

|  |
| --- |
|  |
| Carbon-aware Scheduling for non-crucial jobs in AWS |
|  |
| Green Computing Course Project Report |

Instructor: Dr. Ziliang Zong

Written by: Sri Sudha Kambhampati

Carbon-aware Scheduling for non-crucial jobs in AWS

Green Computing Course Project Report

**Introduction**

In the landscape of modern cloud computing, the rise of serverless architectures has revolutionized the way organizations deploy and manage their applications. Serverless computing offers unparalleled scalability, flexibility, and cost-effectiveness by abstracting the complexity of server management enabling developers to focus on writing code. Serverless workloads comes with the challenge of optimizing resource usage and minimizing the environmental impact.

The concept of environment aware scheduling complements the serverless paradigm by introducing a holistic approach to resource utilization and environmental stewardship. Rather than simply provisioning compute resources on-demand, environment aware scheduling considers factors such as energy efficiency, carbon emissions, and renewable energy integration. **This project focuses scheduling non-critical jobs based on the carbon emissions in a region i.e. carbon-aware scheduling**. By dynamically adjusting job scheduling based on real-time data on carbon emissions, organizations can minimize their environmental footprint while maximizing the efficiency of their serverless deployments thereby demonstrating their commitment to environmental responsibility and contribute to a greener future for generations to come.

**Problem Statement**

The problem statement concerns achieving **sustainability in AWS serverless operations by incorporating carbon awareness into the process of triggering the events on the cloud**. To achieve this, we need to solve for fetching the carbon intensity levels in the region, computing the lowest possible time frame to execute the trigger of the job, waiting for the right period to trigger the job and the finally triggering it. The outcome of this project would be developing a workflow which can be used by cloud users for their non-critical jobs to make them greener. Adapting such a workflow by a company for their day-to-day jobs can create a huge impact. This practice of carbon -aware scheduling aligns with global sustainability goals and demonstrates organizations’ commitment to environmental responsibility which enhances their reputation and motivates the community to drive innovation in green technology.

**Solution Approach**

**A diagram of a cloud job

Description automatically generated**

The idea of achieving sustainability is achieved either by “making software carbon efficient” or “by running software with carbon awareness”. We want to utilize the 2nd method. The methodology can be better explained using a use case and its flow:

Consider uploading a user video to cloud which needs to be converted to 3 different versions in of different resolutions. A lambda function can be defined to trigger this job for converting and uploading to cloud. This lambda function needs to measure the carbon intensity levels in that region, and the job needs to be triggered at the lowest possible level (with a calculation specified below).

To achieve this, we need below steps:

1) It all starts with user triggering a cloud job say, ‘Job 1’, which has a consequent job ‘Job 2’.

2) In order to make carbon-aware decision making for triggering the dependent job, we need the data. To get that, we start by integrating with Watt Time API which gives the carbon emissions in the region given.

3) Once Integrated, make API calls the collect the forecast data about the carbon footprint for the day ahead.

4) Define or build the AWS lambda function which needs to consider the user inputs like Window timeframe during which the Job 2 needs to run and maximum duration for this Job 2. We are taking hardcoded values for now, which can be found out analytically or based on machine learning predictions.

4) Based on the prediction forecast from WattTime API, logically we need to choose best time to schedule the user task within the given window timeframe.

5) Wait for the difference between the current time and best time according to the region of execution.

6) All of the above steps are orchestrated in AWS step functions which has the integration of cloud tools like CloudWatch etc., which can show the delay of the ‘Job 2’ scheduled

**Code Evaluation and Result for each of the above steps**

1. Watt Time Integration - Register with Watt-time API

import requests

register\_url = 'https://api.watttime.org/register'

params = {'username': 'abc',

'password': 'Green@123',

'email': 'abc@gmail.com',

'org': 'green world'}

rsp = requests.post(register\_url, json=params)

print(rsp.text)

Output is user being created successfully.

1. Get access token and make API call to get co2 prediction data.

import urllib3

from base64 import b64encode

import json

# Create a new PoolManager instance

http = urllib3.PoolManager()

# Encode the credentials

credentials = b64encode(b'sudhagreen:Green@123').decode('utf-8')

# Make the request including the basic auth header

response = http.request(

'GET',

'https://api.watttime.org/login',

headers={

'Authorization': f'Basic {credentials}'

}

)

output = response.data.decode('utf-8')

json\_object = json.loads(output)

# Print the response

print(json\_object['token'])

TOKEN = json\_object['token']

# Provide your TOKEN here, see https://docs.watttime.org/#tag/Authentication/operation/get\_token\_login\_get for more information

params = {

"signal\_type": "co2\_moer",

"region": "CAISO\_NORTH"

}

response = http.request(

'GET',

'https://api.watttime.org/v3/forecast',

headers={

'Authorization': f"Bearer {TOKEN}"

},

fields = params

)

data\_output = response.data.decode('utf-8')

print(data\_output)

Sample output is below:

A screenshot of a computer code

Description automatically generated

1. Visualizing the output data using python plots.

import matplotlib.pyplot as plt

from datetime import datetime

import pandas as pd

# Extract CO2 data from 'data' field

co2\_data = data\_output\_dict['data']

# Convert to DataFrame

data\_df = pd.DataFrame(co2\_data)

# Convert 'point\_time' to datetime

data\_df['point\_time'] = pd.to\_datetime(data\_df['point\_time'])

plt.figure(figsize=(10, 6))

plt.plot(data\_df['point\_time'], data\_df['value'], marker='o')

plt.xlabel('Time')

plt.ylabel('CO2 Levels')

plt.title('CO2 Levels Over Time')

plt.xticks(rotation=45)

plt.grid(True)

plt.tight\_layout()

plt.show()

A graph of a graph

Description automatically generated

Entire raw data is saved in the submission folder.

1. Calculating best time to trigger the job.

def find\_best\_trigger\_time(data, ex\_time\_frame, est\_ex\_duration):

"""

Find the best time to trigger a workload within a time window based on CO2 levels.

Parameters:

data (list): List of objects containing timestamp and CO2 levels.

ex\_time\_frame (int): Allowable Duration of the workload in hours.

est\_ex\_duration (int): Time for the execution of workload in minutes.

Returns:

str: Best time to trigger the workload.

"""

ex\_time\_frame = math.ceil((ex\_time\_frame \* 60) / 5) # Convert workload duration to minutes

best\_time = None

lowest\_avg\_co2 = float('inf')

# Iterate through each time slot within the time window

for i in range(ex\_time\_frame):

new\_val = est\_ex\_duration//5

# Calculate average CO2 level for the workload duration starting from this time slot

avg\_co2 = sum(data[i]['value'] for data[i] in data[i:i+new\_val]) / new\_val

if avg\_co2 < lowest\_avg\_co2:

lowest\_avg\_co2 = avg\_co2

best\_time = data[i]['point\_time']

best\_point = i

print(best\_time)

# Parse the given timestamp into a datetime object

given\_timestamp = datetime.fromisoformat(best\_time)

# Convert the given timestamp to the CAISO\_NORTH timezone

caiso\_north\_timezone = tz.gettz('America/Los\_Angeles')

given\_timestamp\_caiso\_north = given\_timestamp.astimezone(caiso\_north\_timezone)

# Get the current time in the CAISO\_NORTH timezone

current\_time\_caiso\_north = datetime.now(caiso\_north\_timezone)

time\_difference = given\_timestamp\_caiso\_north - current\_time\_caiso\_north

time\_difference\_seconds = int(time\_difference.total\_seconds())

#print(time\_difference\_seconds)

response = {

"waitTimeInSec": time\_difference\_seconds

}

#print(best\_point)

return response

Output of the above code depends on the two parameters ex\_time\_frame, est\_ex\_duration.

If we set the ex\_time\_frame= 24 hours, and est\_ex\_duration = 9 min, the above code returns the output 2024-04-29T22:00:00+00:00.

A screenshot of a computer program

Description automatically generated

From previous graph we can see that is the lowest possible point on the graph.

If we set the ex\_time\_frame= 8 hours, and est\_ex\_duration = 9 min, the above code returns the output 2024-04-29T09:50:00+00:00.

A screenshot of a computer program

Description automatically generated

1. Now comes the orchestration using AWS step function. Below is the flow in the step function.

A screenshot of a computer

Description automatically generated

The first step is the lambda function which includes the logic of all the above 4 steps and calculates the ‘waitTimeInSec’ to a wait state which waits for the specified time interval. After exiting from the wait state, we trigger another lambda which does our ‘Job 2’ which we want to schedule in a carbon-friendly manner.

1. Next comes the driver for the above Step function which is a lambda function. This lambda S3Upload\_trigger gets executed when any upload happens to S3.

A screenshot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**

So, let’s say ‘Job 1’ refers to any upload in S3 bucket. This when this job is executed, the lambda ‘S3Upload\_trigger’ gets executed and invokes the step function. As stated in above step, step function calculates best time, waits for the duration, and invokes another lambda which executes ‘Job 2’ which is dependent on ‘Job 1’.

**End-to-end Execution Flow**

1. Start by uploading a file to S3 bucket.

A screenshot of a computer

Description automatically generated

1. This upload triggered S3Upload\_trigger lambda and it executed successfully.

A screenshot of a computer

Description automatically generated

1. This triggered the step function workflow.

A screenshot of a computer

Description automatically generated

1. Step function started execution and computed best time and entered wait state.

**A screenshot of a computer

Description automatically generated**

1. After successfully waiting, the consequent action ‘Job 2’ has been completed (ex\_time\_frame is set to 1 hour). We can see that Job got triggered after 8 min 54 sec.

**A screenshot of a computer

Description automatically generated**

Another case where ex\_time\_frame is set to 8 hours where We can see that Job got triggered after 1 hour 28 min.

A screenshot of a computer

Description automatically generated

**Conclusion**

Carbon aware cloud scheduling using lambda functions can significantly contribute to reducing carbon emissions associated with cloud computing by optimizing the utilization of resources and minimizing energy consumption. By leveraging Lambda and Step functions, which are serverless and event-driven handles, organizations can implement dynamic workload scheduling strategies that align computing tasks with periods of low carbon intensity specific to a region in power grid.

**References**

[1] Challenges and Opportunities in Sustainable Serverless Computing, Prateek Sharma, Indiana University Bloomington, prateeks@iu.edu

[2] R. Ghafari, F. Hassani Kabutarkhani, and N. Mansouri. 2022. Task scheduling algorithms for energy optimization in cloud environment: a comprehensive review. Cluster Computing 25, 2 (Apr 2022), 1035–1093. <https://doi.org/10.1007/s10586-021-03512-z>

[3] <https://docs.watttime.org/>

[4] <https://aws.amazon.com/lambda/>