference 2022. *UCL Open Environment*, 5, 2023.
- [37] Luis Andre Barroso and Jimmy Clidaras. *The datacenter as a computer: An introduction to the design of warehouse-scale machines*. Springer Nature, 2022.
- [38] Sanjoy Baruah and Alberto Marchetti-Spaccamela. The computational complexity of feasibility analysis for conditional DAG tasks. *ACM Transactions on Parallel Computing*, 10(3), 9 2023.
- [39] Noman Bashir, Tian Guo, Mohammad Hajiesmaili, David Irwin, Prashant Shenoy, Ramesh Sitaraman, Abel Souza, and Adam Wierman. Enabling sustainable clouds: The case for virtualizing the energy system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 350–358, 2021.
- [40] Lotfi Belkhir and Ahmed Elmeligi. Assessing ICT global emissions footprint: Trends to 2040 & recommendations. *Journal of cleaner production*, 177:448–463, 2018.
- [41] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefer. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.
- [42] ben174. Profanity. <https://github.com/ben174/profanity>. Accessed: 2024-04-19.
- [43] Anders Bjørn, Shannon M Lloyd, Matthew Brander, and H Damon Matthews. Renewable energy certificates threaten the integrity of corporate science-based targets. *Nature Climate Change*, 12(6):539–546, 2022.

- [44] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Automatically measuring the maintainability of service-and microservice-based systems: a literature review. In *Proceedings of the 27th international workshop on software measurement and 12th international conference on software process and product measurement*, pages 107–115, 2017.
- [45] Matthew Brander, Michael Gillenwater, and Francisco Ascui. Creative accounting: A critical perspective on the market-based method for reporting purchased electricity (scope 2) emissions. *Energy Policy*, 112:29–33, 2018.
- [46] Martin Breitbach, Dominik Schäfer, Janick Edinger, and Christian Becker. Context-aware data and task placement in edge computing environments. In *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–10. IEEE, 2019.
- [47] John L Bresina. Heuristic-biased stochastic sampling. In *Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1*, pages 271–278, 1996.
- [48] Mohak Chadha, Thandayuthapani Subramanian, Eishi Arima, Michael Gerndt, Martin Schulz, and Osama Abboud. Greencourier: Carbon-aware scheduling for serverless functions. In *Proceedings of the 9th International Workshop on Serverless Computing*, pages 18–23, 2023.
- [49] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008.
- [50] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.
- [51] Xinghan Chen, Ling-Hong Hung, Robert Cordingly, and Wes Lloyd. X86 vs. arm64: An investigation of factors influencing serverless performance. In *Proceedings of the 9th International Workshop on Serverless Computing*, pages 7–12, 2023.
- [52] Andrew A. Chien, Richard Wolski, and Fan Yang. The zero-carbon cloud: High-value, dispatchable demand for renewable power generators. *The Electricity Journal*, 28(8):110–118, 2015.
- [53] A. Jeffrey Clark and contributors. Python Imaging Library (Fork). <https://github.com/python-pillow/Pillow/>, 2022. Accessed: 2024-04-17.

- [54] CloudPing. AWS latency monitoring. <https://www.cloudping.co/grid>, 2024. Accessed: 2024-04-04.
- [55] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefer. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.
- [56] Robert Cordingly, Jasleen Kaur, Divyansh Dwivedi, and Wes Lloyd. Towards serverless sky computing: An investigation on global workload distribution to mitigate carbon intensity, network latency, and cost. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*, pages 59–69. IEEE, 2023.
- [57] Robert Cordingly, Wen Shu, and Wes J Lloyd. Predicting performance and cost of serverless computing functions with saaf. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCoM/CyberSciTech)*, pages 640–649. IEEE, 2020.
- [58] Hallie Cramer and Savannah Goodman. Accelerating a carbon-free future with hourly energy tracking. <https://cloud.google.com/blog/topics/sustainability/t-eacs-show-promise-for-helping-decarbonize-the-grid>, 2023. Accessed: 2024-05-08.
- [59] Timothy Crosley. isort. <https://pycqa.github.io/isort/>, 2024. Accessed: 2024-04-17.
- [60] John D. Wilson and Zach Zimmerman. The era of flat power demand is over. <https://gridstrategiesllc.com/wp-content/uploads/2023/12/National-Load-Growth-Report-2023.pdf>, 2023. Accessed: 2024-04-17.
- [61] Mehdiar Dabbagh, Bechir Hamdaoui, Ammar Rayes, and Mohsen Guizani. Shaving data center power demand peaks through energy storage and workload shifting control. *IEEE Transactions on Cloud Computing*, 7(4):1095–1108, 2017.
- [62] Datadog. The state of serverless 2023. <https://www.datadoghq.com/state-of-serverless/>, 2023. Accessed: 2024-05-14.
- [63] Datadog. The state of serverless 2023. <https://www.datadoghq.com/state-of-serverless/>, 2023. Accessed: 2024-05-17.

- [64] Olivier Daxhelet and Yves Smeers. The eu regulation on cross-border trade of electricity: A two-stage equilibrium model. *European Journal of Operational Research*, 181(3):1396–1412, 2007.
- [65] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. Data center energy consumption modeling: A survey. *IEEE Communications surveys & tutorials*, 18(1):732–794, 2015.
- [66] Secretaría de estado de Energía. Balance energético de españa. https://www.miteco.gob.es/content/dam/mitesco/es/energia/files-1/balances/Balances/Documents/balance-20231218/Balance%20Energetico%20Espaa%202021%20y%202022_v0.pdf, 2023. Accessed: 2024-05-17.
- [67] Ministère de la Transition Écologique et de la Cohésion des Territoires. Bilan énergétique de la france en 2022. <https://www.statistiques.developpement-durable.gouv.fr/media/7190/download?inline>, 2 2024. Accessed: 2024-05-17.
- [68] Federal Statistical Office (Destatis). Electricity production in 2022: coal accounted for a third, wind power for a quarter. https://www.destatis.de/EN/Press/2023/03/PE23_090_43312.html, 3 2024. Accessed: 2024-05-17.
- [69] Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, Nathaniel Morris, and Christopher Stewart. Characterizing service level objectives for cloud services: Realities and myths. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 200–206. IEEE, 2019.
- [70] Edinburgh Genome Foundry. Dna features viewer. <https://edinburgh-genome-foundry.github.io/DnaFeaturesViewer/>, 2024. Accessed: 2024-04-26.
- [71] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC international conference on performance engineering*, pages 265–276, 2020.
- [72] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering*, 48(10):4152–4166, 2021.
- [73] Electricity Maps. Reduce carbon emissions with actionable electricity data. <https://www.electricitymaps.com/>, 2024. Accessed: 2024-05-17.

- [74] European Parliament and Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council. <https://data.europa.eu/eli/reg/2016/679/oj>. Accessed: 2024-05-17.
- [75] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *ACM SIGARCH computer architecture news*, 35(2):13–23, 2007.
- [76] Nick Ferris. Could too much permitting reform hurt EU renewables? <https://www.energymonitor.ai/sectors/power/could-too-much-permitting-reform-hurt-eu-renewables/>, 2023. Accessed: 2024-05-08.
- [77] Nicolas Ferry, Hui Song, Alessandro Rossini, Franck Chauvel, and Arnor Solberg. Cloudmf: applying mde to tame the complexity of managing multi-cloud applications. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 269–277. IEEE, 2014.
- [78] Marion Ficher, Françoise Berthoud, Anne-Laure Ligozat, Patrick Sigonneau, Maxime Wisslé, and Badis Tebbani. Assessing the carbon footprint of the data transmission on a backbone network. In *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 105–109. IEEE, 2021.
- [79] José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luís Miguel Pinho. A multi-dag model for real-time parallel applications with conditional execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1925–1932, 2015.
- [80] Department for Energy Security and Net Zero. Digest of uk energy statistics annual data for uk, 2022. https://assets.publishing.service.gov.uk/media/64f1fcba9ee0f2000db7bdd8/DUKES_2023_Chapters_1-7.pdf, 7 2023. Accessed: 2024-05-17.
- [81] Bundesamt für Statistik (BFS). Energie: Panorama. <https://dam-api.bfs.admin.ch/hub/api/dam/assets/30489027/master>, Jan 2024. Accessed: 2024-05-17.
- [82] Jiechao Gao, Haoyu Wang, and Haiying Shen. Smartly handling renewable energy instability in supporting a cloud datacenter. In *2020 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 769–778. IEEE, 2020.
- [83] German Criminal Code. Section 86 dissemination of propaganda material of unconstitutional and terrorist organisations. https://www.gesetze-im-internet.de/englisch_stgb/englisch_stgb.html#p0922. Accessed: 2024-04-17.

- [84] Mahdi Ghamkhari and Hamed Mohsenian-Rad. Optimal integration of renewable energy resources in data centers with behind-the-meter renewable generator. In *2012 IEEE International Conference on Communications (ICC)*, pages 3340–3344. IEEE, 2012.
- [85] Vicent Giménez-Alventosa, Germán Moltó, and Miguel Caballer. A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Generation Computer Systems*, 97:259–274, 2019.
- [86] GitHub. GitHub Copilot. <https://github.com/features/copilot>. Accessed: 2024-05-17.
- [87] Wedan Emmanuel Gnibga, Anne Blavette, and Anne-Cécile Orgerie. Renewable energy in data centers: the dilemma of electrical grid dependency and autonomy costs. *IEEE Transactions on Sustainable Computing*, 2023.
- [88] Muhammed Golec, Sukhpal Singh Gill, Felix Cuadrado, Ajith Kumar Parlikad, Minxian Xu, Huaming Wu, and Steve Uhlig. Atom: Ai-powered sustainable resource management for serverless edge computing environments. *IEEE Transactions on Sustainable Computing*, 2023.
- [89] Google. Text-to-speech. <https://cloud.google.com/text-to-speech?hl=en>. Accessed: 2024-04-19.
- [90] Google. Carbon-aware computing by location. <https://blog.google/outreach-initiatives/sustainability/carbon-aware-computing-location/>, 2023. Accessed: 2024-05-17.
- [91] Google. A smarter way to buy clean energy. <https://cloud.google.com/blog/topics/sustainability/a-smarter-way-to-buy-clean-energy>, 2023. Accessed: 2024-05-08.
- [92] Google. Aiming to achieve net-zero emissions. <https://sustainability.google/operating-sustainably/net-zero-carbon/>, 2024. Accessed: 2024-05-08.
- [93] Google. Carbon free energy for google cloud regions. <https://cloud.google.com/sustainability/region-carbon>, 2024. Accessed: 2024-05-08.
- [94] Google Cloud. COP28: How to decarbonize your Google Cloud carbon footprint. <https://cloud.google.com/blog/topics/sustainability/>

- cop28-how-to-decarbonize-your-google-cloud-carbon-footprint, 2023. Accessed: 2024-05-17.
- [95] Google Cloud. Google Functions pricing. <https://cloud.google.com/functions/pricing>, 2024. Accessed: 2024-05-08.
 - [96] Google Cloud Platform. Gcp workflows. <https://cloud.google.com/workflows/>, 2024. Accessed: 2024-05-17.
 - [97] Mia E Gortney, Patrick E Harris, Tomas Cerny, Abdullah Al Maruf, Miroslav Bures, Davide Taibi, and Pavel Tisnovsky. Visualizing microservice architecture in the dynamic perspective: A systematic mapping study. *IEEE Access*, 10:119999–120012, 2022.
 - [98] Government of Canada. Personal information protection and electronic documents act. <https://laws-lois.justice.gc.ca/eng/acts/p-8.6/FullText.html>, Apr 2000. Accessed: 2024-05-13.
 - [99] Grammarly, Inc. Grammarly. <https://www.grammarly.com/features>. Accessed: 2024-05-17.
 - [100] Alex Grasas, Angel A Juan, Javier Faulin, J sica De Armas, and Helena Ramalhinho. Biased randomization of heuristics using skewed probability distributions: A survey and some applications. *Computers & Industrial Engineering*, 110:216–228, 2017.
 - [101] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back, 2018.
 - [102] Camilla Hodgson. Booming AI demand threatens global electricity supply. <https://www.ft.com/content/b7570359-f809-49ce-8cd5-9166d36a057b>, 2024. Accessed: 2024-04-17.
 - [103] Forrest Iandola. Squeezenet v1.1. https://github.com/forresti/SqueezeNet/tree/master/SqueezeNet_v1.1, 2016. Accessed: 2024-04-17.
 - [104] IBM. Carbon footprint calculations. <https://www.ibm.com/docs/en/tririga/10.7?topic=impact-carbon-footprint-calculations>, 2023. Accessed: 2024-05-17.
 - [105] IEA Secretariat. Electricity grids and secure energy transitions. Technical report, International Energy Agency, Paris, 2023.
 - [106] IEA Secretariat. Electricity 2024 analysis and forecast to 2026. Technical report, International Energy Agency, Paris, 2024.

- [107] INO. Video analytics dataset. <https://www.ino.ca/en/technologies/video-analytics-dataset/>, 2023. Accessed: 2024-04-17.
- [108] David Irwin and Prashant Shenoy. Understanding embodied emissions. *ACM SIGENERGY Energy Informatics Review*, 3(3):1–3, 2023.
- [109] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G Patil, Joseph E Gonzalez, and Ion Stoica. Skyplane: Optimizing transfer cost and throughput using cloud-aware overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1375–1389, 2023.
- [110] Congfeng Jiang, Tiantian Fan, Honghao Gao, Weisong Shi, Liangkai Liu, Christophe C  rin, and Jian Wan. Energy aware edge computing: A survey. *Computer Communications*, 151:556–580, 2020.
- [111] Qingye Jiang, Young Choon Lee, and Albert Y Zomaya. Serverless execution of scientific workflows. In *International Conference on Service-Oriented Computing*, pages 706–721. Springer, 2017.
- [112] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC ’23, page 443–458. ACM, 2023.
- [113] Prajakta S Kalekar et al. Time series forecasting using holt-winters exponential smoothing. *Kanwal Rekhi school of information Technology*, 4329008(13):1–13, 2004.
- [114] Alexey Karyakin and Kenneth Salem. An analysis of memory power consumption in database systems. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, pages 1–9, 2017.
- [115] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky computing. *IEEE Internet Computing*, 13(5):43–51, 2009.
- [116] Jeongchul Kim and Kyungyong Lee. FunctionBench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.
- [117] Young Geun Kim, Udit Gupta, Andrew McCrabb, Yonglak Son, Valeria Bertacco, David Brooks, and Carole-Jean Wu. GreenScale: Carbon-aware systems for edge computing. *arXiv preprint arXiv:2304.00404*, 2023.

- [118] Dragi Kimovski, Roland Mathá, Josef Hammer, Narges Mehran, Hermann Hellwagner, and Radu Prodan. Cloud, fog, or edge: Where to compute? *IEEE Internet Computing*, 25(4):30–36, 2021.
- [119] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.
- [120] Iwona Kotlarska, Andrzej Jackowski, Krzysztof Lichota, Michal Welnicki, Cezary Dubnicki, and Konrad Iwanicki. InftyDedup: Scalable and cost-effective cloud tiering with deduplication. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 33–48, 2023.
- [121] Nane Kratzke. A brief history of cloud application architectures. *Applied Sciences*, 8(8):1368, 2018.
- [122] Saurabh Kumar and Rajkumar Buyya. Green cloud computing and environmental sustainability. *Harnessing green IT: principles and practices*, pages 315–339, 2012.
- [123] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.
- [124] Loïc Lannelongue, Jason Grealey, and Michael Inouye. Green algorithms: quantifying the carbon footprint of computation. *Advanced science*, 8(12):2100707, 2021.
- [125] Wei Li, Ting Yang, Flavia C Delicato, Paulo F Pires, Zahir Tari, Samee U Khan, and Albert Y Zomaya. On enabling sustainable edge computing with renewable energy resources. *IEEE communications magazine*, 56(5):94–101, 2018.
- [126] Yunqi Li and Changlin Yang. Resource allocation and pricing in energy harvesting serverless computing internet of things networks. *Information*, 15(5):250, 2024.
- [127] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th acm international conference on architectural support for programming languages and operating systems*, pages 782–796, 2022.
- [128] Changyuan Lin, Nima Mahmoudi, Caixiang Fan, and Hamzeh Khazaei. Fine-grained performance and cost modeling and optimization

- for faas applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):180–194, 2022.
- [129] Changyuan Lin, Nima Mahmoudi, Caixiang Fan, and Hamzeh Khazaei. Fine-grained performance and cost modeling and optimization for FaaS applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):180–194, 2023.
 - [130] P Lipton, C Lauwers, and D Tamburri. Oasis topology and orchestration specification for cloud applications (tosca) tc. *March2017*, 2017.
 - [131] Ying Liu, Qiang He, Dequan Zheng, Xiaoyu Xia, Feifei Chen, and Bin Zhang. Data caching optimization in the edge computing environment. *IEEE Transactions on Services Computing*, 15(4):2074–2085, 2020.
 - [132] Zhenhua Liu, Minghong Lin, Adam Wierman, Steven H Low, and Lachlan LH Andrew. Geographical load balancing with renewables. *ACM SIGMETRICS Performance Evaluation Review*, 39(3):62–66, 2011.
 - [133] Zhenhua Liu, Minghong Lin, Adam Wierman, Steven H Low, and Lachlan LH Andrew. Greening geographical load balancing. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):193–204, 2011.
 - [134] Zhenhua Liu, Adam Wierman, Yuan Chen, Benjamin Razon, and Ni-angjun Chen. Data center demand response: Avoiding the coincident peak via workload shifting and local generation. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, pages 341–342, 2013.
 - [135] Logilab. Pylint. https://pylint.pycqa.org/en/latest/user_guide/usage/run.html, 2024. Accessed: 2024-04-17.
 - [136] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow: trigger-based orchestration of serverless workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS ’20, page 3–14. ACM, 2020.
 - [137] Pauline Loygue, Khaldoun Al Agha, and Guy Pujolle. Carbon footprint of cloud, edge and internet of edges. *Green Communications*, 2023.
 - [138] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.

- [139] Jianying Luo, Lei Rao, and Xue Liu. Temporal load balancing with service delay guarantees for data center energy cost optimization. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):775–784, 2013.
- [140] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.
- [141] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022.
- [142] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301, 2021.
- [143] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. WISEFUSE: Workload characterization and DAG transformation for serverless workflows. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), jun 2022.
- [144] Diptyaroop Maji, Ben Pfaff, Vipin PR, Rajagopal Sreenivasan, Victor Firoiu, Sreeram Iyer, Colleen Josephson, Zhelong Pan, and Ramesh K Sitaraman. Bringing carbon awareness to multi-cloud application delivery. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, pages 1–6, 2023.
- [145] Marc X Makkes, Arie Taal, Anwar Osseyran, and Paola Grosso. A decision framework for placement of applications in clouds that minimizes their carbon footprint. *Journal of Cloud Computing: Advances, Systems and Applications*, 2:1–13, 2013.
- [146] Jens Malmmodin. Science & society forum: Växande ikt-sektor och fler datacenter – hur påverkas elförsörjningen? IVA-conference Science & Society Forum, 2020.
- [147] Jens Malmmodin, Nina Lövehagen, Pernilla Bergmark, and Dag Lundén. ICT sector electricity consumption and greenhouse gas emissions–2020 outcome. *Telecommunications Policy*, page 102701, 2024.

- [148] Matt Adorjan. AWS latency monitoring. <https://www.cloudping.co/grid>, 2024. Accessed: 2024-05-16.
- [149] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 211–221. IEEE, 2015.
- [150] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.
- [151] Michael Sauter. Crane - lift containers with ease. <https://michaelsauter.github.io/crane/index.html>, 2020. Accessed: 2024-04-17.
- [152] Microsoft. Emissions impact dashboard. <https://www.microsoft.com/en-ca/sustainability/emissions-impact-dashboard>, 2023. Accessed: 2024-05-17.
- [153] André Monteiro, Cláudio Teixeira, and Joaquim Sousa Pinto. Sky computing: exploring the aggregated cloud resources. *Cluster Computing*, 20(1):621–631, 2017.
- [154] Attracta Mooney. Gridlock: how a lack of power lines will delay the age of renewables. <https://www.ft.com/content/a3be0c1a-15df-4970-810a-8b958608ca0f>, 2023. Accessed: 2024-05-08.
- [155] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.
- [156] Janine Morley, Kelly Widdicks, and Mike Hazas. Digitalisation, energy and data demand: The impact of internet traffic on overall and peak electricity consumption. *Energy Research & Social Science*, 38:128–137, 2018.
- [157] myclimate. Co2 emissions calculator for your car. https://co2.myclimate.org/en/car_calculators/new, 2024. Accessed: 2024-05-11.
- [158] mypy project. mypy. <https://mypy-lang.org>, 2024. Accessed: 2024-04-17.

- [159] David Mytton and Masaō Ashtine. Sources of data center energy estimates: A comprehensive review. *Joule*, 6(9):2032–2056, 2022.
- [160] Matteo Nardelli and Gabriele Russo Russo. Function offloading and data migration for stateful serverless edge computing. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, page 247–257, New York, NY, USA, 2024. Association for Computing Machinery.
- [161] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. Response-time analysis of limited-preemptive parallel dag tasks under global scheduling. In *31st Conference on Real-Time Systems*, pages 21–1, 2019.
- [162] National Library of Medicine. Genbank. <https://www.ncbi.nlm.nih.gov/genbank/>, 2022. Accessed: 2024-04-17.
- [163] Thong Trung Nguyen, Thu Anh Thi Pham, and Huong Thi Xuan Tram. Role of information and communication technologies and innovation in driving carbon emissions and economic growth in selected g-20 countries. *Journal of environmental management*, 261:110162, 2020.
- [164] NordPool and Granular Energy. About time: How incorporating timestamped energy certificates into electricity markets could accelerate the energy transition. Technical report, NordPool, Oslo, 2023.
- [165] Sustainable Energy Authority of Ireland. Energy in ireland 2022. <https://www.seai.ie/publications/Energy-in-Ireland-2022.pdf>, 2023. Accessed: 2024-05-17.
- [166] Office of the Federal Register, National Archives and Records Administration. Public law 104 - 191 - health insurance portability and accountability act of 1996. <https://www.govinfo.gov/app/details/PLAW-104publ191>, Aug 1996. Accessed: 2024-05-13.
- [167] Open Data Commons. Open data commons open database license. <https://opendatacommons.org/licenses/odbl/>. Accessed: 2024-05-17.
- [168] OpenAI. Introducing chatgpt. <https://openai.com/blog/chatgpt>. Accessed: 2024-05-17.
- [169] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009.
- [170] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency

- across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, pages 256–267, 2017.
- [171] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.
 - [172] Andres P Perez, Enzo E Sauma, Francisco D Munoz, and Benjamin F Hobbs. The economic effects of interregional trading of renewable energy certificates in the us wecc. *The Energy Journal*, 37(4):267–296, 2016.
 - [173] Dana Petcu, Ciprian Crăciun, Marian Neagul, Silviu Panica, Beniamino Di Martino, Salvatore Venticinque, Massimiliano Rak, and Rocco Aversa. Architecturing a sky computing platform. In *Towards a Service-Based Internet. ServiceWave 2010 Workshops: International Workshops, OCS, EMSOA, SMART, and EDBPM 2010, Ghent, Belgium, December 13-15, 2010, Revised Selected Papers 3*, pages 1–13. Springer, 2011.
 - [174] Pin Hong Long. Daniel’s art gallery. <https://www.daniellong.ca/gallery/>, 2020. Accessed: 2024-04-17.
 - [175] Thibault Pirson and David Bol. Assessing the embodied carbon footprint of iot edge devices with a bottom-up life-cycle approach. *Journal of Cleaner Production*, 322:128966, 2021.
 - [176] Brad Plumer and Nadja Popovich. Giant batteries are transforming the way the U.S. uses electricity. <https://www.nytimes.com/interactive/2024/05/07/climate/battery-electricity-solar-california-texas.html>, 2024. Accessed: 2024-05-07.
 - [177] Brad Plumer and Nadja Popovich. A new surge in power use is threatening U.S. climate goals. <https://www.nytimes.com/interactive/2024/03/13/climate/electric-power-climate-change.html>, 2024. Accessed: 2024-04-17.
 - [178] Lorenzo Posani, Alessio Paccioia, and Marco Moschettini. The carbon footprint of distributed cloud storage. *arXiv preprint arXiv:1803.06973*, 2018.
 - [179] FFmpeg project. FFmpeg. <https://ffmpeg.org>, 2024. Accessed: 2024-04-17.

- [180] PSF. Black: The the uncompromising python code formatter. <https://github.com/psf/black>, 2024. Accessed: 2024-04-17.
- [181] PyTorch Foundation. torchvision. <https://pytorch.org/vision/stable/index.html>, 2024. Accessed: 2024-04-17.
- [182] Ana Radovanović, Ross Koningstein, Ian Schneider, Bokan Chen, Alexandre Duarte, Binz Roy, Diyue Xiao, Maya Haridasan, Patrick Hung, Nick Care, et al. Carbon-aware computing for datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, 2022.
- [183] Brian Ramprasad, Alexandre da Silva Veith, Moshe Gabel, and Eyal de Lara. Sustainable computing on the edge: A system dynamics perspective. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 64–70, 2021.
- [184] Joseph Rand, Rose Strauss, Will Gorman, Joachim Seel, Julie Mulvaney Kemp, Seongeun Jeong, Dana Robson, and Wiser Ryan. Queued up: Characteristics of power plants seeking transmission interconnection as of the end of 2022. https://emp.lbl.gov/sites/default/files/emp-files/queued_up_2022_04-06-2023.pdf, 2022. Accessed: 2024-04-17.
- [185] Paul Reinheimer and Will Roberts. WonderNetwork. <https://wondernetwork.com>, 2024. Accessed: 2024-05-04.
- [186] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. AFCL: An abstract function choreography language for serverless workflow specification. *Future Generation Computer Systems*, 114:368–382, 2021.
- [187] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. xAFCL: Run scalable function choreographies across multiple FaaS systems. *IEEE Transactions on Services Computing*, 2021.
- [188] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’21, page 1–17. ACM, 2021.
- [189] Nicole A Ryan, Jeremiah X Johnson, and Gregory A Keoleian. Comparative assessment of models and methods to calculate grid electricity emissions. *Environmental science & technology*, 50(17):8937–8953, 2016.
- [190] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

- [191] Trever Schirmer, Nils Japke, Sofia Greten, Tobias Pfandzelter, and David Bermbach. The night shift: Understanding performance variability of cloud serverless platforms. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*, pages 27–33, 2023.
- [192] Lars Schor, Andreas Tretter, Tobias Scherer, and Lothar Thiele. Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of opencl. In *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 41–50. IEEE, 2013.
- [193] Skipper Seabold and Josef Perktold. Statsmodels: econometric and statistical modeling with Python. *SciPy*, 7:1, 2010.
- [194] Anders SG Andrae. New perspectives on internet electricity use in 2030. *Engineering and Applied Science Letter*, 3(2):19–31, 2020.
- [195] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: a survey of opportunities, challenges, and applications. *ACM Computing Surveys*, 54(11s):1–32, 2022.
- [196] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.
- [197] Jörgen Sjödin and Stefan Grönkvist. Emissions accounting for use and supply of electricity in the nordic market. *Energy policy*, 32(13):1555–1564, 2004.
- [198] Jamie Smyth and Eva Xiao. Resurgent US electricity demand sparks power grid warnings. <https://www.ft.com/content/9892d043-64bd-42f4-a36c-0f6717a255da>, 2024. Accessed: 2024-05-08.
- [199] Terna S.p.A. Produzione di energia elettrica in italia. https://download.terna.it/terna/5%20-%20PRODUZIONE_8db99b7e93be883.pdf, 2023. Accessed: 2024-05-17.
- [200] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 26–32, 2021.
- [201] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

- [202] Jaspar Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, page 62–71, New York, NY, USA, 1996. Association for Computing Machinery.
- [203] Yuchang Sun, Jiawei Shao, Yuyi Mao, Jessie Hui Wang, and Jun Zhang. Semi-decentralized federated edge learning with data and device heterogeneity. *IEEE Transactions on Network and Service Management*, 2023.
- [204] Rob Sutter. Go serverless! stop worrying about infrastructure and ship more. GoDays Berlin, 2020.
- [205] Swiss Financial Market Supervisory Authority. Circular 2018/3: Outsourcing – banks and insurers. 2018.
- [206] The Guardian. Power grab: the hidden costs of ireland’s data-centre boom. <https://www.theguardian.com/world/2024/feb/15/power-grab-hidden-costs-of-ireland-datacentre-boom>, 2024. Accessed: 2024-04-22.
- [207] Thoughtworks. Cloud carbon footprint methodology. <https://www.cloudcarbonfootprint.org/docs/methodology/>, 2023. Accessed: 2024-05-17.
- [208] Adel Nadjaran Toosi, Chenhao Qu, Marcos Dias de Assunção, and Rajkumar Buyya. Renewable-aware geographical load balancing of web applications for sustainable data centers. *Journal of Network and Computer Applications*, 83:155–168, 2017.
- [209] Bo Tranberg, Olivier Corradi, Bruno Lajoie, Thomas Gibon, Iain Staffell, and Gorm Bruun Andresen. Real-time carbon accounting method for the european electricity markets. *Energy Strategy Reviews*, 26:100367, 2019.
- [210] United States Environmental Protection Agency. Renewable energy certificates (recs). <https://www.epa.gov/green-power-markets/renewable-energy-certificates-recs>, 2024. Accessed: 2024-05-14.
- [211] vHive Ecosystem. vSwarm: A suite of representative serverless cloud-agnostic (i.e., dockerized) benchmarks. <https://github.com/vhive-serverless/vSwarm>, 2024. Accessed: 2024-04-17.
- [212] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms.

- In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.
- [213] Shangguang Wang, Ao Zhou, Ching-Hsien Hsu, Xuanyu Xiao, and Fangchun Yang. Provision of data-intensive services through energy- and qos-aware virtual machine placement in national cloud data centers. *IEEE Transactions on Emerging Topics in Computing*, 4(2):290–300, 2015.
 - [214] Philipp Wiesner, Ilja Behnke, Dominik Scheinert, Kordian Gontarska, and Lauritz Thamsen. Let’s wait awhile: How temporal workload shifting can reduce carbon emissions in the cloud. In *Proceedings of the 22nd International Middleware Conference*, pages 260–272, 2021.
 - [215] Jiajun Wu, Steve Drew, Fan Dong, Zhuangdi Zhu, and Jiayu Zhou. Topology-aware federated learning in edge computing: A comprehensive survey. *arXiv preprint arXiv:2302.02573*, 2023.
 - [216] Michael Wurster, Uwe Breitenbücher, Kálmán Képes, Frank Leymann, and Vladimir Yussupov. Modeling and automated deployment of serverless applications using TOSCA. In *2018 IEEE 11th conference on service-oriented computing and applications (SOCA)*, pages 73–80. IEEE, 2018.
 - [217] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. Exploring power-performance tradeoffs in database systems. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 485–496. IEEE, 2010.
 - [218] Fan Yang and Andrew A Chien. Zccloud: Exploring wasted green power for high-performance computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1051–1060. IEEE, 2016.
 - [219] Fan Yang and Andrew A Chien. Large-scale and extreme-scale computing with stranded green power: Opportunities and costs. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):1103–1116, 2017.
 - [220] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, 2023.
 - [221] Yelp Inc. Yelp dataset. <https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset>, 2022. Accessed: 2024-04-17.

- [222] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020.
- [223] Xianyu Yu, Yuezhi Hu, Dequn Zhou, Qunwei Wang, Xiuzhi Sang, and Kai Huang. Carbon emission reduction analysis for cloud computing industry: can carbon emissions trading and technology innovation help? *Energy Economics*, 125:106804, 2023.
- [224] Vladimir Yussupov, Jacopo Soldani, Uwe Breitenbücher, and Frank Leymann. Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA. *Software: Practice and Experience*, 52(6):1454–1495, 2022.
- [225] Shiwen Zhang, Biao Hu, Wei Liang, Kuan-Ching Li, and Brij B Gupta. A caching-based dual k-anonymous location privacy-preserving scheme for edge computing. *IEEE Internet of Things Journal*, 2023.
- [226] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.
- [227] Haidong Zhao, Zakaria Benomar, Tobias Pfandzelter, and Nikolaos Georgantas. Supporting multi-cloud in serverless computing. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pages 285–290. IEEE, 2022.
- [228] Zhi Zhou, Fangming Liu, Yong Xu, Ruolan Zou, Hong Xu, John CS Lui, and Hai Jin. Carbon-aware load balancing for geo-distributed cloud services. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 232–241. IEEE, 2013.
- [229] Qingyuan Zhu, Xifan Chen, Malin Song, Xingchen Li, and Zhiyang Shen. Impacts of renewable electricity standard and renewable energy certificates on renewable energy investments and carbon emissions. *Journal of environmental management*, 306:114495, 2022.

Appendix A

Workflow Configuration

A.1 Configuration File

```
1 workflow_name: "example"
2 workflow_version: "0.0.1"
3 environment_variables:
4   - key: "ENV_VAR_1"
5     value: "value_1"
6 iam_policy_file: "iam_policy.json"
7 home_region:
8   provider: "aws"
9   region: "region_1"
10 constraints:
11   hard_resource_constraints:
12     cost:
13       type: "absolute" # Absolute value as 'absolute' (in USD)
14       value: 100
15     runtime:
16       type: "relative" # Percentage from deployment at home regions,
17       such as 102 for 2% tolerance
18       value: 102
19     carbon: null
20   soft_resource_constraints:
21     cost: null
22     runtime: null
23     carbon: null
24   priority_order:
25     - carbon
26     - cost
27     - runtime
28 regions_and_providers:
29   allowed_regions:
30     - provider: "aws"
31       region: "region_1"
32   disallowed_regions:
33     - provider: "aws"
34       region: "region_2"
35 providers:
36   aws:
37     config:
38       timeout: 60
39       memory: 128
40       additional_docker_commands:
41         - "yum update -y"
42         - "yum install -y tar gzip"
43         - "yum install -y xz"
```

Listing A.1: Example workflow configuration file.

A.2 IAM Policy File

```
1 {
2   "aws": {
3     "Version": "2012-10-17",
4     "Statement": [
5       {
6         "Action": [
7           "logs:CreateLogGroup",
8           "logs:CreateLogStream",
9           "logs:PutLogEvents"
10        ],
11        "Resource": "arn:aws:logs:*:*:*",
12        "Effect": "Allow"
13      },
14      {
15        "Action": ["sns:Publish"],
16        "Resource": "arn:aws:sns:*:*:*",
17        "Effect": "Allow"
18      },
19      {
20        "Action": ["dynamodb:GetItem", "dynamodb:UpdateItem"],
21        "Resource": "arn:aws:dynamodb:*:*:*",
22        "Effect": "Allow"
23      }
24    ]
25  }
26 }
```

Listing A.2: Minimal IAM Policy File.

Appendix B

Deployment Plan

B.1 Example Deployment Plan

```
1 {
2   "instances": {
3     "text_2_speech_censoring-0_0_1-GetInput:entry_point:0": {
4       "instance_name":
5         ↪ "text_2_speech_censoring-0_0_1-GetInput:entry_point:0",
6       "regions_and_providers": {
7         "allowed_regions": [
8           { "provider": "aws", "region": "us-east-1" },
9           { "provider": "aws", "region": "us-east-2" },
10          { "provider": "aws", "region": "us-west-1" },
11          { "provider": "aws", "region": "us-west-2" },
12          { "provider": "aws", "region": "ca-central-1" }
13        ],
14        "disallowed_regions": null,
15        "providers": {
16          "aws": {
17            "config": {
18              "timeout": 300,
19              "memory": 1024,
20              "additional_docker_commands": [
21                "yum update -y",
22                "yum install -y tar gzip",
23                "yum install -y xz",
24              ]
25            }
26          }
27        },
28        "succeeding_instances": [
29          "text_2_speech_censoring-0_0_1-Text2Speech:...",
30          "text_2_speech_censoring-0_0_1-Profanity:..."
31        ],
32        "preceding_instances": [],
33        "dependent_sync_predecessors": [
34          [
35            "text_2_speech_censoring-0_0_1-Compression:...",
36            "text_2_speech_censoring-0_0_1-Censor:sync:"
37          ],
38          [
39            "text_2_speech_censoring-0_0_1-Profanity:...",
40            "text_2_speech_censoring-0_0_1-Censor:sync:"
41          ]
42        ]
43      },
44      ...

```

```

45     },
46     "current_instance_name":
47         ⇨ "text_2_speech_censoring-0_0_1-GetInput:entry_point:0",
48     "workflow_placement": {
49         "home_deployment": {
50             "text_2_speech_censoring-0_0_1-GetInput:entry_point:0": {
51                 "identifier": "example_identifier",
52                 ⇨ "provider_region": { "provider": "aws", "region": "us-east-1"
53             },
54             "function_identifier": "example_function_identifier"
55         },
56     },
57     ...
58 }
59 }

```

Listing B.1: Example deployment plan based on the Text2Speech Censoring Workflow.

Appendix C

Developer Permissions

C.1 Required Permissions for Framework Developer Access

```
1 {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Action": [
6         "logs:CreateLogGroup",
7         "logs:CreateLogStream",
8         "logs:PutLogEvents"
9       ],
10      "Resource": "arn:aws:logs:*:*:*",
11      "Effect": "Allow"
12    },
13    {
14      "Action": [
15        "sns:Publish"
16      ],
17      "Resource": "arn:aws:sns:*:*:*",
18      "Effect": "Allow"
19    },
20    {
21      "Action": [
22        "dynamodb:GetItem",
23        "dynamodb:UpdateItem"
24      ],
25      "Resource": "arn:aws:dynamodb:*:*:*",
26      "Effect": "Allow"
27    },
28    {
29      "Action": [
30        "s3:GetObject",
31        "s3:PutObject"
32      ],
33      "Resource": "arn:aws:s3:*:*:*",
34      "Effect": "Allow"
35    },
36    {
37      "Sid": "Statement1",
38      "Effect": "Allow",
39      "Action": [
40        "iam:AttachRolePolicy",
41        "iam:CreateRole",
42        "iam:CreatePolicy",
43        "iam:PutRolePolicy",
```

```

44     "ecr:CreateRepository",
45     "ecr:GetAuthorizationToken",
46     "ecr:InitiateLayerUpload",
47     "ecr:UploadLayerPart",
48     "ecr:CompleteLayerUpload",
49     "ecr:BatchGetImage",
50     "ecr:BatchCheckLayerAvailability",
51     "ecr:DescribeImages",
52     "ecr:DescribeRepositories",
53     "ecr:GetDownloadUrlForLayer",
54     "ecr:ListImages",
55     "ecr:PutImage",
56     "ecr:SetRepositoryPolicy",
57     "ecr:GetRepositoryPolicy",
58     "ecr:DeleteRepository",
59     "lambda:GetFunction",
60     "lambda:AddPermission",
61     "pricing:ListPriceLists",
62     "pricing:GetPriceListFileUrl",
63     "logs:FilterLogEvents"
64 ],
65     "Resource": "*"
66 },
67 {
68     "Sid": "Statement2",
69     "Effect": "Allow",
70     "Action": [
71         "iam:GetRole",
72         "iam:PassRole"
73     ],
74     "Resource": "arn:aws:iam::*:*"
75 }
76 ]
77 }

```

Listing C.1: Outlining the minimal permissions required by the AWS role of the developer user and framework component functions.

Appendix D

Framework Implementation Details

D.1 Framework Source Code Files

Implementation File	Location in framework Architecture
caribou/common/models/remote_client/ remote_client.py	All Components (§5.4.3)
caribou/common/models/remote_client/ aws_remote_client.py	All Components
caribou/monitors/deployment_manager.py	Deployment Manager (§5.5.2)
caribou/deployment_solver/deployment_algorithms/ deployment_algorithm.py	Deployment Solver (§5.5.3)
caribou/deployment_solver/deployment_algorithms/ stochastic_heuristic_deployment_algorithm.py	Deployment Solver
caribou/deployment/client/cli/cli.py	Framework Command Line Interface (§5.4.1)
caribou/deployment/client/caribou_workflow.py	Deployment Utility (§5.5.4) & Execution (§5.5.6)
caribou/deployment/client/caribou_function.py	Deployment Utility & Execution
caribou/deployment/common/deploy/deployer.py	Deployment Utility & Deployment Migrator (§5.5.5)
caribou/deployment/common/deploy/ workflow_builder.py	Deployment Utility & Deployment Migrator
caribou/deployment/common/deploy/executor.py	Deployment Utility & Deployment Migrator
caribou/deployment/common/deploy/ deployment_packager.py	Deployment Utility
caribou/deployment/common/deploy_instructions/ deploy_instructions.py	Deployment Utility & Deployment Migrator
caribou/deployment/common/deploy_instructions/ aws_deploy_instructions.py	Deployment Utility & Deployment Migrator
caribou/monitors/deployment_migrator.py	Deployment Migrator
caribou/deployment/server/re_deployment_server.py	Deployment Migrator
caribou/deployment_solver/workflow_config.py	Multiple Components (§5.5.1)
caribou/deployment_solver/deployment_input/ input_manager.py	Metrics Manager (§5.5.7)

Table D.1: Main implemented Python files as part of the framework architecture and their corresponding component.

D.2 Framework Component Interaction

Table Name	Used by	Key	Value
workflow_placement_solver_-staging_area_table	Deployment Solver, Deployment Migrator	Workflow ID	New deployment placement
workflow_placement_decision_-table	Workflow & Function Wrapper, Invocation Client, Deployment Manager	Workflow ID	Deployment Plan
deployment_manager_workflow_-info_table	Deployment Manager	Workflow ID	Workflow state & config
deployment_resources_table	Deployment Utility, Deployment Migrator	Workflow ID	Deployed function information
sync_messages_table	Workflow & Function Wrapper	Workflow ID + Run ID + Sync Node Name	Intermediate Data
sync_predecessor_counter_table	Workflow & Function Wrapper	Workflow ID + Run ID + Sync Node Name	Conditional annotation data
caribou_workflow_images_table	Deployment Utility, Deployment Migrator	Workflow ID	Home region image names
available_regions_table	Deployment Solver, Metrics Manager	Region ID	Availability information
carbon_region_table	Metrics Manager	Region ID	Carbon forecasts for the next 24h from last update
performance_region_table	Metrics Manager	Region ID	Region to region latency data from CloudPing
provider_region_table	Metrics Manager	Region ID	Provider per region information such as pricing
provider_table	Metrics Manager	Provider ID	Provider general information
workflow_instance_table	Metrics Manager	Workflow ID	Aggregated past invocation data
workflow_summary_table	Metrics Manager	Workflow ID	Collected Logs

Table D.2: Component interaction tables.

Appendix E

Extended Evaluation Plots

E.1 Extended Efficiency of Self-Adaptive Re-Deployment

E.1.1 DNA Visualization

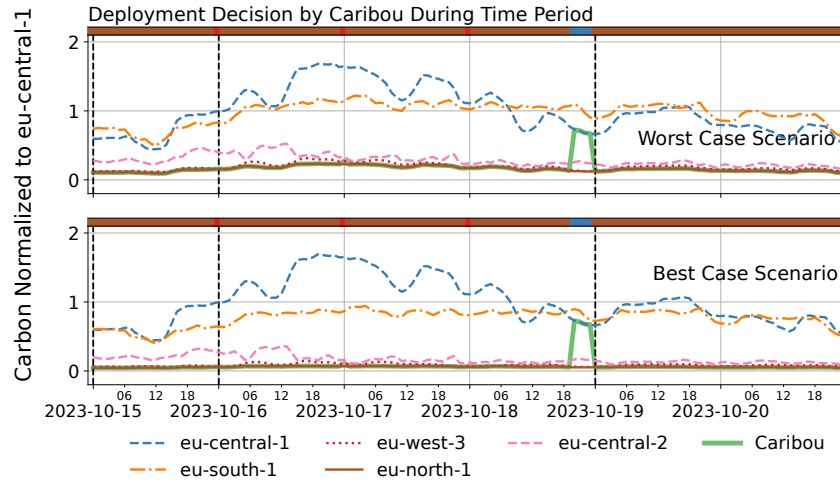


Figure E.1: Showcasing CARIBOU’s deployment decisions for DNA Visualization using the large input size, normalized to mean carbon emission of execution in `eu-central-1`.

E.1.2 Image Processing

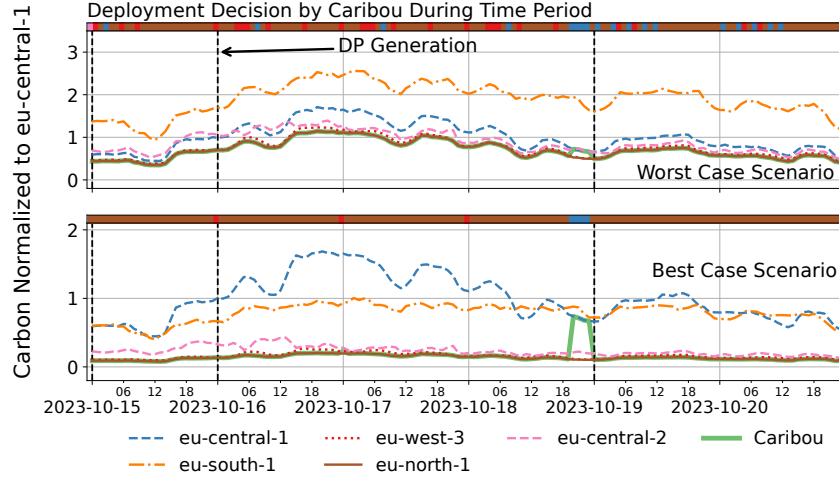


Figure E.2: Showcasing CARIBOU’s deployment decisions for Image Processing using the large input size, normalized to mean carbon emission of execution in eu-central-1.

E.1.3 Text2Speech Censoring

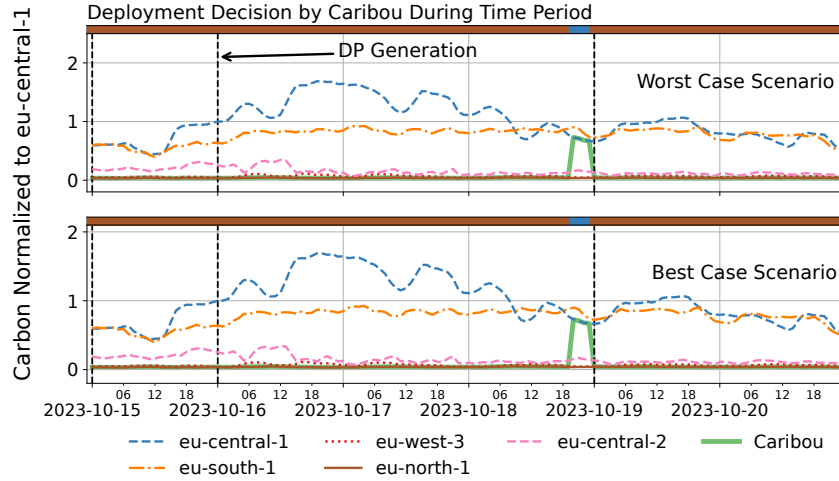


Figure E.3: Showcasing CARIBOU’s deployment decisions for Text2Speech Censoring using the large input size, normalized to mean carbon emission of execution in eu-central-1.

E.1.4 Video Analytics

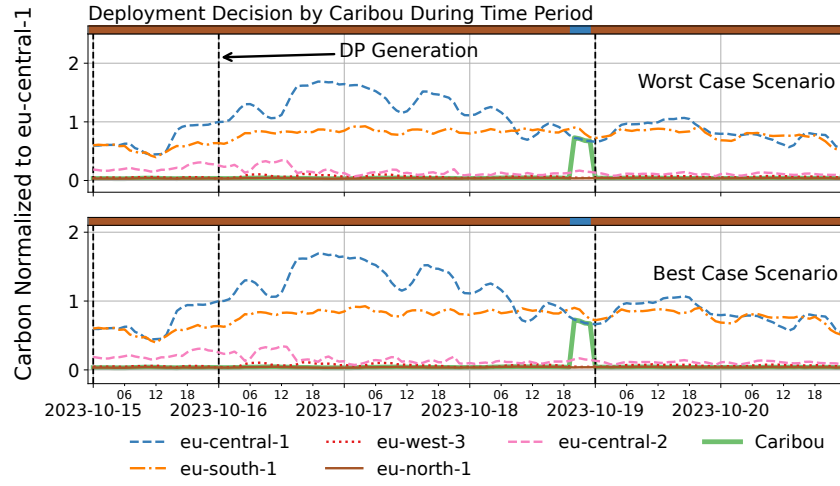


Figure E.4: Showcasing CARIBOU's deployment decisions for Video Analytics using the large input size, normalized to mean carbon emission of execution in `eu-central-1`.