



# Lab 4 Report

50.046 CCIoT

SUTD 2022 ISTD

*Group 3*

Han Xing Yi	( 1004330 )
Ang Song Gee	( 1004589 )
James Raphael Tiovalen	( 1004555 )
Velusamy Sathiakumar Ragul Balaji	( 1004101 )

# Table of Contents

<b>Prerequisite: Setup SageMaker Studio</b>	<b>3</b>
<b>Task 1: Streaming Data Generation</b>	<b>4</b>
<b>Task 2: Streaming Data Processing</b>	<b>5</b>
<b>Task 3: ML Model Training</b>	<b>7</b>
<b>Task 4: Storing the Prediction</b>	<b>9</b>
<b>Answers to Reflection Questions</b>	<b>11</b>

## Prerequisite: Setup SageMaker Studio

- Create a New IAM Role
- Provisioned SageMaker Domain
- Created Jupyter Notebook
- Downloaded the `nyc-taxi.csv` dataset

The screenshot shows the Amazon SageMaker Studio interface. On the left, there's a file browser window titled 'Amazon SageMaker Studio' showing a directory structure for 'amazon-sagemaker-immersion-day'. In the center, a Jupyter notebook tab is open with the name 'Group3Lab4.ipynb'. Below the tabs, a terminal window displays the command `git clone https://github.com/aws-samples/amazon-sagemaker-immersion-day.git` being run. The terminal output shows the progress of cloning the repository from GitHub, indicating it is 100% complete.

```
[2]: git clone https://github.com/aws-samples/amazon-sagemaker-immersion-day.git
Cloning into 'amazon-sagemaker-immersion-day'...
remote: Enumerating objects: 521, done.
remote: Counting objects: 100% (404/404), done.
remote: Compressing objects: 100% (301/301), done.
remote: Total 521 (delta 150), reused 302 (delta 87), pack-reused 117
Receiving objects: 100% (521/521), 33.28 MiB | 10.52 MiB/s, done.
Resolving deltas: 100% (200/200), done.
Checking out files: 100% (96/96), done.
```

The tasks below are executed in the following order as specified in the [workshop's instructions](#):

1. Task 3: ML Model Training
2. Task 2: Streaming Data Processing
3. Task 1: Streaming Data Generation
4. Task 4: Storing the Prediction

## Task 1: Streaming Data Generation

- Create Cloud9 Instance and Configure `data.py` Script

```

import datetime
import json
import random
import boto3
STREAM_NAME = "ExampleInputStream"
my_session = boto3.session.Session()
my_region = my_session.region_name
def get_data():
    return random.choice([
        "1,1.44,-73.967393,40.756458,1,-73.98366,40.745642,0.5,0.5,0.0,0.0,7.5,300.0,0.0,1.0,0.0",
        "-1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.0000000000003,1.0,0.0",
        "-2,1.5,-73.98050400000001,40.783272,1,-73.963669,40.794529,0.5,0.5,0.0,0.0,7.5,360.0,1.0,0.0",
        "-2,13.6,-73.98812,40.748923,1,-73.90385500000001,40.887425,0.5,0.5,0.0,0.0,38.5,1320.0,1.0,0.0"
    ])
def generate(stream_name, kinesis_client):
    while True:
        data = get_data()
        print(data)
        kinesis_client.put_record(
            StreamName=stream_name,
            Data=json.dumps(data),
            PartitionKey="partitionkey")
if __name__ == '__main__':
    generate(STREAM_NAME, boto3.client('kinesis', region_name=my_region))

```

The terminal window shows the script being run and the AWS CLI installing dependencies:

```

[sudo - ip-172-31-13-96 ~] | 79 KB 11.1 MB/s
Requirement already satisfied: urllib3<1.27,>=1.25.4 in /usr/local/lib/python3.7/site-packages (from botocore<1.25.0,>=1.24.32->boto3) (1.26.9)
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /usr/local/lib/python3.7/site-packages (from botocore<1.25.0,>=1.24.32->boto3) (2.8.2)
Requirement already satisfied: six<1.5 in /usr/local/lib/python3.7/site-packages (from python-dateutil<3.0.0,>=2.1->botocore<1.25.0,>=1.24.32->boto3) (1.16.0)
Installing collected packages: botocore, s3transfer, boto3
  Attempting uninstall: botocore
    Found existing installation: botocore 1.24.26
      Uninstalling botocore-1.24.26:
        Successfully uninstalled botocore-1.24.26
Successfully installed boto3-1.21.52 botocore-1.24.32 s3transfer-0.5.2
[ec2-user:~/environment]$ 

```

- Run `data.py` Script for Streaming Data Injection

```

python3 - "ip-172-31-13-96 ~" | Immediate
2,1.5,-73.98050400000001,40.783272,1,-73.963669,40.794529,0.5,0.5,0.0,0.0,7.5,360.0,1.0,0.0
2,13.6,-73.98812,40.748923,1,-73.90385500000001,40.887425,0.5,0.5,0.0,0.0,38.5,1320.0,1.0,0.0
2,13.6,-73.98812,40.748923,1,-73.90385500000001,40.887425,0.5,0.5,0.0,0.0,38.5,1320.0,1.0,0.0
1,1.44,-73.967393,40.756458,1,-73.98366,40.745642,0.5,0.5,0.0,0.0,7.5,300.0,0.0,1.0
1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.0000000000003,1.0,0.0
1,1.44,-73.967393,40.756458,1,-73.98366,40.745642,0.5,0.5,0.0,0.0,7.5,300.0,0.0,1.0
2,1.5,-73.98050400000001,40.783272,1,-73.963669,40.794529,0.5,0.5,0.0,0.0,7.5,360.0,1.0,0.0
1,1.44,-73.967393,40.756458,1,-73.98366,40.745642,0.5,0.5,0.0,0.0,7.5,300.0,0.0,1.0
1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.0000000000003,1.0,0.0
1,1.44,-73.967393,40.756458,1,-73.98366,40.745642,0.5,0.5,0.0,0.0,7.5,300.0,0.0,1.0
2,1.5,-73.98050400000001,40.783272,1,-73.963669,40.794529,0.5,0.5,0.0,0.0,7.5,360.0,1.0,0.0
2,13.6,-73.98812,40.748923,1,-73.90385500000001,40.887425,0.5,0.5,0.0,0.0,38.5,1320.0,1.0,0.0

```

## Task 2: Streaming Data Processing

### ✓ Create Kinesis Data Stream

The screenshot shows the AWS Kinesis Data Stream creation page. A green success message at the top states "Data stream ExampleInputStream successfully created." The main section displays "ExampleInputStream" details:

Status	Capacity mode	ARN	Creation time
Active	Provisioned	arn:aws:kinesisap-southeast-1:317718598459:stream/ExampleInputStream	April 04, 2022, 11:26 GMT+8
Data retention period		1 day	

Below the summary, tabs for Applications, Monitoring, Configuration, and Enhanced fan-out (0) are visible. The Applications tab is selected.

### ✓ Create and Configure Kinesis Data Analytics App

The screenshot shows the AWS Kinesis Data Analytics application creation page. A green success message at the top states "Successfully created application TaxifareKDA." Below it, another message says "Application TaxifareKDA has been successfully updated." The main section displays "TaxifareKDA" details:

Application details		
Version ID: 2 (Latest) All versions	Configure	Run
Open Apache Flink dashboard		
Actions ▾		
▶ How it works: Apache Flink application Info		
Status	ARN	Runtime
⌚ Ready	arn:aws:kinesisanalytics:ap-southeast-1:317718598459:application/TaxifareKDA	Apache Flink 1.11
IAM role	Last updated	Description
kinesis-analytics-TaxifareKDA-ap-southeast-1	April 04, 2022, 11:35 GMT+8	TaxifareKDA Apache Flink App
Creation time		
April 04, 2022, 11:30 GMT+8		

## ✓ Add IAM Policies to the KDA Role

The screenshot shows the AWS Identity and Access Management (IAM) service. On the left, the navigation pane is open with 'Identity and Access Management (IAM)' selected. Under 'Access management', 'Roles' is selected. In the main content area, the role 'kinesis-analytics-TaxifareKDA-ap-southeast-1' is displayed. The 'Permissions' tab is active, showing a table of permissions. Two policies are listed: 'kinesis-analytics-service-TaxifareKDA-ap-southeast-1' (Customer managed) and 'KDA-S3-KDA-Inline-Policy' (Customer inline). The 'KDA-S3-KDA-Inline-Policy' row is highlighted with a blue background.

## ✓ Run the KDA App and View Apache Flink Dashboard

The screenshot shows the Apache Flink Dashboard. The left sidebar has sections for 'Overview', 'Jobs' (with 'Running Jobs' and 'Completed Jobs'), and 'Task Managers'. The main area shows a job named 'Async IO Example' with status 'RUNNING'. Below it, the job ID is '001767c7ac6548dedb786d086155076a', start time '2022-04-04 11:43:35', and duration '2m 48s'. The 'Overview' tab is selected. The job graph shows three components: 'Source: flink\_kinesis\_consumer\_01 -> Map' (Parallelism: 1), 'async wait operator' (Parallelism: 1), and 'Sink: Unnamed' (Parallelism: 1). Arrows indicate the flow from source to operator and operator to sink. At the bottom, a table provides detailed metrics for each task:

Name	Status	Bytes Received	Records Received	Bytes Sent	Records Sent	Tasks
Source: flink_kinesis_consumer_01 -> Map	RUNNING	0 B	0	0 B	0	1
async wait operator	RUNNING	60 B	0	0 B	0	1

## Task 3: ML Model Training

### ✓ Train and Deploy the ML Model

```

#Metrics {"StartTime": 1649041869, "EndTime": 1649041882, "Dimensions": {"Algorithm": "Linear Learner", "Host": "algo-1", "Operation": "training"}, "Metrics": {"initialize_time": {"sum": 363.1591796875, "count": 1, "min": 363.1591796875, "max": 363.1591796875}, "epochs": {"sum": 11.260271072387695, "count": 17, "min": 0.2284049877929688, "max": 1.6498565673828125}, "update_time": {"sum": 12537.19.2075252532959, "count": 1, "min": 19.2075252532959, "max": 1023.158788610303}, "finalize_time": {"sum": 197.88742065429688, "count": 1, "min": 197.88742065429688, "max": 31.044960021972656}, "setup_time": {"sum": 1972656, "count": 1, "min": 31.044960021972656, "max": 777544021606}}, "totaltime": {"sum": 13486.777544021606, "count": 1, "min": 13486.777544021606, "max": 13486.777544021606}}
[04/04/2022 03:11:22 INFO 140059338196800 integration.py:636] worker closed

2022-04-04 03:11:49 Completed - Training job completed
Training seconds: 112
Billable seconds: 112
CPU times: user 645 ms, sys: 80.4 ms, total: 726 ms
Wall time: 4min 12s

Set up hosting for the model 


Once the training is done, we can deploy the trained model as an Amazon SageMaker real-time hosted endpoint. This will allow us to make predictions (or inference) from the model. Note that we don't have to host on the same instance (or type of instance) that we used to train. Training is a prolonged and compute heavy job that requires a different type of instance and memory requirements than hosting typically do not. We can choose any type of instance we want to host the model. In our case we chose the ml.m4.xlarge instance to train, but we choose to host the model on the less expensive cpu instance, ml.c4.xlarge. The endpoint deployment can be accomplished as follows:


```

```

[18]: %time
    #Creating the endpoint out of the trained model
    linear_predictor = linear.deploy(initial_instance_count=1, instance_type="ml.c4.xlarge")
    print(f"\nCreated endpoint: {linear_predictor.endpoint_name}\n")
    -----
    created endpoint: linear-learner-2022-04-04-03-12-32-670
    CPU times: user 125 ms, sys: 2.13 ms, total: 127 ms
    Wall time: 3min 1s

```

Copy the endpoint name of the deployed model from above and save it for later

Inference

Created endpoint: `linear-learner-2022-04-04-03-12-32-670`

### ✓ Run a Simple Inference on Our Endpoint

```

payload: 1,1.6,-73.973587,40.754896,1,-73.998192,40.760822,0.5,0.5,0.0,0.0,8.5,420.0,1.0,0.0
<class 'str'>
Result: {'predictions': [{'score': 7.277685165405273}]}
Actual fare: 7.5
Prediction: 7.28
Accuracy: 97.07
CPU times: user 28.5 ms, sys: 3.38 ms, total: 31.9 ms
Wall time: 142 ms

```

✓ Create Lambda Function and API Gateway

The screenshot shows the AWS CloudFormation Outputs page for the 'bryxjrsqAPIGW' stack. The stack was created on 2022-04-04 11:22:41 UTC+0800 and is in the 'CREATE\_COMPLETE' state. The 'Outputs' tab is selected, displaying two outputs:

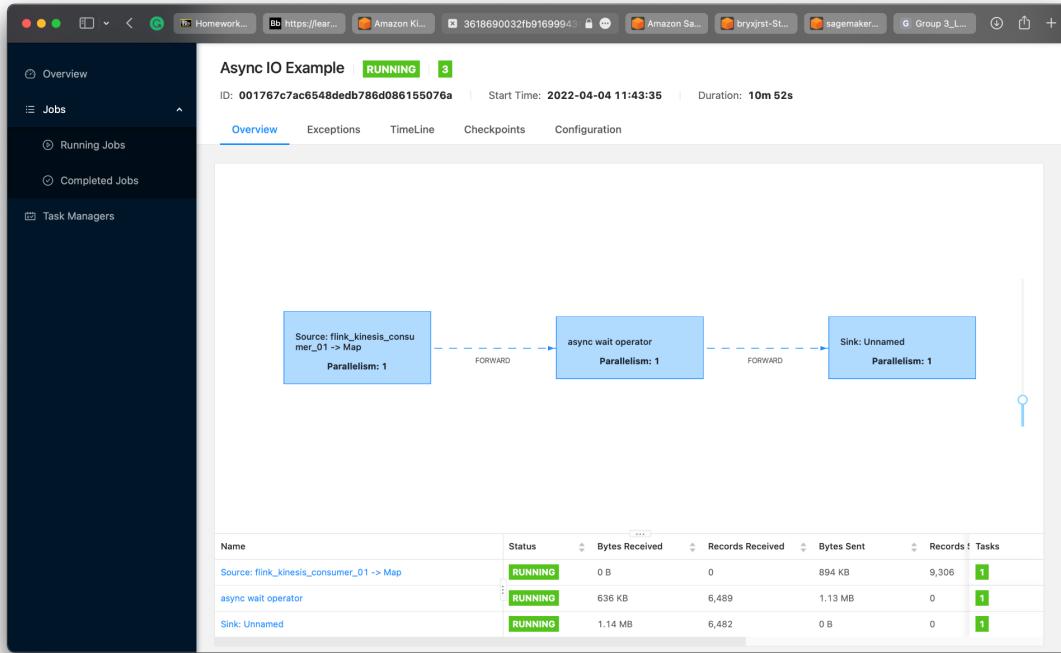
Key	Value	Description	Export name
apiGatewayInvokeURL	<a href="https://9gpihsr06e.execute-api.ap-southeast-1.amazonaws.com/prod">https://9gpihsr06e.execute-api.ap-southeast-1.amazonaws.com/prod</a>	-	-
lambdaArn	arn:aws:lambda:ap-southeast-1:317718598459:function:my-function-SMInvokeEndpoint	-	-

Invoke URL:

<https://9gpihsr06e.execute-api.ap-southeast-1.amazonaws.com/prod>

## Task 4: Storing the Prediction

✓ 3 Jobs Running on Apache Flink Dashboard



✓ Prediction Data Generated Stored in S3 Bucket

 Download Results File and View Prediction Results

## Answers to Reflection Questions

1. What's the job of AWS Kinesis Data Stream in this pipeline? Why can't we link the Python data generator script directly to the AWS Kinesis Data Analytics (KDA) engine?

The role of AWS Kinesis Data Stream (KDS) is to collect and process all incoming streams of data arriving from the python data generator script `data.py` running in the Cloud9 IDE. The python script does this by placing records into the created KDS stream.

AWS KDS is required in this pipeline because the AWS KDA engine does not collect input data. It reads data processed by KDS, before transforming and analysing the data in real-time using Apache Flink.

In other words, the KDA engine relies on the KDS as a data source to continuously intake and process data records from external input sources. Then, the processed data records in KDS can be retrieved and consumed by the KDA engine for data analytics purposes.

2. What's the relation between AWS Kinesis Data Analytics (KDA) and Apache Flink? What are the roles played by the AWS KDA and SageMaker respectively in this pipeline?

Apache Flink is a framework to build streaming applications. The relationship between KDA and Apache Flink is that KDA is a fully managed and serverless AWS service for Apache Flink.

KDA provides the underlying infrastructure and handles core capabilities such as provisioning compute resources, parallel computation, automatic scaling, and application backups (from the docs: [What Is Amazon Kinesis Data Analytics for Apache Flink?](#)).

KDA's role is to read data from KDS and asynchronously invoke the SageMaker endpoint for all incoming streaming data using the API Gateway. Upon receiving the inference results, KDA will then store the results in an S3 bucket. In the background, KDA will also provide real-time data

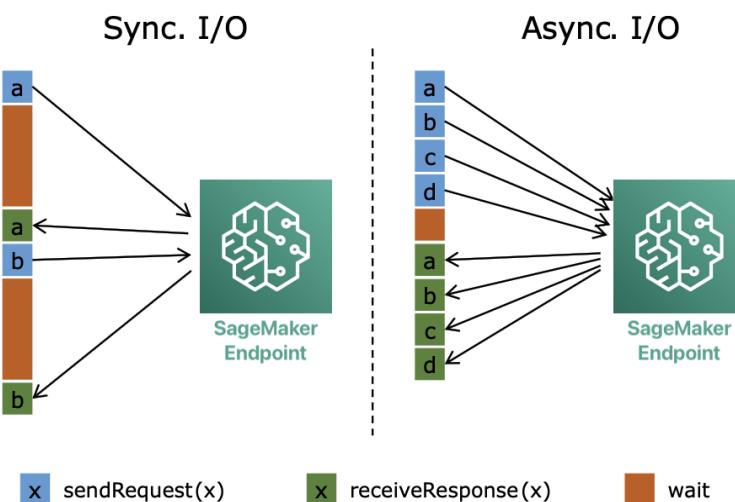
analytics on the data volume and number of records received and sent at each task.

SageMaker's role is to provide a cloud environment for training a machine learning model to predict taxi fares. After the training, SageMaker helps to deploy the model on a SageMaker real-time hosted endpoint. We can then make real-time inferences about predicted taxi fares by querying this SageMaker endpoint that can be invoked using a Lambda function.

3. Why is an *Asynchronous I/O* operator used between the AWS Kinesis Data Analytics and API Gateway in this pipeline?

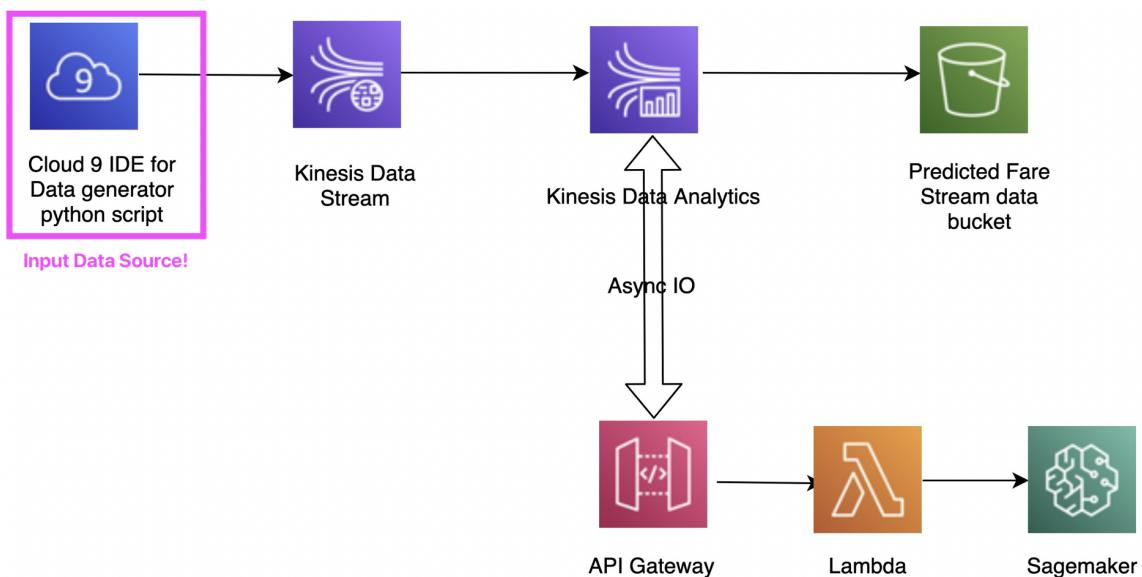
The Async I/O operator is used because it is non-blocking and allows for interactivity with high scalability. By using Async I/O, a single parallel function instance can handle many requests concurrently and receive the responses concurrently (from [Async I/O | Apache Flink](#)). This leads to a much higher streaming throughput for our pipeline. (See image below adapted from [Async I/O | Apache Flink](#)).

Interacting with the SageMaker endpoint to obtain the predicted taxi fares takes time. The Async I/O operator helps to ensure that the communication delay with the SageMaker endpoint does not cause the application to be blocked. This prevents the waste of precious compute resources, and allows incoming streaming data to be handled with low latency.



4. This demo is using simulated data created by a Python script. What changes in the pipeline does it need to ingest real-world data in a production environment? (You don't need to implement that.)

To ingest real-world data in a production environment, we have to change the input data source to Kinesis Data Stream. To do so, the customer application can either use the AWS SDK for Python (Boto3) to put data records into the Kinesis Data Stream, or other AWS SDKs available for other programming languages. We need to ensure that the KDS can handle multiple incoming streams of data from different sources.



Next, since we might not be able to reliably predict the throughput required, we would need to configure KDS's capacity mode to be "On-demand". This ensures that the data stream scales automatically with the amount of data received in a production environment.

**Data stream capacity** [Info](#)

**Capacity mode**

**On-demand**  
Use this mode when your data stream's throughput requirements are unpredictable and variable. With on-demand mode, your data stream's capacity scales automatically.

**Provisioned**  
Use provisioned mode when you can reliably estimate throughput requirements of your data stream. With provisioned mode, your data stream's capacity is fixed.

**Provisioned shards**  
The total capacity of a stream is the sum of the capacities of its shards. Enter number of provisioned shards to see total data stream capacity.

1 [Shard estimator](#)

Minimum: 1, Maximum available: 199, Account quota limit: 200. [Request shard quota increase](#)

Finally, we would also need to ensure that the KDA engine can handle the data load in a production environment. To do so, we would need to change the application settings from Development → Production mode. This would help to ensure that the KDA engine is optimised for high availability and fast performance.

**Template for application settings**  
Choose a sample template to meet your use case. All settings can be edited after creating the application.

**Templates**

**Development**  
Use these settings for lowest cost.

**Production**  
Use these settings for high availability and fast, consistent performance.

Note how we don't have to change much of the pipeline in the cloud, but only the input data source. This is the advantage of cloud computing - the same pipeline can be used across multiple applications that require the same functionalities (e.g., data ingestion and machine learning inferences for this lab).