Cloud Offloading for Image classification

Alexander Lorenz
Vienna University of Technology

Tobias Grantner
Vienna University of Technology

Maximilian Kleinegger
Vienna University of Technology

1 Introduction

This report presents the challenges, approaches, and results of offloading an object detection task in the cloud versus executing it locally within a web server. Simply put, we aim to evaluate the different performance aspects of transfer- and inference-time between those execution models and how well task offloading is suitable for such kind of tasks. Therefore, this report includes a description of the problem, detailed information about it, the approach taken to solve it, and a conclusion on the topic.

2 Problem Overview

This section explains the problem we want to solve with a task offloading. We want to perform object detection with the YOLO-model[1] using the OpenCV[2]-library in Python. For comparison, we have to implement two approaches:

- *Local execution (LE)* uses a simple web server where we perform object detection. Therefore the clients send images to a specified endpoint and receive the objects detected with their respective accuracy.
- *Remote execution (RE)* uses a lambda function to offload the object-detection task into the cloud. Therefore the procedure is slightly different by uploading an image, which then triggers the lambda function and saves the results into a DynamoDB database.

Both approaches are needed to compare the effectiveness of the offloading approach, by considering image transfer times and inference times. The evaluation happens through a sample group of the COCO-dataset[3]. Further details of implementation, results and conclusions can be found in the following sections.

3 Methodology and Approach

This section presents our methodology and approach. Therefore we present architectural design choices, and implementation details of our LE and RE, as well as the setup of our AWS environment. This had to be done because we got access to an AWS learners lab to do this assignment.

3.1 Local Architecture

This section briefly describes the local architecture, so we can keep it in mind when comparing *LE* vs *RE*.

We provided a Flask[4] Webserver, which initially loads the config and weights for the YOLO-model and initializes it. Furthermore, we included an API-endpoint *http://localhost:5001/object-detection* which is reachable with the POST-verb and takes in a JSON object containing a unique-id and the image encoded as base64-string looking like:

[1] `https://pjreddie.com/darknet/yolo/`
[2] `https://pypi.org/project/opencv-python/`
[3] `http://cocodataset.org/#home`
[4] `https://flask.palletsprojects.com/en/3.0.x/`

```
{
    "id": "UUID",
    "image_data": "base64_encoded_image"
}
```

This endpoint responds with a JSON object containing the provided unique-id, an array containing accuracy for each label detected, and the inference time as milliseconds looking like:

```
{
    "id": "UUID"
    "objects": [
        {"label": "label1", "accuracy": 1}
        {"label": "label2", "accuracy": 0.5}
        ...
    ],
    "inference_time": 10
}
```

This comparable rather simple design, is effective and efficient, especially for our use case.

3.1.1 Client

To easily test the application, we built a client that sends all images provided, through a folder, to the *LE* and then prints the results and calculates the average inference time.

3.2 Remote Architecture

The figure 1 illustrates the conceptual architecture of our offloading task for a serverless workflow.

First, the client uploads images to an S3 bucket via a POST request, which triggers a Lambda function. This function executes the code in a container image provided by us, which is fetched from a repository and includes the YOLO config files, and performs object detection on the uploaded images. The detection results are then stored in a DynamoDB database.

3.2.1 AWS setup

This sub-section precisely describes how we set up the AWS environment. We used the AWS CLI to set up all the needed parts and the commands used to set up the environment can be found in the `Makefile` in our code. Important to note: we did not change any default settings and only chose the LabRole for permissions.

First of all, we set up an Elastic Container Registry (ECR) repository called `object-detection-lambda` to provide the image containing the code to be run by the lambda function. We then build the image including the YOLO configs and push it to the repository. This approach allowed us to avoid problems we had when importing AWS lambda function layers directly and is likely more efficient than an approach loading the YOLO configs externally upon execution.

Secondly, we created the S3 bucket, to which we can upload the images, which should be processed by the lambda function. We called this bucket `image-bucket-group41`.

Next, we set up the DynamoDB table with the name `object-detection-group41`, which is used to store the results of the object detection on an image. It used the key `image-url` as a key, which is the URL pointing to the S3 location where the image is stored, and stores the confidence score of a detected object in a column named according to the object label. This approach is beneficial to storing a list of all matched objects in a single column since it allows more efficient use of object labels in queries.

After that, we created a lambda function, with the name `object-detection`, which takes a URL pointing to an image, loads the image, performs object detection on it and stores the result in the table as described before. We configure the lambda function to be able to use 512 MB of memory for a maximum of 15 seconds since object detection is a rather compute-heavy task.

Lastly, we add a trigger to the lambda function we created, in order to start the execution of the function every time an item gets created in the `image-bucket-group41`-bucket.

All these steps can be executed by using the `make lambda-setup` target in the `Makefile` we provided. With everything set now, we only need a way to upload the images to the bucket to get the object detection started.

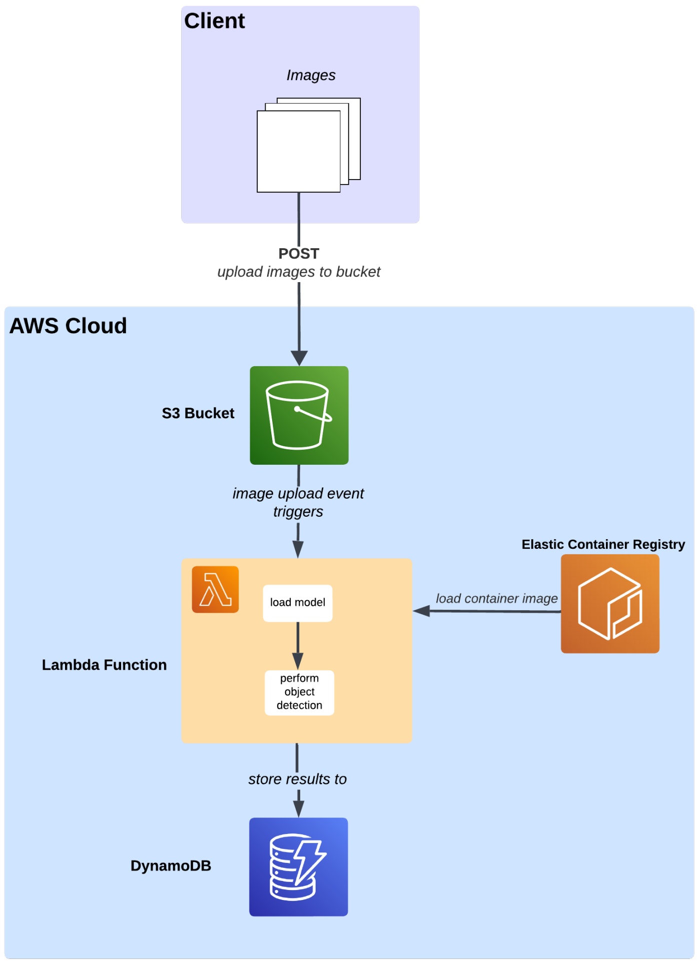

Figure 1: Offloading architecture

3.2.2 Client

To also easily test the *RE*, we extended the Flask application and client to upload the images to the S3 bucket and calculate the average transfer time.

4 Results

To compare the performance of the local and the remote execution approaches, we took a closer look at different aspects of their execution. For the local execution, which we performed on a laptop with a Ryzen 7840S CPU and 32GB of RAM, we measured the execution time from the request initiation to receiving the object detection results. For the remote execution, we also evaluated the execution time provided in the logs of the lambda function and additionally included the memory used. However, the execution time of the lambda function is only one part of the remote execution, for the upload of the image to the S3 bucket, we collected the transfer time. The results are summarized in table 1.

Task	Metric	Mean	SD	Min	Max
Local Execution	Duration (ms)	32.41	7.62	21.00	58.00
Remote Image Upload	Duration (ms)	48.17	14.86	36.00	152.00
Remote Execution	Duration (ms)	1780.20	378.99	1214.06	3008.06
Remote Execution	Memory Used (MB)	452.97	1.47	449.00	454.00

Table 1: Performance of local and remote execution

The time it takes to execute the object detection locally generally appears to be the fastest of all reported metrics. This could lead one to believe that local execution is always the best option, however, these results were achieved on rather modern hardware and could look completely different on an older machine.

Uploading the images takes only slightly longer than local execution in our environment. However, upload time highly depends on the available internet connection and could change drastically in a different environment. The execution of object detection inside of a lambda function is on average about 5 times as slow as our local execution measurements.

Overall, we can see that the remote execution takes significantly longer. However, if the user does not need the results in real-time, the latency directly apparent to the user during remote execution, the upload time, could be lower for users with limited hardware availability.

Therefore, we can say that for settings requiring real-time results, the use of local execution is beneficial due to its low latency. In case of tasks that are not time-critical and for which performant hardware availability is not expected, it might make more sense to choose remote execution.

Additionally, serverless execution offers excellent scalability out of the box and would be advantageous when dealing with a large number of tasks are expected, demand fluctuates heavily or when local resources are limited. This is also reflected in the costs of each approach, where having the necessary resources available for local execution would be costly for infrequent spikes in demand, cloud-based execution costs are based on usage and thus adjust the any type of demand.

5 Conclusion

Our analysis reveals that local execution outperforms remote execution in terms of latency, making it suitable for applications requiring immediate response times. However, offloading to AWS Lambda provides a scalable solution for handling intensive computations without taxing local resources, but resulting in higher latency.

Conclusively, local execution is ideal for real-time, low-latency requirements, while remote execution is better suited for batch processing or scenarios where computational demands exceed local capabilities.