Image Classification using Amazon Web Service

Trina Sahoo (1901254)
sahoo.1901254@studenti.uniroma1.it

Debodeep Banerjee (1901253)
banerjee.1901253@studenti.uniroma1.it

Introduction

In modern world, the far fetched applications of deep learning has revolutionized the domain of real world data. While talking about data, we no longer restrict ourselves in thinking about conventional tabular data, rather we realized that data can of any form such as image, text, sound, video etc. Parallel to this, another task is to show case a real time application of the deep learning model. So, in this project, we are addressing both the pillars and hence we are proposing a work flow which would start with developing a deep learning image classification model and eventually we will deploy the model to amazon web services with the help of few technologies provided in the AWS. Similarly, in today's world AWS provides a lot of facilities that a client wants while working with data. When we talk about data, the first thing that comes to our mind in the data storage. Sometimes it became a tedious and capacious process to store and save the data, we can get rid of this problem with the facilities provided by AWS, viz, Amazon S3. Again, when it comes to developing the model using deep learning or any other method we can use the facilities like AWS SageMaker, AWS EC2 instance, AWS Lambda function etc. So, AWS enables us to select the operating system, programming language, web application platform, database and other services that a client needs.

Description of the problem discussed

In this project we will address an image classification problem with the help of AWS. Essentially, this project has two parts. The first part is model development and the second part is deployment and testing the models. In the first part, we have developed a models which typically addresses image classification task. The dataset consists of two types of images-
i) Images with fire.
ii)Images without fire.
The next part is model deployment. In this part, we will check the model performance under AWS. The training of the model is performed in the visual studio code and then by forming a container image with the help of docker the necessary files are pushed in the AWS Elastic Container Registry. The other part of the project is deployed as a cloud web application using Amazon EC2 services. In this case, we have employed cloudwatch to visualize several metrics.

Design of the solution

For this task, we are using Amazon Web Service. Under AWS, there are multiple tools available. Our task is mainly focused on AWS Lambda, AWS Elastic Container Registry, S3 bucket and AWS EC2.

Deployment in AWS Lambda

Lambda is a serverless computer service that allow us to run code without provisioning or managing server. The code runs in parallel and processes each event individually, scaling with the size of the workload, from a few requests per day to hundreds of thousands of workloads. The below diagram explains the complete flowchart of the work:

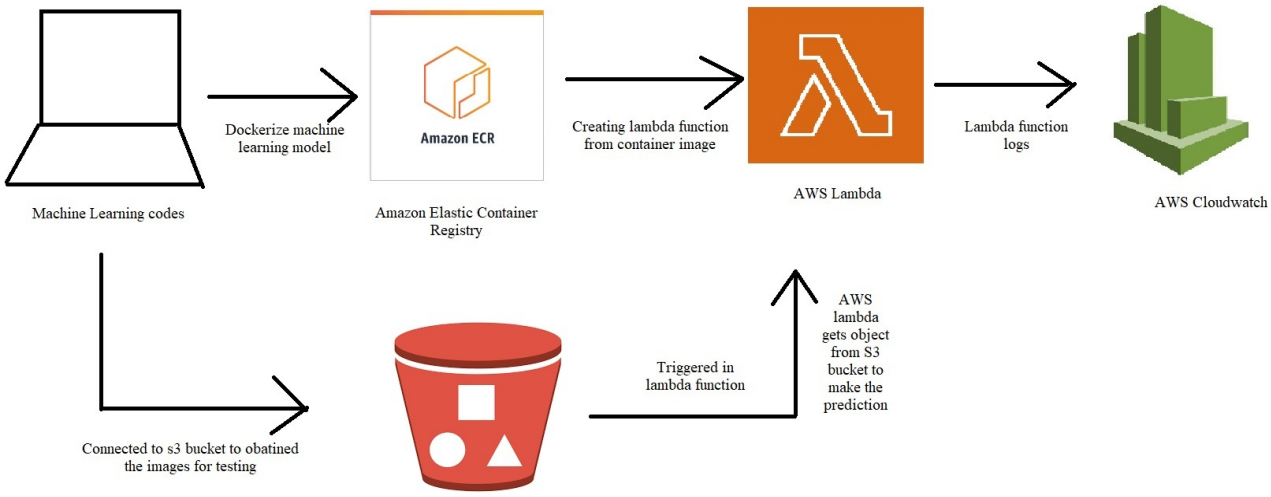


Figure 1: Flowchart

We have trained the model in the visual studio code and then package the code and dependencies as a container image using tools like Docker. After the model for inference is dockerized, we can upload the image to Amazon Elastic Container Registry (Amazon ECR). We can then create lambda function from the container imaged stored in Amazon ECR. The later part is to trigger the S3 bucket in the Lambda function. Once an image is uploaded in the S3 bucket, the lambda starts predicting whether the image is a fire or a non-fire image.

Deployment of the web application in AWS EC2

In this part, we deploy the web application using AWS EC2. For this we have taken help from the following tools and services.
1. Flask
2. WinSCP
3. Puttygen
4. Putty 5. AWS EC2
6. Load balancer
7. Security groups
8. Cloudwatch

The web application is available to the client with the public DNS provided by the EC2 and the port specified, which is, hypothetically, can be defined as, *Public DNS + port*. The model is provided using ubuntu virtual machine and for application purpose, there is no specific data provided because this is a web application for image classification. So it is upon the user which images he/she provides.

Description of the Implementation of the solution

In this project we have used forest fire dataset collected from the website. The dataset contains 1520 training and validation data in a file and another file contains 380 test image. In order to get a separate training and validation data we have splited the file using splitfolders package using python. So finally we have 1064 images in training data, 456 images in validation data. Let us also discuss breiefly about the cloud architectures used in the project:

Amazon Lambda:
AWS Lambda is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers. In lambda function we can build data-processing triggers for AWS services such as Amazon S3 and Amazon DynamoBD.

Amazon Simple Storage Service(S3):
A bucket is a container for object stored in Amazon S3. Amazon S3 is a service offered by Amazon Web Services that provides object storage through a web service interface.

Amazon CloudWatch: Amazon CloudWatch collects monitoring and operational data in the form of logs, metrics and events. CloudWatch gives us actionable insights that help you optimize application performance, manage resource utilization, and understand system-wide operational health. Alarms have been set in this to get notified when the system fails and also it will take automated actions or troubleshoot issues.

Amazon Elastic Compute Cloudù(EC2):
The application is hosted using the instances launched in the EC2. In order to host the application to the instance, we use Linux virtual machine with ubuntu 18.04 OS.

Security groups:
A security group controls the traffic that is allowed to reach and leave the resources that it is associated with the EC2 instance. Inbound rules controls the incoming traffics and the outbound rules controls the outgoing traffics for the instance.

Load balancer:
Load balancer works as a single point of contact for the client. The load balancer distributes incoming application traffics across multiple targets such as EC2 instance, in multiple application zones. This increases the availability of your application.

Deployment in AWS Lambda

We have used a pre-trained model Xception with weight imagenet for image classification. When an image is uploaded in Amazon S3 bucket, a Lambda function is invoked to detect the image and print it in Amazon CloudWatch logs. The detailed performance of this part is illustrated by the following image:

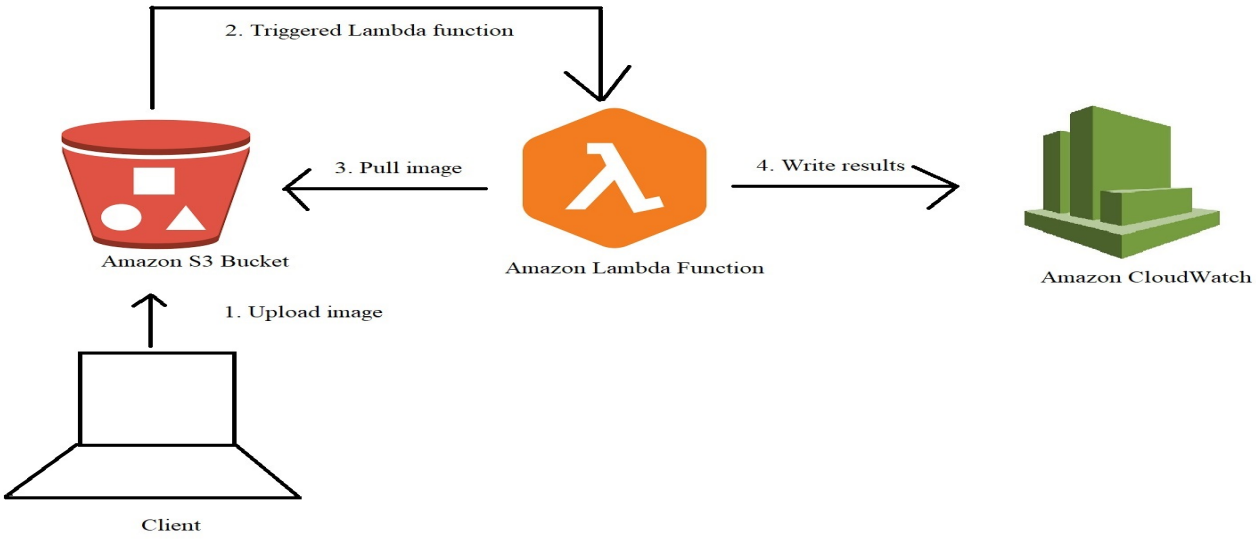


Figure 2: S3 to Lambda complete workflow

At first we have created the files required in the local machines, the instructions are given below:

1. On the local machine, create a folder *final_project*.
2. Create a *requirements.txt* file where all the required packages are mentioned.
3. A *train_model.ipynb* file is created where the model has been trained on our dataset.

4. Create a *lambda_function.py* script which contains the code for Lambda function.

5. Create a Dockerfile in the same directory which contain the list of all the necessary files to be uploaded as a container in the Amazon ECR.

The following image shows the content of the requirements.txt file to run the TensorFlow code for our use case:

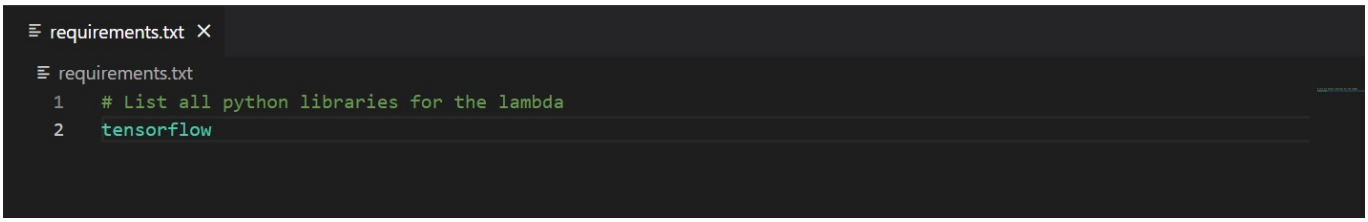


Figure 3: Content of requirements file

The python code prepared for the lambda function is placed in *lambda_function.py* file. The inference function in *lambda_function.py* needs a specific structure to be invoked by the Lambda runtime. After making all the files we have to build and push the container image to Amazon ECR.

Deployment of the web application in AWS EC2

As mentioned in the previous section, we have made use of the aforementioned 8 tools and services in order to host our web application using EC2.

- **Web development**

Web development is done fundamentally using python programming language and the Flask app. Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. In order to create the web pages, we have used HTML.

- **Deployment on EC2**

For deploying our application to EC2, the most essential tools were putty, puttygen and winscp. putty provides a tool named puttygen which converts the .pem private key to a .ppk private key and using this key, we connect our linux instances from windows operating system.
Winscp is used to transfer files between local computer to a remote computer. WinSCP is an open source free SFTP client, FTP client, WebDAV client, S3 client and SCP client for Windows.

Description of the deployment of your solution

In this project there are 2 part of deployment. The first one is the deployment using docker following some steps and the second part of the deployment is creating a server and launch it in EC2 instance.

Deployment in AWS Lambda

In order to deploy in AWS Lambda, at first we need to build and push the container image to Amazon ECR, then we will move to the next part which consists of formation of S3 bucket, creating a function in Lambda and creating roles in IAM. For building the container using docker with the following commands:

```
# Code to build the docker image with the tag name forest-fire-images
docker build -t fire-images .
# Register docker to AWS ECR by logging in to the ECR from VSCode
aws ecr get-login-password --region <REGION> | docker login --username AWS --password-stdin
    <AWS_ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com
# Create a ECR repository from VSCode
aws ecr create-repository --repository-name forest-fire-images --region <REGION>
# To get the tag associated with the container created
docker images
# Tag the image to match the repository name
docker tag <tag ID>
    <AWS_ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/forest-fire-images:fire-images
# Push the images to ECR
docker push <AWS_ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/forest-fire-images:fire-images
```

The AWS ECR image that is formed in after the execution of the above commands in shown below:

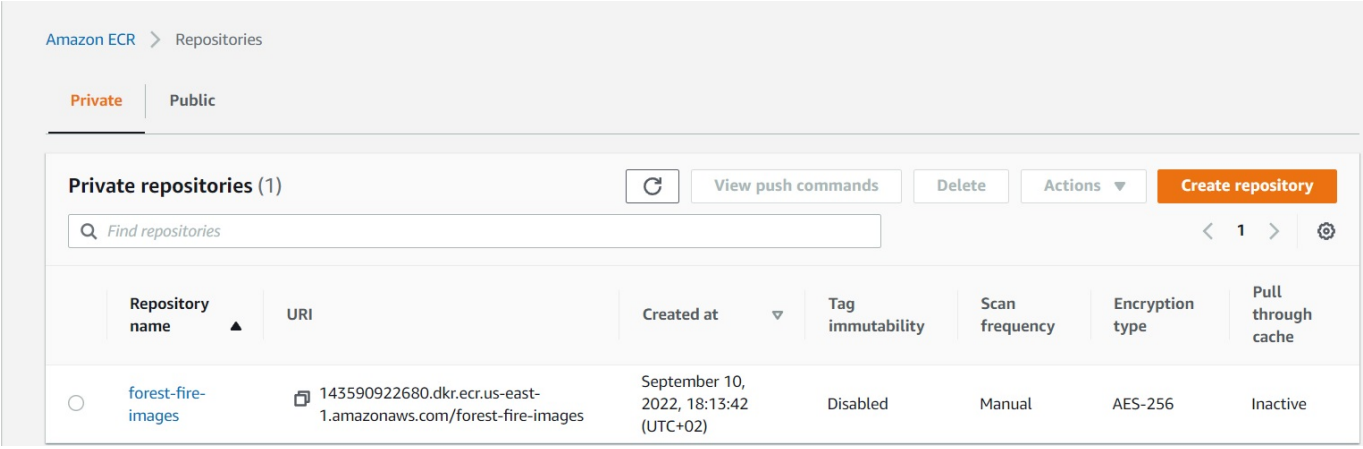


Figure 4: Repository created in AWS ECR

Next we will create a S3 bucket. The steps are given below:

1. On the AWS console management, choose S3.
2. The S3 bucket is given the name *forest-fire-database* formed in the AWS Region us-east-1.
3. Choose **create bucket**.

From the above steps we have created the bucket in S3 shown below:

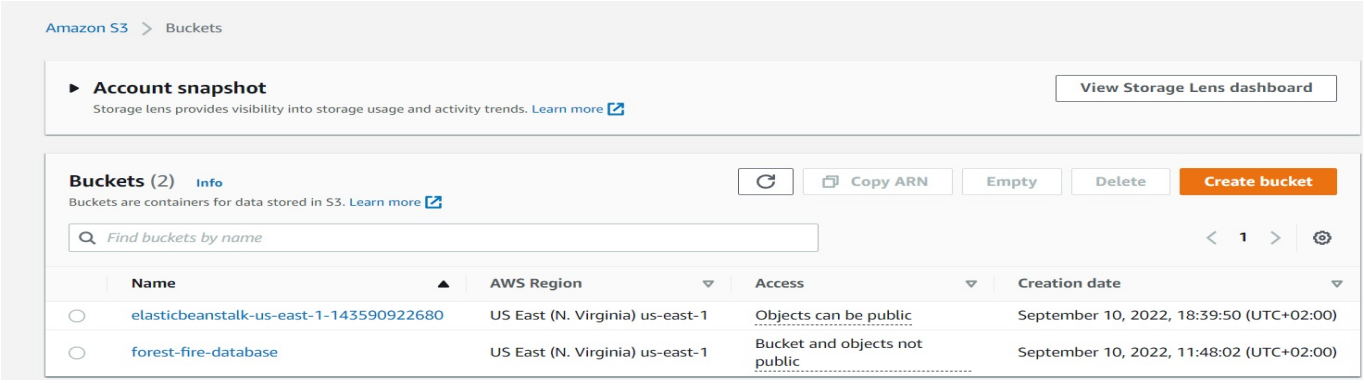


Figure 5: S3 bucket

Now our aim is to form a lambda function. We follow the below steps to create the lambda function with the TensorFlow code:

1. On the Amazon console management, select Lambda and choose Functions.
2. Next we have to select **Create function**
3. Select **Container image** to form the container
4. In the **Function name** we can provide name such as *classification-forest-fire*.
5. For **Container image URL**, we have to enter the latest container created earlier named *forest-fire-images*.
6. In order to select the latest image, we have to choose **Browse images**.
7. Click **Create function** to initiate the creation of the function.

The below image shows the structure of the AWS Lambda function formed:

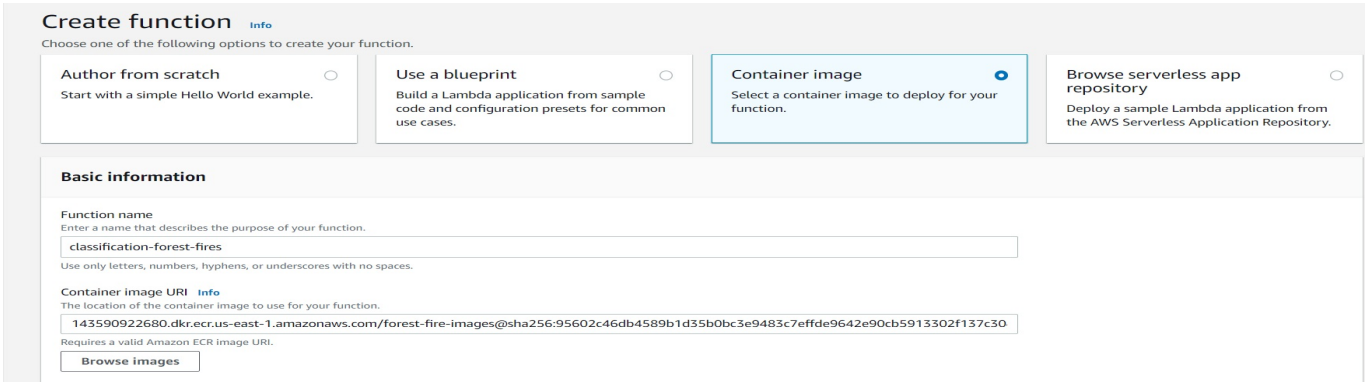


Figure 6: AWS Lambda function

The image shows the selection of the container used in the AWS Lambda:

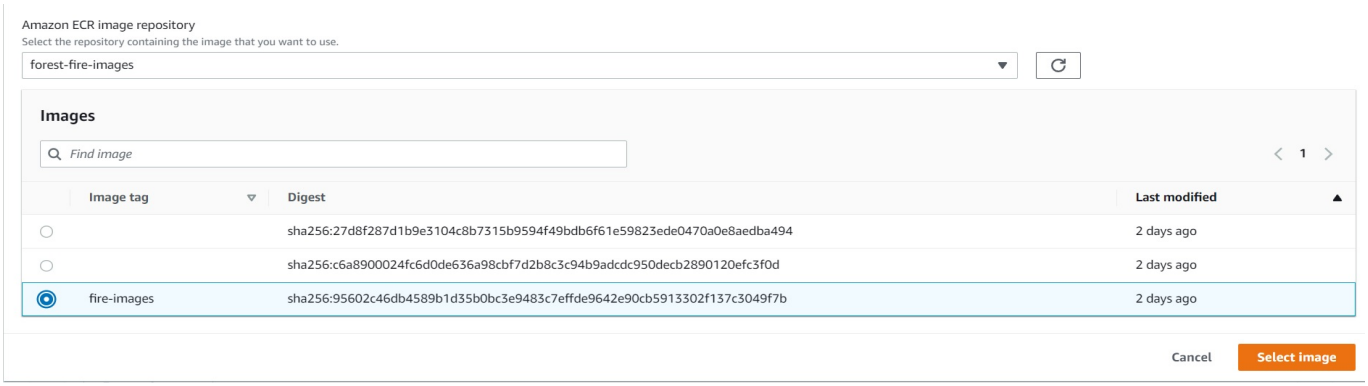


Figure 7: AWS ECR Image Repository

To improve the runtime of Lambda, we can change the memory to 3000 MB and the timeout to 5 min in the general configuration. Next we have to create the s3 bucket to our lambda function, that is, by triggering the Lambda function. The step to be followed are:

1. On the Amazon console management, choose the Lambda function.
2. Choose **Add Trigger** from the Lambda function.
3. In the trigger configuration, we have to select **S3**.
4. Then we have to browse the bucket that have been create earlier in the S3 bucket.

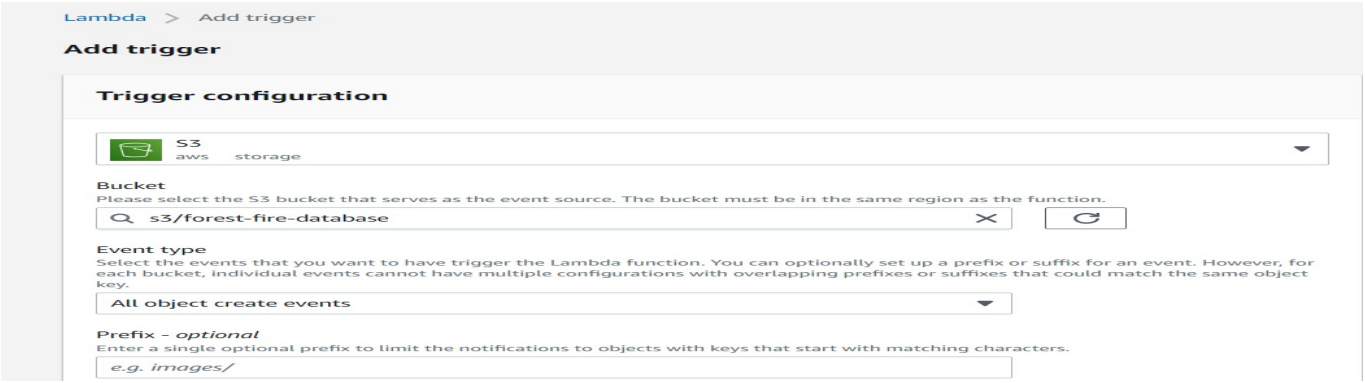


Figure 8: S3 Trigger

The picture below shows the structure of the triggering in the Lambda function: After the trigger is added to the lambda function, we need to allow the Lambda function to connect to the S3 bucket by setting the appropriate AWS Identity and Access Management (IAM) rights for its execution role. Also we have selected the Roles for AWS CloudWatch in IAM. To do so we have to follow the below steps:

1. On the Amazon console management, we have to select **IAM**.
2. On the left panel under **Access Management** select **Roles**.
3. We have to select the specific function created from the list of the roles.
4. In the **Permissions**, we have to select **Add permissions**.
5. Go to **Attach policies** and select *AmazonS3ReadOnlyAccess* for the S3 bucket and *AWSLambdaBasicExecutionRole* for the CloudWatch.
6. At the end, click on **Attach policies**.

The below picture shows the roles attached in the IAM:

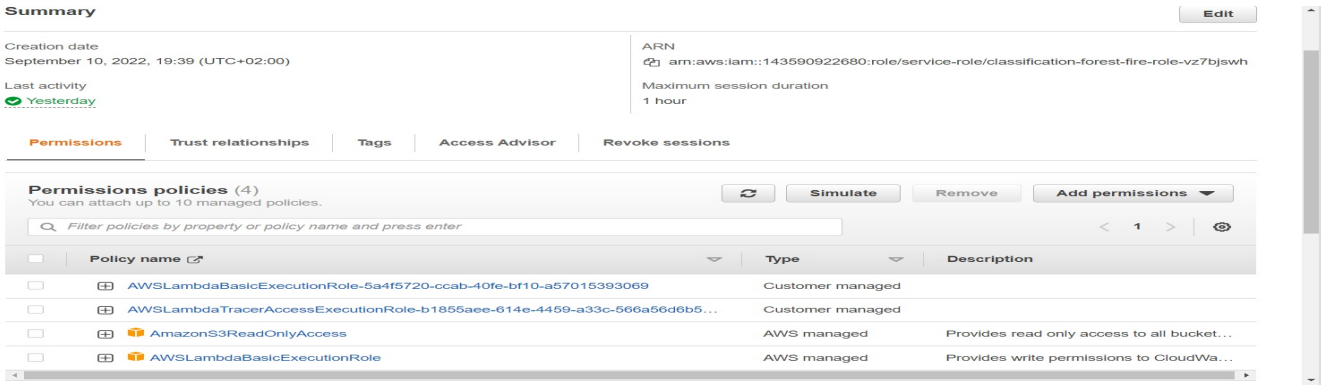


Figure 9: IAM Roles

Deployment in EC2 instance

For the deployment process, we first create an instance in EC2. For this purpose, the EBS storage has been set to 8GB. We have used ubuntu 18.04 as the Linux virtual machine. In order to conduct a successful deployment from the ubuntu to the EC2 instance, we have used two other tools, viz. Putty and Winscp. Putty is used to save the private key and to launch the virtual machine. When we launch the instance, we save the private key as the .PEM file. However, this .PEM file is converted to a .ppk file using puttygen. Finally we use the public dns address as the host and the save ppk file as the private key to launch the ubuntu VM. In parallel we use winscp to transfer file between the local machine and the remote machine. Figure 10 depicts how the EC2 instance looks like.

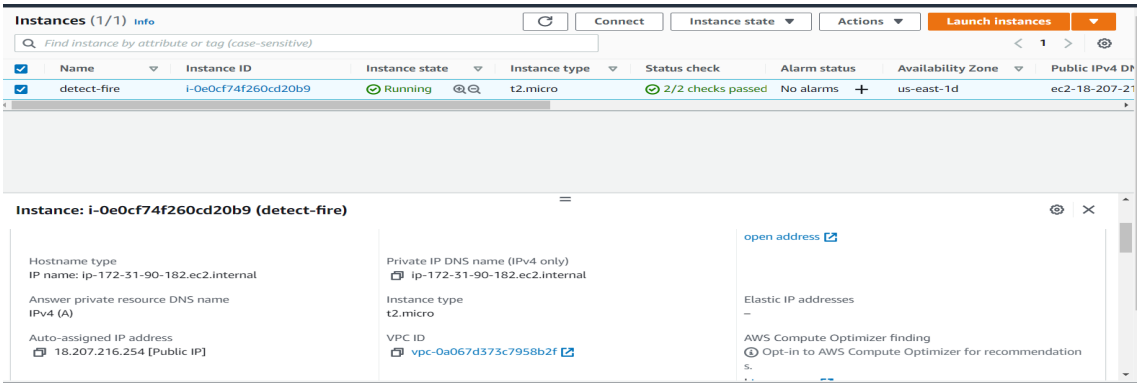


Figure 10: EC2 instance

The virtual machine contains all the necessary files for the process. for our purpose, the files are as follows:

(i) application.py- Main code using the Flask app for the web application.
(ii) Template - A folder contains the index.html file.
(iii) static: Folder contains the necessary images.
(iv) requirements.txt - A text file contains the necessary packages.
(v) Model - The trained model is stored as a .h5 file.

Figure 11 depicts how the ubuntu is synced with the remote computer.
Once the system is ready, we perform the following codes.

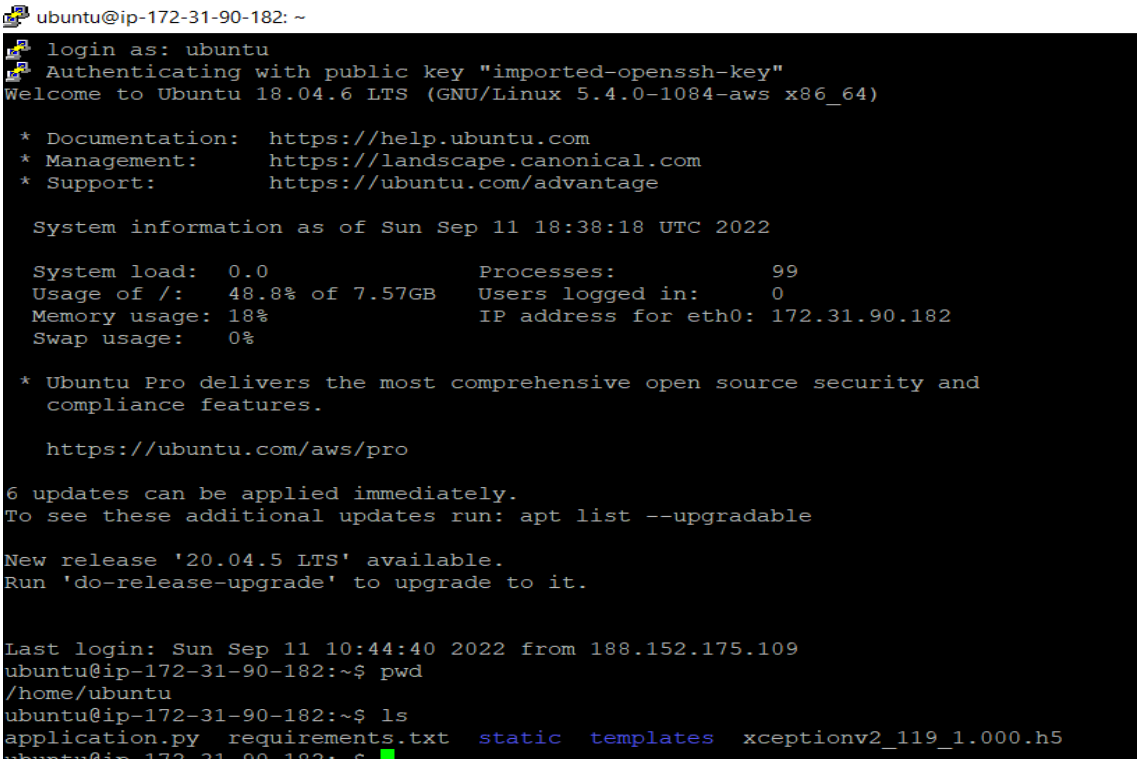


Figure 11: Ubuntu interface

```
pwd # check the drive
ls  # check if the files are correct
sudo apt-get update
sudo apt-get install python3-pip # install python
sudo -H pip3 install --upgrade pip # update the pip
pip3 install -r requirements.txt # install the packages
python3 application.py # run the python file
```

Once, the application runs, we go to the instance again, we copy the public dns and run it with the port. The address of the application is-
http://ec2-18-207-216-254.compute-1.amazonaws.com:8080/

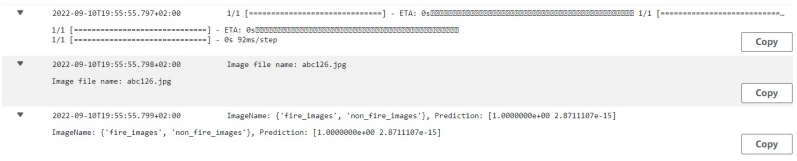
Test/validation design

Until now we have configured all the necessary services to test our AWS Lambda function and make EC2 prepare to launch the website. The next step will be to obtain the test/validation design.

Testing in AWS Lambda

In order to start testing, we have to upload an image to the created S3 bucket by opening the bucket in the AWS console and clicking upload. After uploading the image, within a few seconds we can see the result of the prediction in the CloudWatch logs. After uploading the image to the S3 bucket we will get the predicted image class with the probability. The correct class will have the higher probability than the other class. The image and the corresponding predictions are provided below:

(a) Fire images



(b) Prediction

Figure 12: Fire prediction

(a) Non-fire images

(b) Prediction

Figure 13: Non-fire prediction

Testing with EC2

The interface of the application looks as following images:

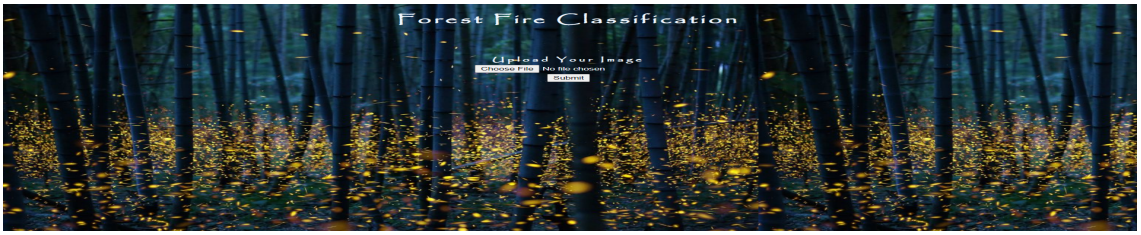


Figure 14: Web interface for forest fire detection

Figure 15: Forest Fire classification

Figure 16: Forest non-fire classification

Experimental results

In this part of the project we will discuss about the metrics obtained from the testing in both AWS Lambda and AWS EC2.

AWS Lambda

The metrics obtained from AWS Lambda are invocation, Duration, Error count and success rate and concurrent execution.

Invocation

In the invocation metrics, it is shown that the number of times a function is invoked. In the below picture the peak number of invocation is noted as 236. Basically, it monitors the values as a general barometer for the amount of traffic flowing through the serverless application.

Duration

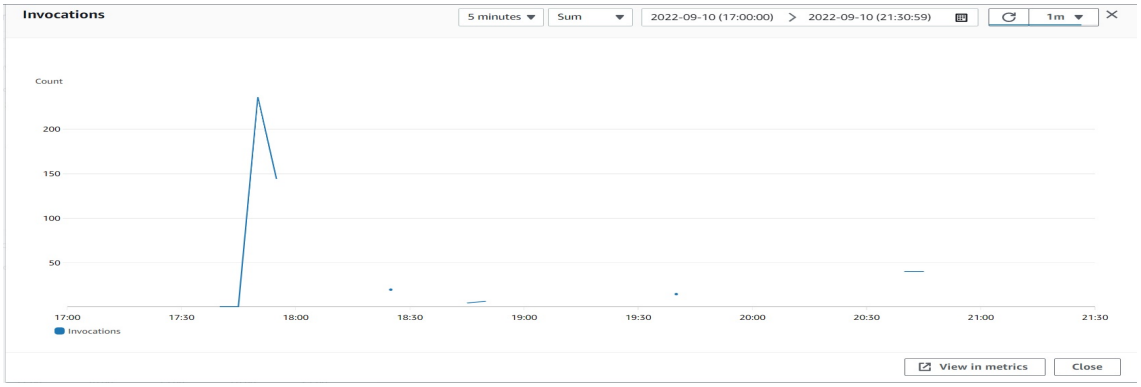


Figure 17: Invocation

In the duration it shows the amount of time taken for a Lambda function.

Error rate and success rate

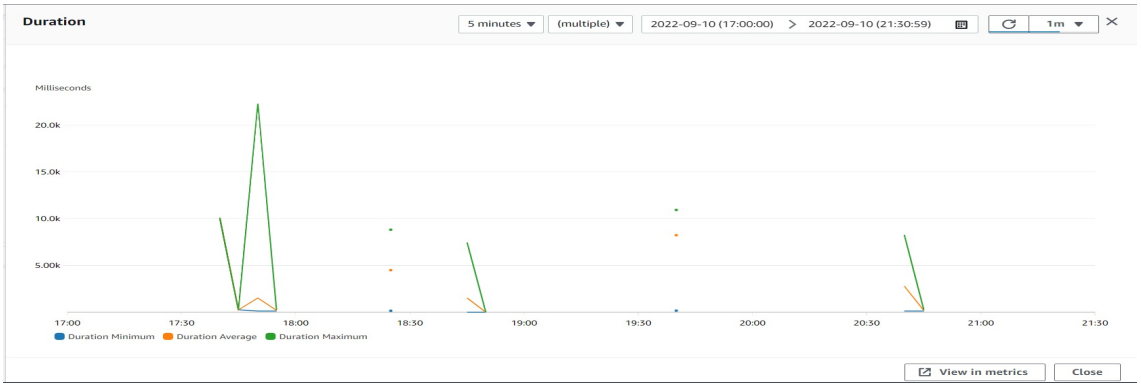


Figure 18: Duration

It monitors the number of errors thrown by a function. It can be used with the invocations metric to calculate the total percentage of errors.

Concurrent Executions

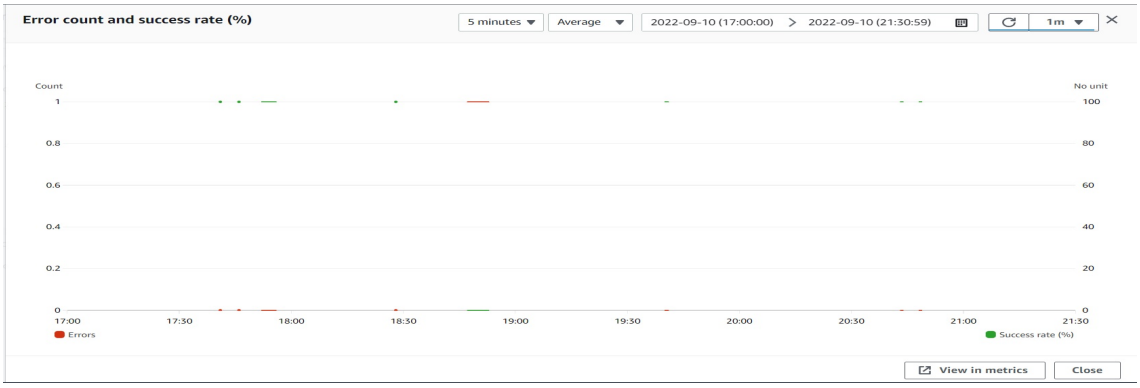


Figure 19: Error and success rate

It monitors the value to ensure that the function are not running close to the total concurrency limit for the AWS account.

CloudWatch Logs

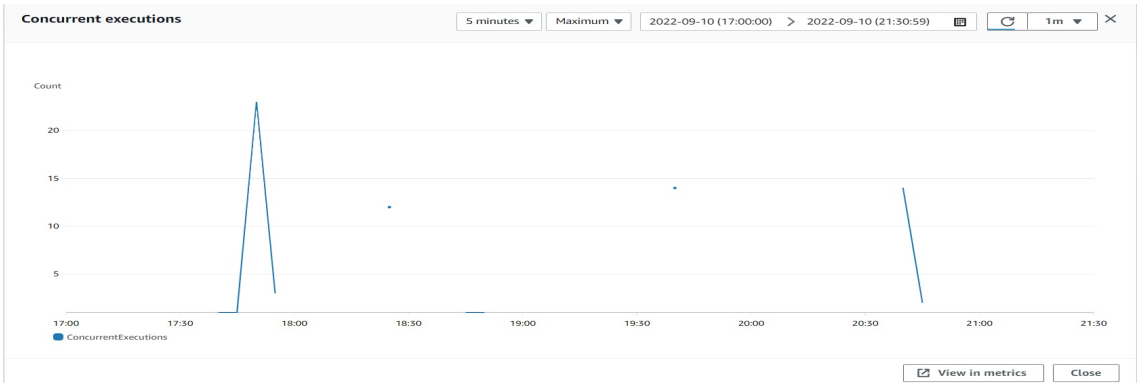


Figure 20: Concurrent Execution

Lambda automatically streams details about each function invocations, along with logs, and other output from the function's code to CloudWatch Logs.

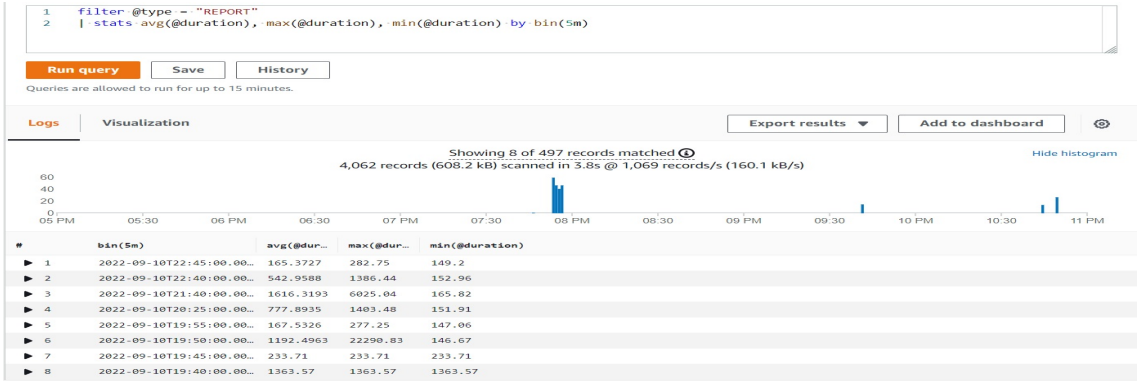


Figure 21: Log insights

AWS EC2

When we run the application, we have observed the CPU utilization for 24 hours. 22 shows the fluctuations over the hours. The percentage of allocated EC2 compute units that are currently in use on the instance. This metric identifies the processing power required to run an application on a selected instance. In the following figure, we show the changes in the network in, network out and

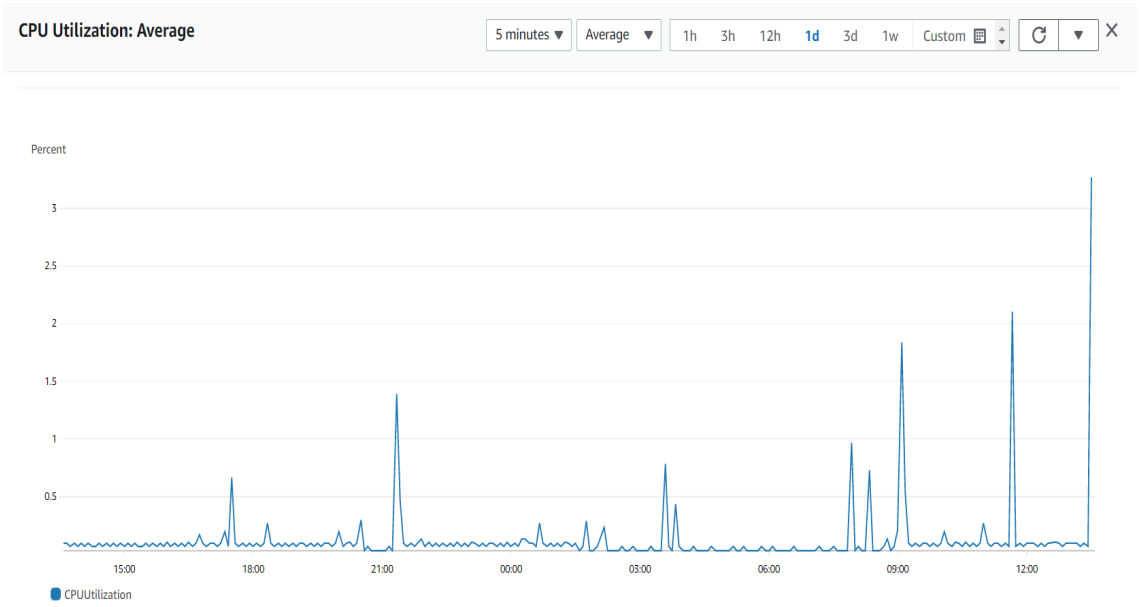
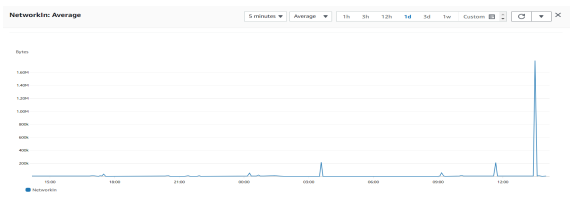
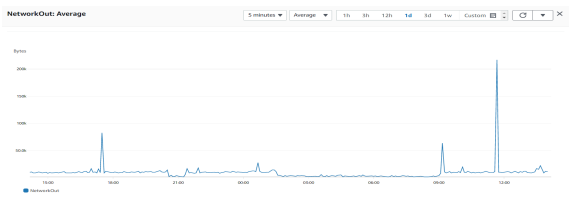


Figure 22: CPU Utilization in EC2

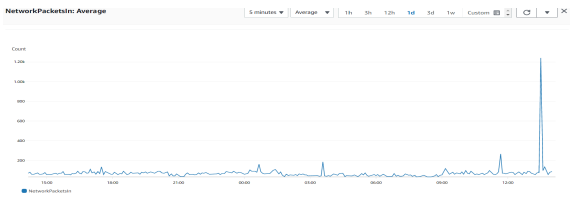
network packein and network packetout.



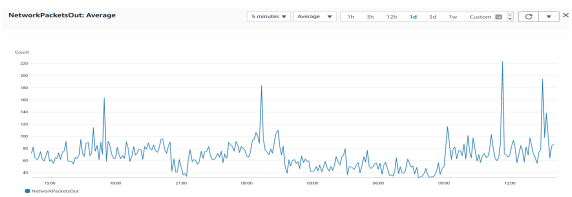
(a) Network In



(b) Network Out



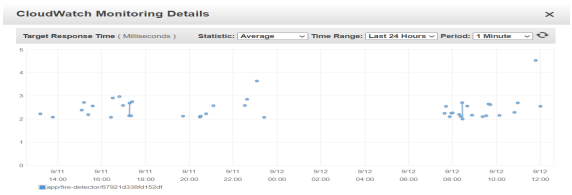
(c) Network packet in



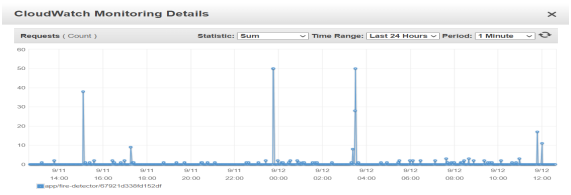
(d) Network packet out

Figure 23: EC2 metric evaluation

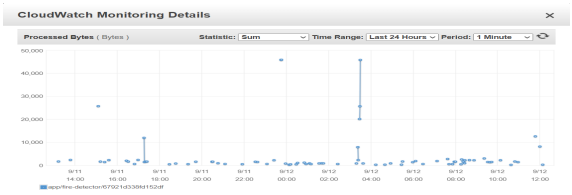
The following results were obtained after the implementation of the load balancer.The results of the load balancer is that the requests have been uniformely distributed over the resources and then the resources were limited when there is no load.



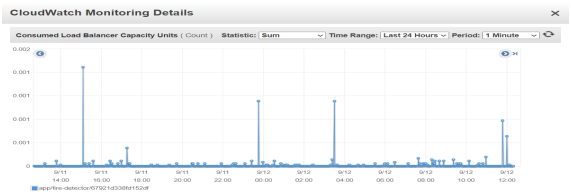
(a) Target requests



(b) Number of requests



(c) Processed bytes



(d) Consumed bytes

Figure 24: Load balancer metric evaluation

Conclusion

In this project we have considered the forest fire data and it was an image classification task. The first part of the project is to deploy the model as a container image in the AWS Lambda and the next part of the project is to deploy the model as an EC2 instance in order to form a website. After the deployment we have tested in both the cases with a new Test dataset that has been formed previously. We have also shown some metrics involved with the function and the instance created such as invocation, duration, error rate and success rate and concurrent execution for the lambda function and for the EC2 instance CPU utilization, Network and Network package and Load balancer.

Reference

1. Using container images to run TensorFlow models in AWS Lambda
2. AWS Documentation

Website:http://ec2-18-207-216-254.compute-1.amazonaws.com:8080/