Carbon aware scheduling of serverless workflows

Sara Malaspina
dept. Computer Engineering, University of Rome Tor Vergata
Rome, Italy
sara.malaspina@alumni.uniroma2.eu

Silvia Perelli
dept. Computer Engineering, University of Rome Tor Vergata
Rome, Italy
silvia.perelli@alumni.uniroma2.eu

***Abstract*— The large use of cloud computing services is causing an increase in carbon emissions, making it necessary to pay more attention to resource allocation and consumption. The objective of this work is the implementation of a carbon aware scheduling of Lambda functions that distributes the load among the AWS Regions, considering the carbon footprint data from Electricity Maps API.**

I. Introduction

"Simply put, cloud computing is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping you lower your operating costs, run your infrastructure more efficiently, and scale as your business needs change" [1]. All these benefits make cloud computing largerly used in everyday life. On the other hand, excessive consumption of data centers is harmful to the environment. In fact, "Data centers consume an estimated 200 terawatt hours (TWh) of energy per year, and their electricity use is likely to increase about fifteen-fold by 2030 to 8% of projected total global electricity demand. For comparison, that is more than the national energy consumption of some countries, and 1% of global electricity demand. The data center industry contributes around 0,3% to overall carbon emissions, and the information and communications technology (ICT) ecosystem that depends on them — including personal devices, mobile phone networks and televisions — accounts for more than 2% of global emissions, according to Nature magazine. The carbon footprint of data centers is therefore a matter for concern, given that the growing ICT sector depends on them for its existence" [2]. As a result, it is necessary to develop new decarbonization techniques that lead to a reduction in the environmental impact of cloud services. Our work is focused on making more eco-sustainable the scheduling of serverless workflows. Serverless computing is a cloud computing model fully managed by Cloud provider. This platform allows users to develop, run and distribute applications code without the need to manage the underlying server and to take low-level infrastructure tasks, such as provisioning and scaling computing resources. Serverless benefits are automatic scaling, built-in high availability, event driven and a pay-for-use billing model, to increase agility and optimize costs. In this work, we used as cloud provider Amazon Web Services and especially we exploited as service AWS Lambda that runs code in response to events and makes easy and fast the communication beetween decoupled services. In our application we also used Amazon CloudWatch, Amazon EventBridge, Amazon Simple Storage Service (S3) and Amazon DynamoDB. We collected carbon footprint data with Electricity Maps, a platform with access to carbon emissions and electricity sources data for more than 230 regions, but we focused only on the zones corresponding to the AWS regions. The user can interact with the system choosing between four different services: calculation of Fibonacci number, matrix inversion, image resizing and calculation of linear regression coefficients.

II. Background

For the development of the application, we used the AWS Academy Learner Lab, which allowed us to use the AWS services from the console, assuming the LabRole IAM role. The programming language used was Python. The application provides a command line client, while all other components are implemented as Lambda functions. To execute the Lambda functions we imported the following libraries: *boto*, *numpy*, *pandas*, *json*, *sys*, *ast*, *pillow*, *io*, *so*, *requests* and *random*.

III. Solution desing

In Fig.1 is shown the architecture of the system. The user interacts with the command line client and can choose between four different services corresponding to the Lambda functions: *Fibonacci*, *ImageResizing*, *InverseMatrix* and *LinearRegression*. The user provides the input of the selected function; in particular, for the *ImageResizing*, he has to specify the path of an image on his device, and the client itself will upload the image on the S3 bucket *imgsource-bucket*. The client is also responsible for calling the *Scheduler*, to which it communicates the chosen function and the associated input in JSON format. The *Scheduler* is a Lambda function that routes the request on the most efficient AWS region. Based on the size of the user's input and the function data history stored on Amazon DynamoDB, the *Scheduler* classifies the function to be invoked as high or low intensity. The six best regions, sorted by EEI metric, are divided into two groups: the first three are intended to perform high-intensity functions, while the remaining three are intended for low-intensity functions. To distribute the load between regions of the same category is performed a round robin scheduling, keeping two indexes as state in the S3 bucket *roundrobin-bucket*. Then, the *Scheduler* invokes the Lambda function, chosen by the user, in the calculated region and updates the corresponding index for the round robin scheduling. The Lambda function *MetricServer* is responsible for retrieving data from the Electrity Maps API corresponding to the AWS regions and stores them in a file kept in the S3 bucket *apicarbon-bucket*. Data is overwritten hourly by an Amazon EventBridge event that triggers the function. Writing to *apicarbon-bucket* triggers the Lambda function *Analyzer* which calculates the EEI metric for each region and stores the six with the highest value in the S3 bucket *carbon-state-bucket*. To determine the final region, the *Scheduler* makes use of these six regions.

IV. Solution details

Below are described in more detail all the methodologies and tools used for application development.

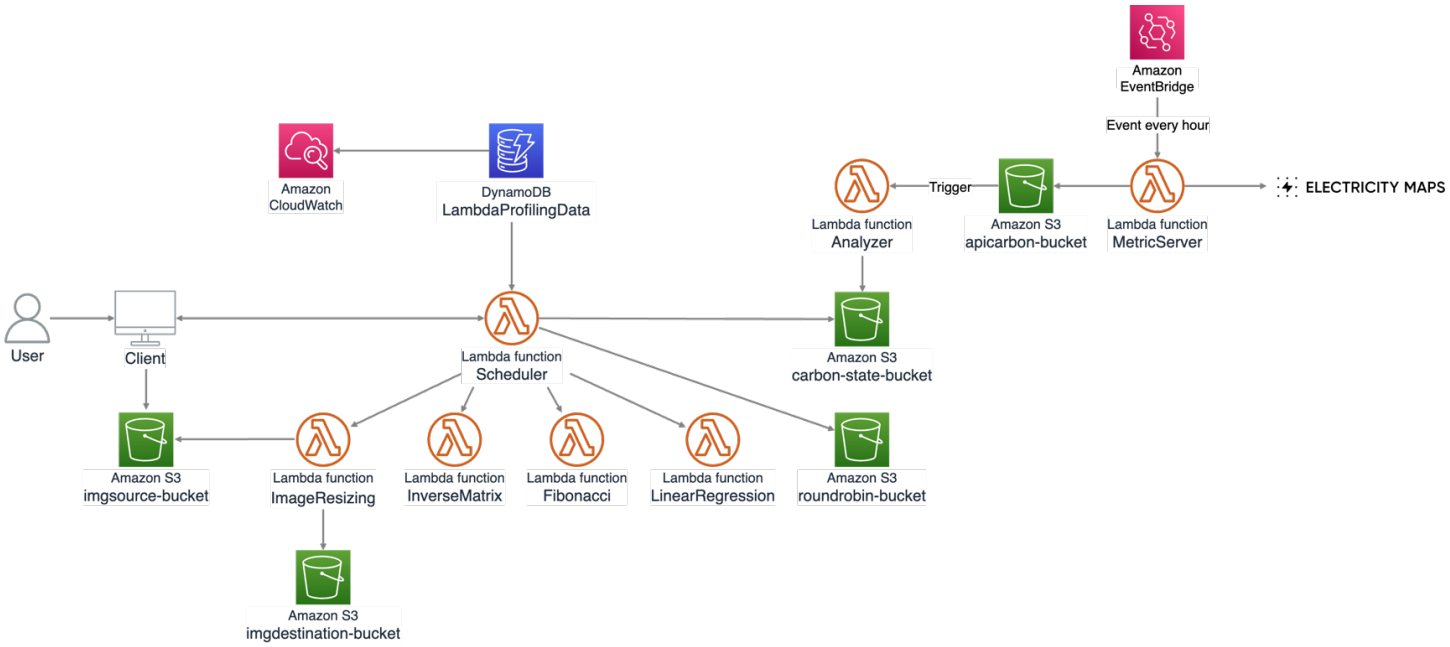

Fig. 1. System Architecture

A. Amazon Lambda Functions

The Lambda functions responsible for scheduling management, *Scheduler*, *Analyzer* and *MetricServer*, are instantiated in the us-east-1 AWS region, while Lambda functions related to the services offered by the application, *Fibonacci*, *ImageResizing*, *InverseMatrix* and *LinearRegression*, are replicated in all AWS regions where it is possible to invoke them. One of the advantages of Lambda functions is that you can easily configure the maximum amount of memory available and the timeout for the function used during execution. In our application all Lambda functions have a memory available of 128 MB and a timeout of 1 minute, except for *ImageResizing* which has 512 MB of memory available, requiring more intensive use of the CPU, whose allocation is proportional to the amount of memory configured. The instruction set architecture of our Lambda functions is x86_64. Furthermore, in every Lambda function code it is necessary to define a handler method that processes events; in our code it is the *lambda_handler*. When the function is invoked, Lambda runs the handler method.

B. Amazon EventBridge

Amazon EventBridge is a serverless service that uses events to connect application components together. We created an event to trigger every hour the MetricServer because Electricity Maps returns data with a resolution of one hour.

C. Amazon Simple Storage Service (S3)

Amazon Simple Storage Service (S3) is an object storage service where the objects are contained in buckets. All buckets of our system are instantiated in the AWS region us-east-1, but they can also be reached by Lambda functions in different regions without the need to create multiple instances of them. We created five buckets:

- *apicarbon-bucket*: stores two JSON files, *carbon_data.json* and *energy_type.json*, containing respectively the carbon intensity and the origin of the electricity in each AWS region.
- *carbon-state-bucket*: stores the JSON file *best_region.json* which contains the six best regions sorted by the EEI metric and divided into two groups, the ones intended for the invocation of high intensity functions and the others for low intensity functions.
- *imgsource-bucket*: stores images intended for resizing by the Lambda function *ImageResizing*.
- *imgdestination-bucket*: stores resized images after the invocation of the Lambda function *ImageResizing*.
- *roundrobin-bucket*: stores the JSON file *round_robin_state.json* that contains the indexes representing the next region where a Lambda function will be invoked, following a round robin scheduling.

D. Amazon CloudWatch

Amazon CloudWatch is a service that monitors applications and allows to visualize performance data. We used CloudWatch Logs Insights to search and analyze our log data and retrieve average memory and execution time of the Lambda functions through queries. In Fig. 2 is written the query used to obtain the average execution time, of multiple invocations, of a single Lambda function specified in the Logs groups section. In Fig. 3 is shown the equivalent query for the average memory. The queries were executed separately for the Lambda functions *Fibonacci*, *ImageResizing*, *InverseMatrix* and *LinearRegression*. We invoked the Lambda functions by running tests created on the AWS console, with different inputs, a significant number of times to be able to capture the performance trend. It was necessary to consider more invocations, as the first one suffers from cold start, due to which is experienced an increase in the time taken to perform the function.

E. Amazon DynamoDB

Amazon DynamoDB is a serverless, NoSQL database service. We created the table *LambdaProfilingData* to store the average execution time and memory used, obtained from

```
fields @timestamp, @message
| filter @message like /REPORT/
| parse @message /Duration: (?<duration>\d+\.\d+) ms/
| display @timestamp, duration
| stats avg(@duration) as avgDuration
```

Fig. 2. Query average execution time

```
fields @timestamp, @message
| filter @message like /REPORT/
| parse @message /Max Memory Used: (?<maxMemory>\d+\.\d+) MB/
| display @timestamp, maxMemory
| stats avg(maxMemory) as avgMaxMemory
```

Fig. 3. Query average memory

CloudWatch Logs Insights, and maximum and minimum size of input, of the Lambda functions: *Fibonacci*, *ImageResizing*, *InverseMatrix* and *LinearRegression*. We used this information as a data history of the Lambda functions.

F. Electricity Maps

We obtained carbon footprint data from Electricity Maps API. The API delivers data on carbon intensity and the power breakdown of electricity production and consumption with hourly granularity. We retrieved the last known carbon intensity (in gCO2eq/kWh) of electricity consumed in each AWS region with the following HTTP request:

https://api.electricitymap.org/v3/carbon-intensity/latest

specifying the zone identifier at the end. It returns the JSON, with units in gCO2eq/kWh, shown in Fig. 4. Origin of electricity in an area was collected with the following HTTP request:

https://api.electricitymap.org/v3/power-breakdown/latest

specifying the zone identifier as above. In this case the resulting JSON is shown in Fig. 5. According to Electricity Maps API documentation, ""powerProduction" (in MW) represents the electricity produced in the zone, broken down by production type, "powerConsumption" (in MW) represents the electricity consumed in the zone, after taking into account imports and exports, and broken down by production type, "powerExport" and "powerImport" (in MW) represent the physical electricity flows at the zone border, "renewablePercentage" and "fossilFreePercentage" refers to the % of the power consumption breakdown coming from renewables or fossil-free power plants (renewables and nuclear)" [3]. The API was called every hour to update the latest data. The obtained data were stored in the S3 bucket *apicarbon-bucket*, matching the zones provided by Electricity Maps with the AWS regions.

G. Metrics

The metric we considered to define the most sustainable regions is EcoEfficiency Index (EEI), defined as follows:

$$EEI = \frac{renewablePercentage}{carbonIntensity} * 100$$

where "carbonIntensity" and "renewablePercentage" are obtained respectively from the results in Fig. 4 and Fig. 5. Then, we applied a Min-Max Normalization to the EEI values and, multiplying the result by 100, we obtained values between 0 and 100:

$$EEI_{normalized} = \frac{EEI - EEI_{min}}{EEI_{max} - EEI_{min}} * 100$$

where EEI_{min} and EEI_{max} are the minimum and maximum values calculated on all regions. Alternatively, "renewablePercentage" can be calculated by summing the values of fields "geothermal", "biomass", "wind", "solar" and "hydro" related to "powerConsumptionBreakdown", dividing the result by "powerConsumptionTotal" and multiplying by 100 to obtain the percentage. The higher the EEI value, the more eco-friendly the region is, because the metric is directly proportional to the percentage of renewable energy and inversely proportional to the intensity of carbon emitted. We therefore ordered the regions in descending order of EEI value and selected the first six which were divided into two groups: the first three are used for invocation of high intensity functions being the most green, while the other three for low intensity functions. To distinguish between low and high intensity Lambda functions *Fibonacci*, *ImageResizing*, *InverseMatrix* and *LinearRegression*, we used a different approach based on the average values of runtime and memory used, related to the type of function invoked and stored in DynamoDB. We initially set two threshold values for both the runtime and memory used, so that if only one is exceeded by the corresponding average value, the function is classified directly as high intensity. Otherwise, a linear combination is made between the size of the input, the average runtime and the average memory used, each with its own weight, to calculate an index for the intensity. The three metrics are previously normalized, using a Min-Max normalization, and reduced to a range between 0 and 1, in order to be compared. For the input size, we considered the minimum and maximum value allowed, also stored in DynamoDB. For the execution time and for the memory, instead, we used as minimum 1 ms and 1 MB and for the maximum the threshold mentioned previously.

$$intensityIndex = w_{input} * input_{norm} + w_{time} * time_{norm} + w_{memory} * memory_{norm}$$

If the intensity index exceeds a given value, the function is classified as high intensity, otherwise it is low intensity.

H. Round Robin

We have chosen to make a round robin scheduling to distribute the load between regions. Each invocation selects a region in a circular way within a group. As round robin state we maintained an index, which indicates the position of the next region to choose. As the best regions have been divided into two groups, we decided to keep two separate round robin indexes, one referring to the regions where high intensity functions are invoked, and one for low intensity functions, as shown in Fig. 6. After each invocation, the corresponding index is updated as follows:

$$index = (index \bmod 3) + 1$$

where 3 is the number of the regions in each group. Since the regions are sorted by decreasing values of the EEI metric, it would be more sustainable to invoke all Lambda functions on the first region. However, in a context where many consecutive invocations are made, the best region would be overloaded, making it more convenient to distribute the workload also at the expense of energy efficiency.

```
{
  "zone": "US-MIDA-PJM",
  "carbonIntensity": 391,
  "datetime": "2024-08-07T09:00:00.000Z",
  "updatedAt": "2024-08-07T08:56:58.167Z",
  "createdAt": "2024-08-04T09:32:33.806Z",
  "emissionFactorType": "lifecycle",
  "isEstimated": true,
  "estimationMethod": "TIME_SLICER_AVERAGE"
}
```


Fig. 4. Carbon intensity JSON


```
{
  "zone": "US-MIDA-PJM",
  "datetime": "2024-08-07T09:00:00.000Z",
  "updatedAt": "2024-08-07T08:56:58.167Z",
  "createdAt": "2024-08-04T09:32:33.806Z",
  "powerConsumptionBreakdown": {
    "nuclear": 27705,
    "geothermal": 0,
    "biomass": 0,
    "coal": 11563,
    "wind": 3488,
    "solar": 0,
    "hydro": 1051,
    "gas": 33520,
    "oil": 231,
    "unknown": 552,
    "hydro discharge": 0,
    "battery discharge": 0
  },
  "powerProductionBreakdown": {
    "nuclear": 30762,
    "geothermal": null,
    "biomass": null,
    "coal": 12874,
    "wind": 3913,
    "solar": 0,
    "hydro": 1111,
    "gas": 37424,
    "oil": 260,
    "unknown": 617,
    "hydro discharge": null,
    "battery discharge": 0
  },
  "powerImportBreakdown": {
    "US-NY-NYIS": 0,
    "US-TEN-TVA": 686,
    "US-MIDW-MISO": 0
  },
  "powerExportBreakdown": {
    "US-NY-NYIS": 3199,
    "US-TEN-TVA": 0,
    "US-MIDW-MISO": 6339
  },
  "fossilFreePercentage": 41,
  "renewablePercentage": 6,
  "powerConsumptionTotal": 78110,
  "powerProductionTotal": 86962,
  "powerImportTotal": 686,
  "powerExportTotal": 9538,
  "isEstimated": true,
  "estimationMethod": "TIME_SLICER_AVERAGE"
}
```


Fig. 5. Power breakdown JSON


```
{
    "round_robin": [
        {
            "index": 1,
            "group": "low"
        },
        {
            "index": 3,
            "group": "high"
        }
    ]
}
```


Fig. 6. Round Robin State

V. RESULTS

The data history of Lambda functions *Fibonacci*, *LinearRegression*, *ImageResizing* and *InverseMatrix* is shown in Table I, where the fields "Execution Time" and "Memory Used" are average values of multiple invocations, measured in ms and MB respectively. Input values have different meanings: for *Fibonacci* they represent a minimum and maximum integer value, for *LinearRegression* the number of coefficients to calculate, for *ImageResizing* the image area in pixels x pixels and for *InverseMatrix* the number of row or column of the square matrix.

TABLE I

Name	*Execution Time*	*Memory Used*	*Min Input*	*Max Input*
Fibonacci	8,4982	31,6951	0	10.000
LinearRegression	7,9916	75	1	1.000
ImageResizing	1496,6932	160,3684	1	64.000.000
InverseMatrix	15,1337	77,7408	1	50

For a given function, the maximum memory allocated was almost unchanged between the various invocations, even with different input sizes, except for *ImageResizing* where we noticed different values as the image size changed. On the contrary, the time of execution varied more significantly, but only if the input size changes considerably as well. For all functions, the first invocation requires a longer execution time due to a cold start phase, where the service prepares the run environment. Let's examine the cold start: "When the Lambda service receives a request to run a function via the Lambda API, the service first prepares an execution environment. During this step, the service downloads the code for the function, which is stored in an internal S3 bucket (or in Amazon ECR if the function uses container packaging). It then creates an environment with the memory, runtime, and configuration specified. Once complete, Lambda runs any initialization code outside of the event handler before finally running the handler code. […] You are not charged for the time it takes for Lambda to prepare the function but it does add latency to the overall invocation duration. After the execution completes, the execution environment is frozen. To improve resource management and performance, the Lambda service retains the execution environment for a non-deterministic period of time. During this time, if another request arrives for the same function, the service may reuse the environment. This second request typically finishes more quickly, since the execution environment already exists and it's not necessary to download the code and run the initialization code. This is called a "warm start". According

to an analysis of production Lambda workloads, cold starts typically occur in under 1% of invocations. The duration of a cold start varies from under 100 ms to over 1 second" [4]. Comparing the results between the different functions, it appears that *ImageResizing* is much more intensive than the others, which have an average execution time of less than 20 ms. As for memory used, the difference between the functions is less drastic, but ImageResizing confirms to be the most expensive. Therefore, it seemed reasonable to consider as high intensity, regardless of the size of the input, the functions that present an average value of execution time and memory above, respectively, 100 ms and 100 MB. In all other cases we calculate the intensity index by normalizing the execution time with respect to a minimum and maximum value of 1 ms and 100 ms, while for the memory with respect to 1 MB and 100 MB. The input size of the invocated function is normalized assuming as minimum and maximum dimensions the ones stored in Table I. We choose the minimum input as the first admissible value to invoke the function, while for the maximum input we relied on our test experiences; the function *Fibonacci* does not return a valid result with a number in input greater than 10.000, while for *LinearRegression*, *ImageResizing* and *InverseMatrix* we set the highest reasonable value that can be specified via command line. In addition, to calculate the intensity index, we set the weights used in the linear combination. We have stored, for each function, only the average values of runtime and memory used, without distinguishing them by different input ranges; however, the size of the input is significant in determining the intensity. For this reason, we have chosen to give a weight equal to 0,4 for the size of the input and a weight equal to 0,3 both for time and memory. With this set of parameters, even functions with time and memory values below 100 ms and 100 MB can be classified as high intensity, comparing the intensity index with a threshold value of 0,5. It is possible to derive the input values by which the function will be classified as high intensity as follows:

$$\text{input} \geq \frac{0{,}5 - 0{,}3 * \text{time}_{\text{norm}} - 0{,}3 * \text{memory}_{\text{norm}}}{0{,}4}$$

The minimum normalized input to be classified as high intensity function is shown in Table II.

TABLE II

Name	***Min High Intensity Input***
Fibonacci	0,96
LinearRegression	0,64
InverseMatrix	0,56

ImageResizing was not considered because it does not require the calculation of the intensity index, being immediately classified as high intensity function. From the result in Table II, it follows that *Fibonacci* is the least computationally expensive Lambda function.

VI. Discussion

With AWS Academy Learner Lab all service access is limited to the us-east-1 and us-west-2 regions, being able to invoke the Lambda functions only in these two zones; when the *Scheduler* selects a different region, a random choice is made between the two reachable. With a permission upgrade, the system could be improved by extending the function invocation to all desired regions. Moreover, it is easy to add new features to the system, by modifying the Client and *Scheduler* and instantiating a new Lambda function in all regions where it can be invoked. The most demanding part is to characterize the new function because you need to collect time and memory, to store a data history on DynamoDB. Further refinement may be to classify functions, not only between low and high intensity, but also add a class of medium intensity. To make the prediction of the type of function more accurate, a more detailed history can be created, dividing it according to different input ranges.

References

[1] Microsoft Azure, "What is cloud computing?," [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing.[Accessed: Aug. 10, 2024]

[2] Enel X, "Data centers: the impact of digital transformation on energy consumption," Dec. 2021. [Online]. Available: https://corporate.enelx.com/en/stories/2021/12/data-center-industry-sustainability. Accessed: Aug. 10, 2024]

[3] Electricity Maps API Documentation, "Recent power breakdown history," [Online]. Available: https://static.electricitymaps.com/api/docs/index.html#recentpower-breakdown-history. [Accessed: Aug. 10, 2024]

[4] Amazon Web Services. Execution environments. AWS Lambda Operator Guide. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html. [Accessed: Aug. 10, 2024]