Serving Deep Learning Model: Server vs Serverless Approach

Aanchal Khandelwal (ak8257)
Ruchika Upadhyay (ru2025)

1.Introduction

Our goal is to compare the inference performance of an object detection model in two different paradigms: server and serverless platforms. We will be choosing AWS Lambda and AWS Ec2 for our experiments. For Deep Learning models, we want to test and experiment to find out which of the two paradigms would perform better for the task of object detection.

2.Background Work

- Amazon Web Services' Elastic Compute Cloud (EC2) service was introduced to ease the provision of computing resources for developers. Its primary job is to provide self-service, on-demand, and resilient infrastructure.
- AWS Lambda was launched to eliminate infrastructure management of computing. It enables developers to concentrate on writing the function code without having to worry about provisioning infrastructure.

3.Approach

List of chosen Pretrained models based on speed and size:

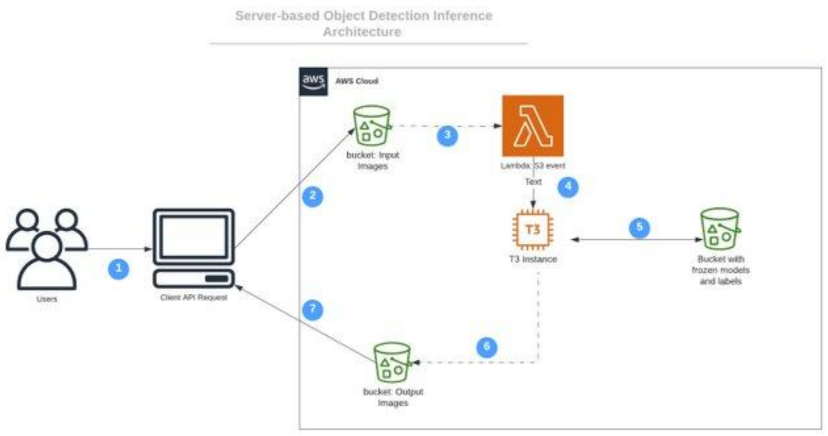
Model	Speed(ms)	Size(MB)
ssd_mobilenet_v1_0.75_depth_coco	26	15.8
ssdlite_mobilenet_v2_coco	27	17.5
faster_rcnn_inception_v2_coco	58	50.9
faster_rcnn_resnet50_coco	89	168.7
faster_rcnn_resnet101_coco	106	241.4

Dataset: https://cocodataset.org/#home

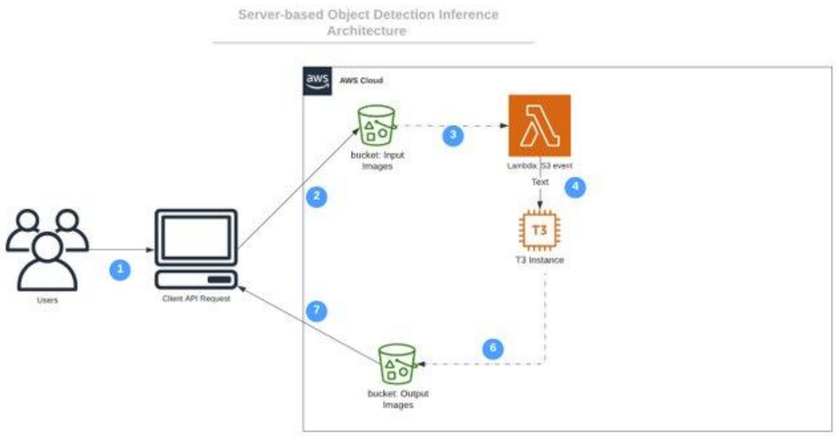
ARCHITECTURES

Server-Based:

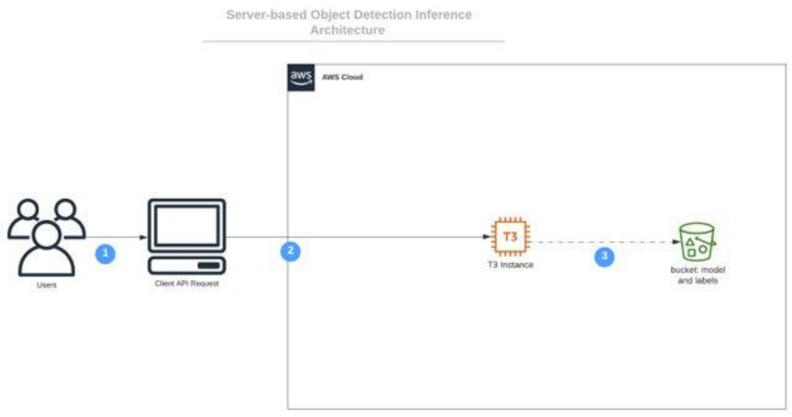
1. The input image by user is stored in s3 bucket, which invokes a lambda function, the encoded image is then sent to ec2 for inferencing. Ec2 picks up the model from s3 bucket and puts the resulting image in s3.



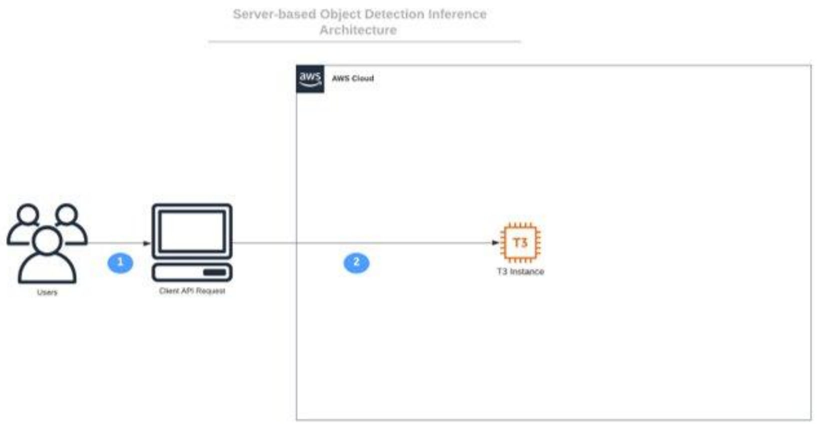
2. The input image by user is stored in s3 bucket, which invokes a lambda function, the encoded image is then sent to ec2 for inferencing. The model is stored in Ec2 instance itself, the resulting image is put in s3.



3. The input user provides an encoded string for the image in the API request body. The image is decoded in ec2 and the model is picked up from s3 bucket. The inference image is returned to the user in API response as an encoded string.

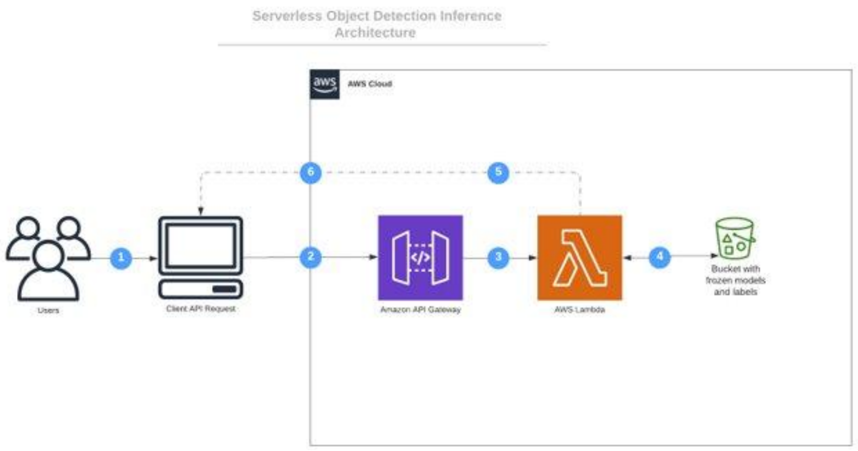


4. The input user provides an encoded string for the image in the API request body. The decoded image and the model are present is Ec2. The inference image is returned to the user in API response as an encoded string.



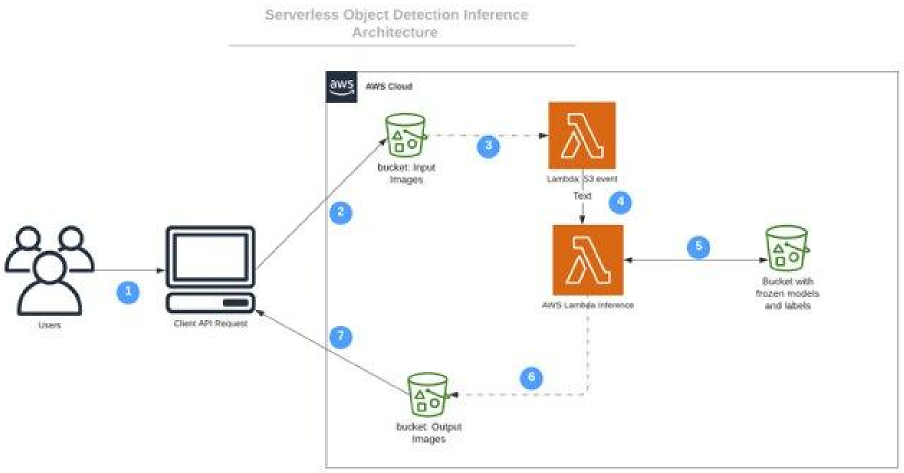
Serverless:

5. The input user provides an encoded string for the image in the API request body. The API gateway invokes a lambda function for inferencing, the model is picked up from s3 bucket. The result is returned to the user as an encoded string.

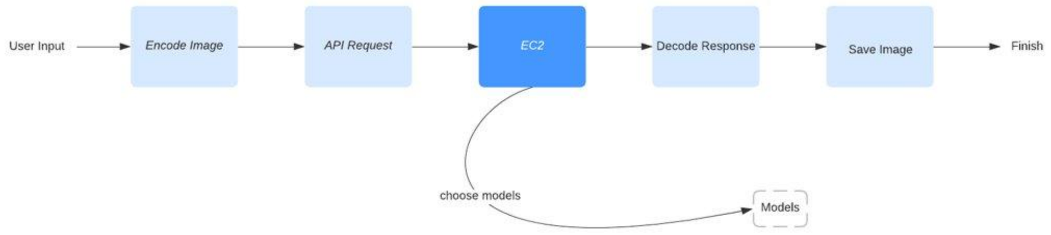


Serverless Object Detection Inference Architecture
AWS Cloud
Users
Client API Request
Amazon API Gateway
AWS Lambda
Bucket with frozen models and labels

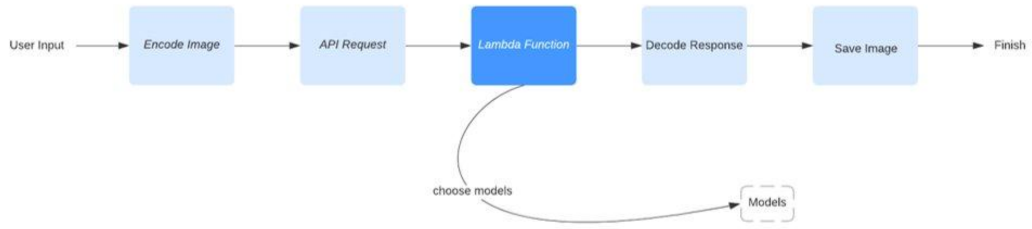
6. The input user uploads image to s3 bucket, s3 invokes lambda function first to encode the image which calls another lambda for inferencing. The second lambda stores result in s3 bucket.



The generic flow for server based inferencing:



The generic flow for serverless inferencing



Architecture #	Ease of use	Cost Rank S and SL	Speed Rank
1	Good	Server-based-4	4
2	Good	Server-based-2	3
3	Average	Server-based-3	2
4	Average	Server-based-1	1
5	Average	Serverless-1	5
6	Good	Serverless-2	6

4.Technical Challenges

1. Tensorflow is a heavy library and can not be installed on t2.micro, so we used t2.small which is not available in free tier.
2. Lambda allows upto 500 MB in /tmp and lambda code and layers allowance is upto 250 MB, the tensorflow library needs to be trimmed such that only required modules are added.

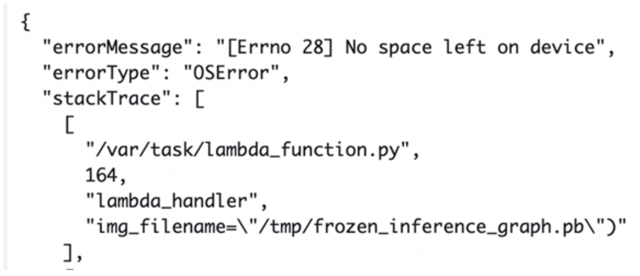
5. Results

Setup: In case of EC2, setup involves figuring out the instance type we needed for our API to work, whereas in Lambda we won't need to worry about the provisioning of such instances, as it is handled by Lambda service. When using EC2, there is no restriction on the amount of dependencies that an application can have. However, there are limits to the maximum size of a package with Lambda - 50 MB (zipped, for direct upload) and 250 MB (unzipped, including layers). These sizes aren't always enough, especially for machine learning systems that require a lot of third-party libraries.

During function execution, AWS advises installing and downloading dependencies to the /tmp directory. When a new container is formed, however, downloading all of the dependencies from scratch can take a long time. So, this option is good when your lambda container is up most of the time, otherwise it may cause a long cold start time for each invocation. Also, /tmp folder can hold a maximum of 512MB only. So, it is again restricted for limited use only. Hence, EC2 was easier to set up for a DL inference model.

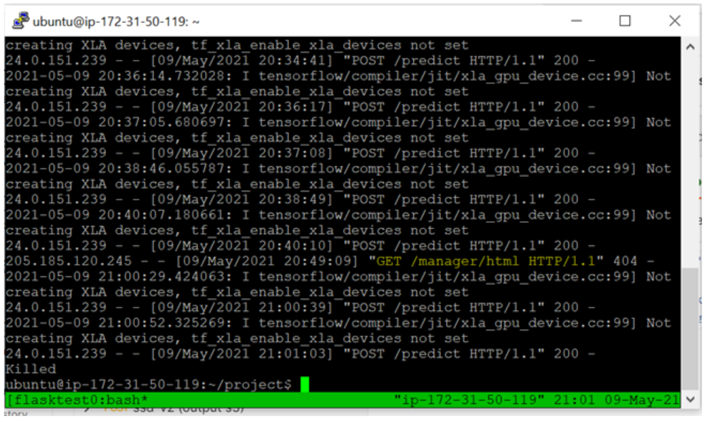
Scalability: In EC2, we need to manage the scaling by setting up an Auto Scaling group. ALB also needs to be set up to do the load balancing in case multiple instances of applications are installed using multiple EC2 instances. Whereas in Lambda, scaling is automated. We just need to configure how many max concurrent executions we want to allow for a function. Load balancing will be handled by Lambda itself.

Memory Issue: For heavier models such as Faster RCNN Resnet 101 and Faster RCNN Resnet 50, Lambda failed, as 500MB was getting exhausted. In the case of the former, 0 calls worked successfully, however for the later, it failed after multiple calls.



```
{
  "errorMessage": "[Errno 28] No space left on device",
  "errorType": "OSError",
  "stackTrace": [
    [
      "/var/task/lambda_function.py",
      164,
      "lambda_handler",
      "img_filename=\"/tmp/frozen_inference_graph.pb\")"
    ],
```

As for EC2, due to just 2GB RAM, multiple calls for Faster RCNN Resnet 101 were killing the Flask application as the model is computationally really expensive.

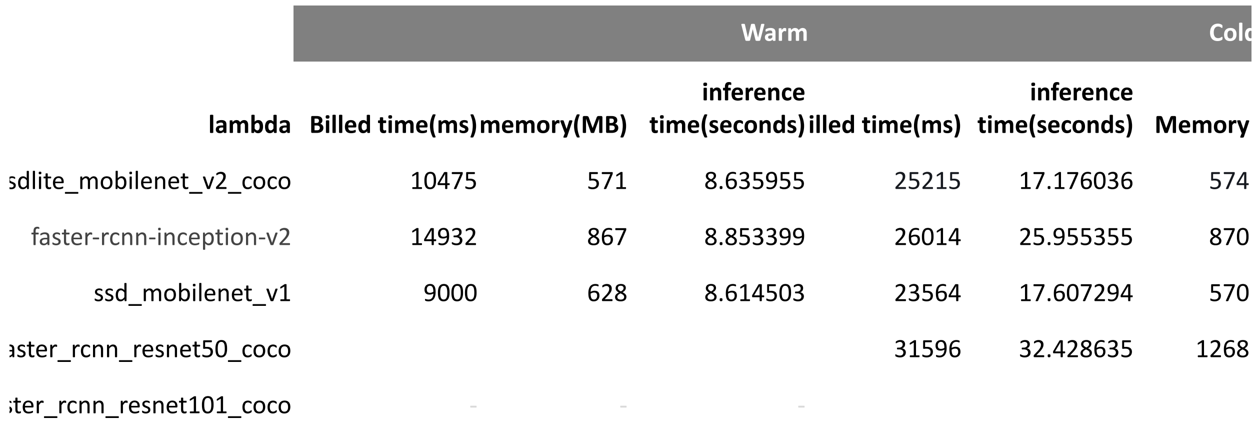


```
ubuntu@ip-172-31-50-119: ~
creating XLA devices, tf_xla_enable_xla_devices not set
24.0.151.239 - - [09/May/2021 20:34:41] "POST /predict HTTP/1.1" 200 -
2021-05-09 20:36:14.732028: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not
creating XLA devices, tf_xla_enable_xla_devices not set
24.0.151.239 - - [09/May/2021 20:36:17] "POST /predict HTTP/1.1" 200 -
2021-05-09 20:37:05.680697: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not
creating XLA devices, tf_xla_enable_xla_devices not set
24.0.151.239 - - [09/May/2021 20:37:08] "POST /predict HTTP/1.1" 200 -
2021-05-09 20:38:46.055787: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not
creating XLA devices, tf_xla_enable_xla_devices not set
24.0.151.239 - - [09/May/2021 20:38:49] "POST /predict HTTP/1.1" 200 -
2021-05-09 20:40:07.180661: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not
creating XLA devices, tf_xla_enable_xla_devices not set
24.0.151.239 - - [09/May/2021 20:40:10] "POST /predict HTTP/1.1" 200 -
205.185.120.245 - - [09/May/2021 20:49:09] "GET /manager/html HTTP/1.1" 404 -
2021-05-09 21:00:29.424063: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not
creating XLA devices, tf_xla_enable_xla_devices not set
24.0.151.239 - - [09/May/2021 21:00:39] "POST /predict HTTP/1.1" 200 -
2021-05-09 21:00:52.325269: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not
creating XLA devices, tf_xla_enable_xla_devices not set
24.0.151.239 - - [09/May/2021 21:01:03] "POST /predict HTTP/1.1" 200 -
Killed
ubuntu@ip-172-31-50-119:~/project$
[flasktest0:bash*                        "ip-172-31-50-119" 21:01 09-May-21
```

Cost: If we were to make 1800 calls per day, both EC2 and Lambda would cost us the same amount. However, If there are more than 1800 API calls for inferencing per day, EC2 is a winner and vice-a-versa.

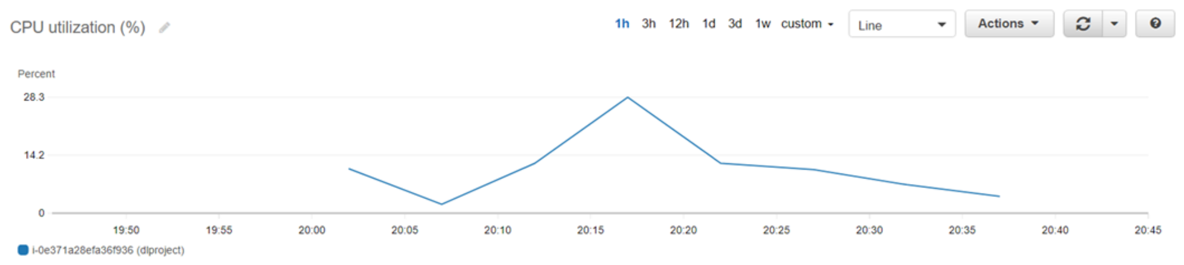
EC2 (1800 calls/day)	**Lambda(1800 calls/day : 1024 MB)**
$0.023 per hour	Pay for what you use
.023(usd) X 24 (hours) = $0.552	0.0000000167(usd) X 18386.40(ms) X 1800 = $0.552

Performance: Lambda Cold Start: Setting up a container and performing the appropriate bootstrapping takes time, which adds delay to any serverless function call. The client often notices this increased latency, known as the cold start effect, when the serverless function is invoked for the first time. The platform tries to reuse the container for successive invocations of the serverless function to reduce latency and prevent bootstrapping time. Warm start refers to the latency experienced by the client when a container is reused. Below are the observed results for the two from our experiments.

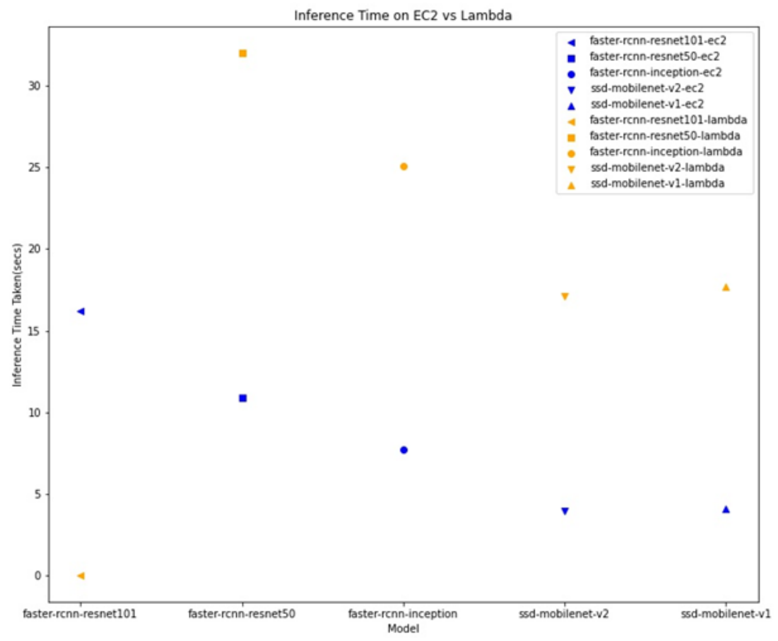


			Warm			Colc
lambda	**Billed time(ms)**	**memory(MB)**	**inference time(seconds)**	**illed time(ms)**	**inference time(seconds)**	**Memory**
sdlite_mobilenet_v2_coco	10475	571	8.635955	25215	17.176036	574
faster-rcnn-inception-v2	14932	867	8.853399	26014	25.955355	870
ssd_mobilenet_v1	9000	628	8.614503	23564	17.607294	570
aster_rcnn_resnet50_coco				31596	32.428635	1268
ter_rcnn_resnet101_coco	-	-	-			

EC2 CPU Utilization: Below is a graph of average CPU utilization during the experiments. The order of models in which we ran the experiments is: Faster RCNN Resnet 101, Faster RCNN Resnet 50, Faster RCNN Inception, SSD Mobilenet V2, SSD Mobilenet V1. As expected least utilization is during the SSD Mobilnet runs, whereas the max utilization was during Faster RCNN Resnet 50 as the application was restarted after Faster RCNN Resnet 101 calls killed the application.



Below is the final comparison between the inference time of the two paradigms.



6. Conclusion

Through our results one can find out a better suitable solution for their use case. In case of setup, we found EC2 to be the clear winner. Same was observed for inference latency. There were memory issues in both the cases: Lambda failed for the 2 heavier models while EC2 failed for the most computationally expensive one. While Lambda is a more scalable option, in terms of cost, either of the two could be a better approach (depending on the use case). So overall, we find that the better paradigm depends on the use case of the user, the decision maker being the load (number of API calls) on the application.

7. References

- Tensorflow coco trained models: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md
- Model Analysis: https://arxiv.org/abs/1605.07678
- Lambda Setup - https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
  https://aws.amazon.com/lambda/pricing/
- Lambda DL Setup Example - https://aws.amazon.com/blogs/machine-learning/how-to-deploy-deep-learning-models-with-aws-lambda-and-tensorflow/
- EC2 Setup - https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html
- EC2 Flask deployment - https://medium.com/techfront/step-by-step-visual-guide-on-deploying-a-flask-application-on-aws-ec2-8e3e8b82c4f7