ILLINOIS INSTITUTE OF TECHNOLOGY

Department of Computer Science

CSP-554 Big Data Technologies

Taxi fare prediction using AWS Kinesis and Machine learning

Naveen Raju Sreerama Raju Govinda Raju
nsreeramarajugovinda@hawk.iit.edu
A20516868

Raghunath Babu
rbabu@hawk.iit.edu
A20511598

Prof. Joseph Rosen
Date:05/3/2023

Table of Contents

INTRODUCTION:

Recent machine learning and Artificial Intelligence model shifted its paradigm on relying on instant and streaming data from generators like the Internet of Things(IOT). This brings in the necessity of making predictions on continuous data streams instead of static data. Traditional AI/Ml frame works like python generators have limitations in handling shear amounts of data and instant processing. These models and frameworks which relied on persistent datasets and static data, became impractical for applications especially in business operations, stocks, health care etc. A new approach is laid to tackle this problem using streaming services like kafka or AWS kinesis, where the continuous data will be streamed from a cloud storage through a streaming service and a machine learning model that is deployed on the endpoint will make predictions for the streamed data and store the results. This also helped in managing and reusing the data streams which resulted in no utilization of data storage/ file systems.

This project presents a machine learning modeling on a streaming data using AWS tools(cutting edge tool that is extensively used in the industry)

LITERATURE REVIEW:

The paper [3], authored by Cristian Martin proposes an open-source framework called Kafka-ML that allows the management of machine learning and artificial intelligence pipelines through data streams using Apache Kafka. It offers a user-friendly web interface for defining, training, evaluating, and deploying ML models, and is managed through containerization technology for portability and easy distribution. The framework also includes features such as fault tolerance and high availability, and a novel approach to manage and reuse data streams to minimize data storage and file system utilization.

<u>Kafka ML architecture</u>

a) Kafka-ML has a user-friendly front-end for managing ML models and configurations, implemented in Angular. Has multi customer distribution and high rate message dispatching which helps in training multiple ML models in streaming.
b) The back-end serves a RESTful API for managing information, using Django and the Kubernetes API for deployment and management.
c) Model Training uses Kubernetes Jobs to train and containerize models, submitting them to the back-end.
d) Model Inference downloads trained models from the back-end, uses received stream data for predictions, and balances data ingestion with Kafka consumer groups.
e) The control logger sends control messages to the back-end for easy configuration of data parameters when deploying an ML model for inference.
f) Kafka-ML deploys Apache Kafka and ZooKeeper as Jobs in Docker containers in Kubernetes for efficient deployment and management, both internally and externally.

2) Pipeline of an ML model in Kafka-ML

a) Design and define ML models using popular frameworks such as TensorFlow/Keras in Kafka-ML through a Web UI. The pipeline can be automated using a RESTful API.
b) Create a logical set of models called a configuration in the Kafka-ML Web UI for evaluation, comparison, or training in parallel with the same data stream.
c) Set training parameters in the Kafka-ML Web UI and deploy the configuration for training. Models are fetched and loaded from Kafka-ML architecture.
d) Send data streams for training through Kafka topics, including data topics with training and evaluation streams and a control topic with information about the deployment.
e) Submit trained models and their metrics to the Kafka-ML architecture, view/edit results in the Web UI, and deploy for inference with load balancing and fault-tolerance using Apache Kafka's consumer group feature.
f) Deploy the trained model for inference by selecting the number of replicas to be deployed and configuring input/output topics and formats in the Web UI. Send encoded data streams with the defined format to the input topic for inference results to be sent to the output topic.

3) Data stream management through Apache Kafka distributed Log

This section discusses how Apache Kafka's distributed log allows for flexible data stream management and retention policies. The control messages specify the topic and position in the log, and the retention policy determines whether data streams can be reused for training. The delete retention policy is preferred for Kafka-ML.

DATASET DESCRIPTION:

All taxi and for-hire vehicle trips in New York City have been recorded in the publicly accessible Trip Record Data of the New York City Taxi and Limousine Commission (TLC). The data includes multiple data points like number of passengers, fare amounts, tip amounts, and pickup and dropoff times and locations. The dataset could be utilized by researchers, experts, and developers to investigate and understand New York City's transportation patterns. The TLC Trip Record Data is constantly being revised and can be downloaded in both raw and processed formats. The TLC website[5] provides users the option to download the data, alternatively they can access massive, publicly accessible datasets housed on AWS through the AWS Open Data Registry.

METHODOLOGY:

OVERVIEW:

In this project, a machine learning prediction is made on streaming data using tools namely Amazon SageMaker, Amazon Kinesis Datastream, Amazon APIs and Lambda function.
For this project the New York taxi data set is considered. This dataset is splitted into train and validation data. Upon training our model, that is built on AWS SageMaker using Linear Learner algorithm, the model is deployed on the endpoint to call during real time that is invoked using APIs and AWS lambda function. The stream of data is generated from a cloud based IDE Cloud 9, and using a python data generator scripts the data is injected by AWS Kinesis, and calls the Sagemaker using Apache Flink to make predictions for the streamed data. Then these predicted fares will be stored in a S3 bucket.

1.1 AMAZON SAGEMAKER

Amazon SageMaker[6] is a completely maintained machine learning platform which enables developers and data scientists to rapidly build, train, and launch models using machine learning in a fully functional and operational hosting environment. SageMaker's incorporation of a Jupyter notebook instance enables simple research and analysis of data sources and eliminating the necessity for server administration. It additionally includes enhanced machine learning algorithms that are capable of handling enormous data sets effectively in a distributed environment. These algorithms cover a wide range of machine learning tasks, such as regression, classification, clustering, and anomaly detection, among others. SageMaker also supports customized algorithms and frameworks natively, as well as flexible distributed training alternatives adapted to distinctive workflows.

1.2 AMAZON KINESIS DATASTREAM

Amazon Kinesis Data Streams [7] is a completely managed Amazon Web Services (AWS) serivce that facilitates users to collect, examine, and preserve large data streams in actual time. Producers, streams, and consumers constitute the Kinesis Data Streams framework. Producers are sources of data that produce records of data on a continual basis, whilst streams are data pipelines that retain and process the data records that comes in. Consumers are applications that ingest records of data via streams and perform some processing. Kinesis Data Streams also connects with additional services offered by AWS like Lambda, Elasticsearch, and Amazon S3, allowing you to build end-to-end real-time data processing pipelines.

1.3 AMAZON KINESIS DATA STREAMS

Amazon Kinesis Data Analytics[8] is a completely managed AWS service that allows you to handle and evaluate live data streams utilizing traditional SQL without coding or configuring infrastructure. It relies on the open-source Apache Flink stream processing architecture and facilitates processing of data through an array of sources, which includes Kinesis data streams, Kinesis Data Firehose, and Amazon DynamoDB Streams. Kinesis Data Analytics can utilize SQL queries to promptly evaluate gather insight from enormous amounts of streaming data, and the results can be preserved in Amazon S3 for future studies. Kinesis Data Analytics has been optimized to be flexible and fault-tolerant. It automatically allocated and expanded the amount of resources needed for processing the streaming data, and it is able to recover from errors automatically, guaranteeing your data analytics pipeline remains functional and operational.

1.4 AMAZON API GATEWAY

A managed service called Amazon API Gateway[9] aims to make it simpler to develop, implement, and maintain APIs for web applications. Developers can effortlessly create and publish RESTful APIs that provide users with access to the back end services, data, or business logic using API Gateway. The Lambda, EC2, and S3 services of AWS are all easily integrated with the API Gateway, which can handle multiple HTTP requests. Developers can create APIs with Amazon API Gateway in an array of languages, including Node.js, Python, Ruby, and Java. To restrict accessibility to their APIs, they may also offer various authorization protocols and API keys. For users to manage and safeguard APIs, the API Gateway additionally offers features like caching, throttling, and request validation.

1.5 AMAZON LAMBDA

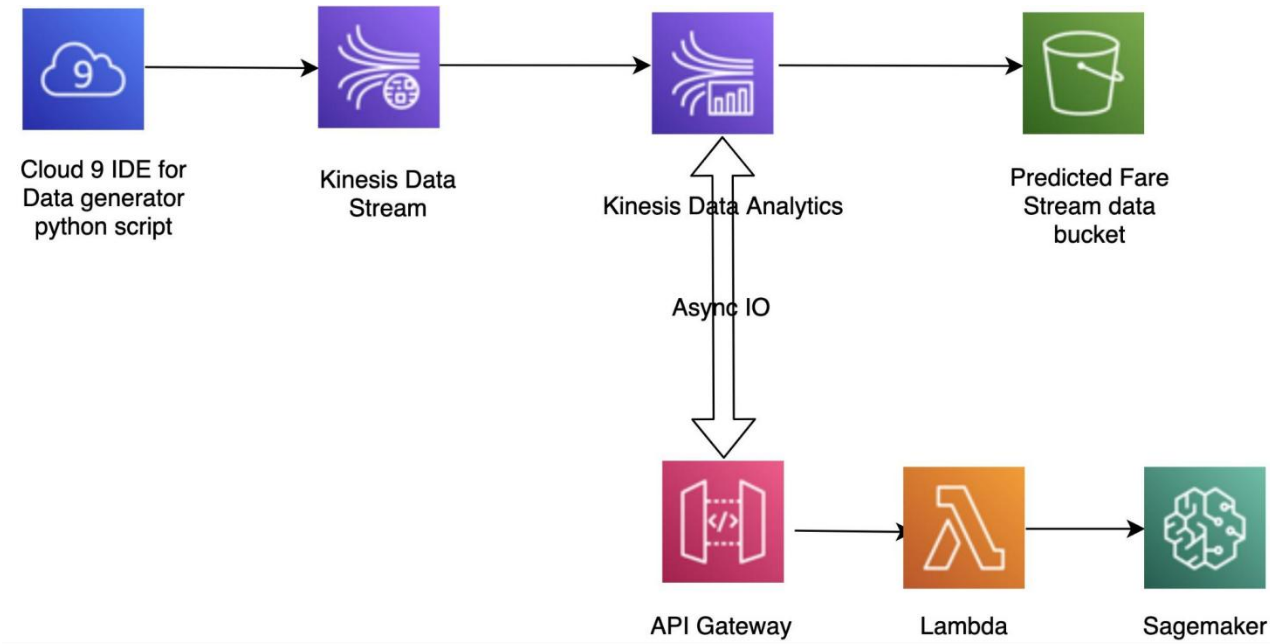
AWS's serverless computing tool called Amazon Lambda[10] enables developers to develop and maintain applications without being concerned about maintaining servers. It can execute code in response to numerous events, such as API calls, modifications to data in S3, and SQS alerts. It accepts an array of programming languages like Node.js, Python, Java, C#, and Go. For developing sophisticated serverless applications, Lambda interfaces with other AWS services like S3, DynamoDB, and Kinesis and modifies computing resources according to incoming requests. The platform provides a simple interface for creating, testing, and deploying Lambda functions as well as integrated monitoring and logging capabilities to help with optimizing performance and troubleshooting. With Lambda, developers are able to guarantee that applications are extremely scalable and accessible while lowering operational costs and complexity.

1.6 LINEAR LEARNER MACHINE LEARNING APPROACH

AWS provides the Amazon SageMaker Linear Learner machine learning approach[11] for regression tasks along with binary and multiclass classification problems. It is able to train models on data that is structured such as CSV files, Apache Parquet, or data formatted in LibSVM. It is intended for handling massive amounts of data. Linear Learner integrates with other AWS services which includes Amazon S3, Amazon Athena, and AWS Glue and supports both dense and sparse input formats. The method can dynamically modify hyperparameters to optimize model performance and combines gradient descent optimization with linear models to accomplish training. After the model has been trained, Linear Learner supports batch and real-time inference and has built-in functionality for monitoring and debugging. The Linear Learner algorithm also supports L1 and L2 regularization to prevent overfitting and improve model generalization. The algorithm can process both tabular and sparse data formats, and it supports several input data types, including CSV, RecordIO, and JSON. The SageMaker console provides a user-friendly interface for training, tuning, and deploying models using the Linear Learner algorithm.

IMPLEMENTATION:

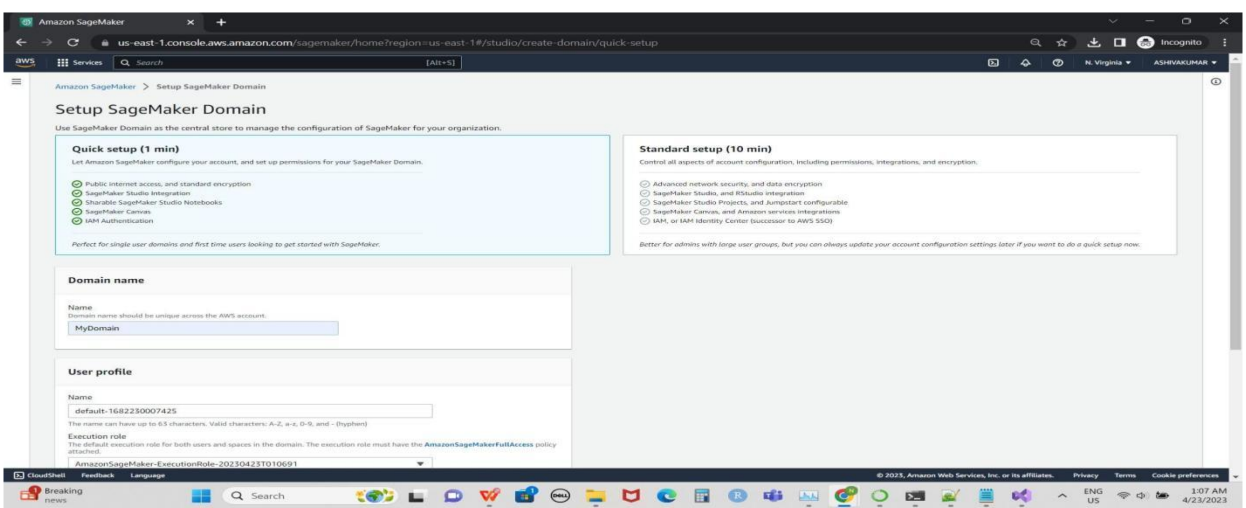
In this project, by referring [4], we're going to train a model using just a small portion of the NYC taxi rides dataset. The SageMaker linear learner algorithm, an integrated approach that is renowned for machine learning projects, will be used as the model's foundation. Then, in order to make this model accessible in actual time, we will implement it behind a SageMaker endpoint. We will utilize a Python data generation tool to stream taxi rides in real-time via Cloud 9. We will make use of the Apache Flink application combined with the Amazon Kinesis Datastream and Amazon Kinesis Data Analytics services to process the streaming data and invoke the SageMaker endpoint. We are going to set up a Java application for Apache Flink in the Amazon Kinesis Data Analytics service. For any incoming streaming data, this application will asynchronously invoke the SageMaker endpoint. The data will be processed and stored in the S3 bucket as a dataset with predicted fare. The architecture is built with great scalability and availability to handle an enormous amount of streaming data. The architecture of the entire model referred from [4] is given below:

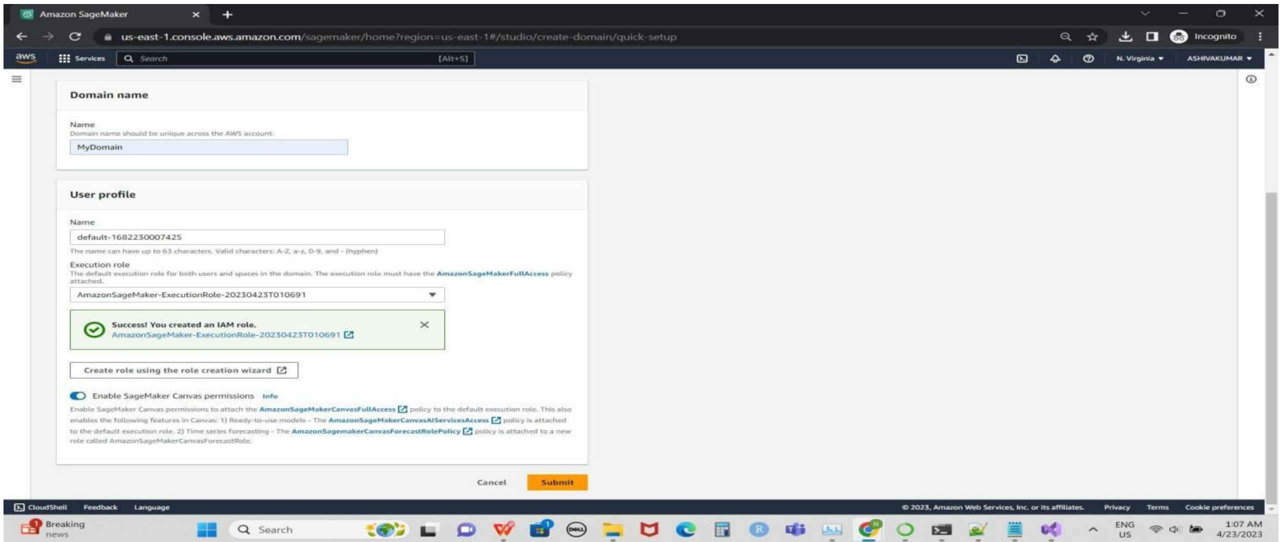


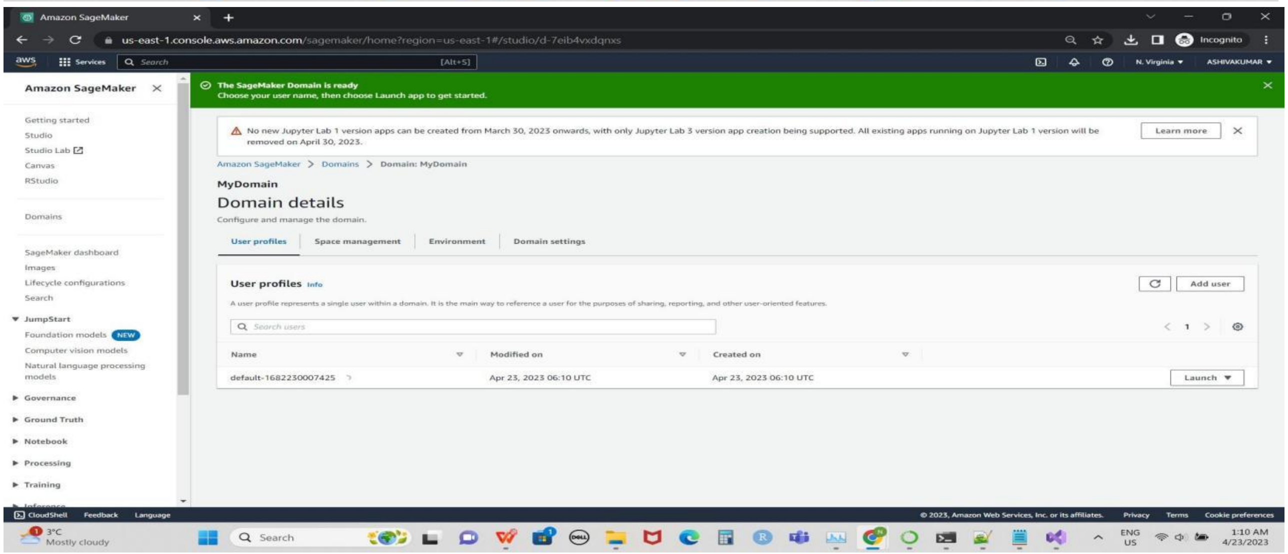
1) Launching Amazon SageMaker studio

Amazon SageMaker is an IDE for machine learning which helps to build, train, debug, deploy and monitor machine learning models

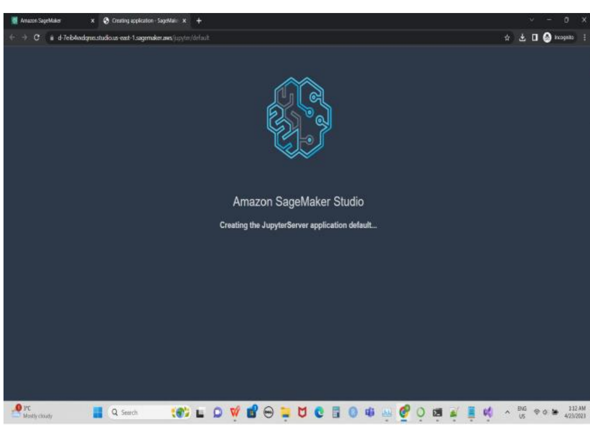
- In AWS console first select nearest AWS region. We have chosen N.Virginia.
- Search for Amazon SageMaker in search and Click on Get Started and select setup SageMaker Domain.
- Enter the domain name and user profile name
- Create a new IAM role under Execution role. Select access option "Any S3 bucket".
- Click on create role.

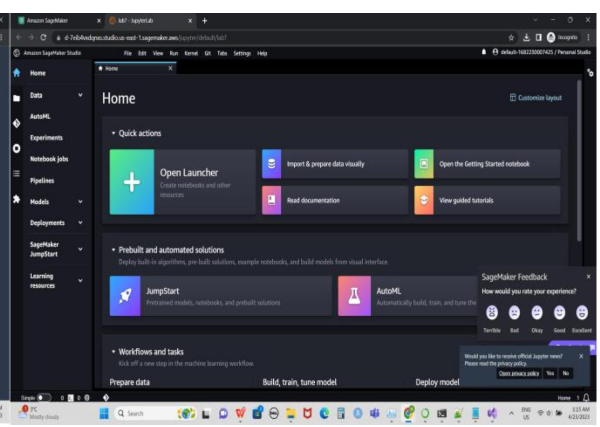






2) Launch the SageMaker studio





From the above shown window click on File and select create new jupyter note notebook(configuration: kernel - Python3, Instance type - ml.t3.medium)

3) Download Apache Flink Java application jar file

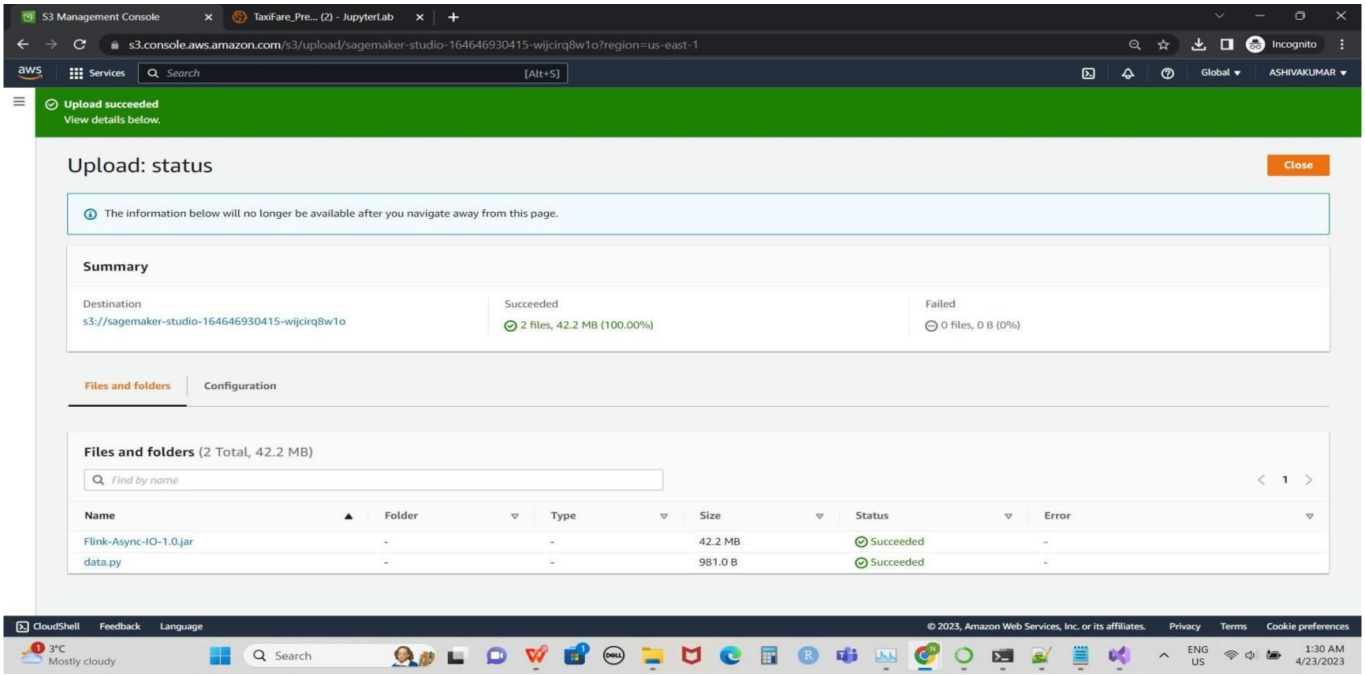
Amazon SageMaker utilizes the Apache Flink Java application jar file for developing and executing streaming data processing applications. Flink is an open-source distributed platform that allows for efficient processing of real-time data streams. The jar file contains the code and dependencies required to run a Flink application, which can be uploaded to SageMaker to create a job for real-time data processing. With SageMaker's managed Flink environment, users can deploy, operate, and monitor Flink applications without the need to manage the underlying infrastructure.

4) Write data generator streaming python 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",
    "1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0",
    "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))
```

5) S3 bucket

Upload downloaded Apache Flink Java application jar file and data generator streaming python script under sage maker studio in s3 bucket.



6) Model training

Upload the dataset to the same directory where we created a jupyter notebook in SageMaker studio.

A) Write program for training

```
import os
import boto3
import re
import sagemaker
import numpy as np
```

```
role = sagemaker.get_execution_role()
sess = sagemaker.Session()
region = boto3.Session().region_name

data_bucket=sess.default_bucket()
data_prefix = "1p-notebooks-datasets/taxi/text-csv"

output_bucket = data_bucket
output_prefix = "sagemaker/DEMO-linear-learner-taxifare-regression"
```

The above lines of code creates S3 bucket for training data. It also creates another s3 bucket for saving code and model artifacts.

B) Read training data

```
# cell 04
import boto3
FILE_TRAIN = "nyc-taxi.csv"


import pandas as pd
df = pd.read_csv(FILE_TRAIN, sep=",", encoding="latin1", names=["fare_amount","vendor_id","pickup_datetime","d
print(df.head(5))
```

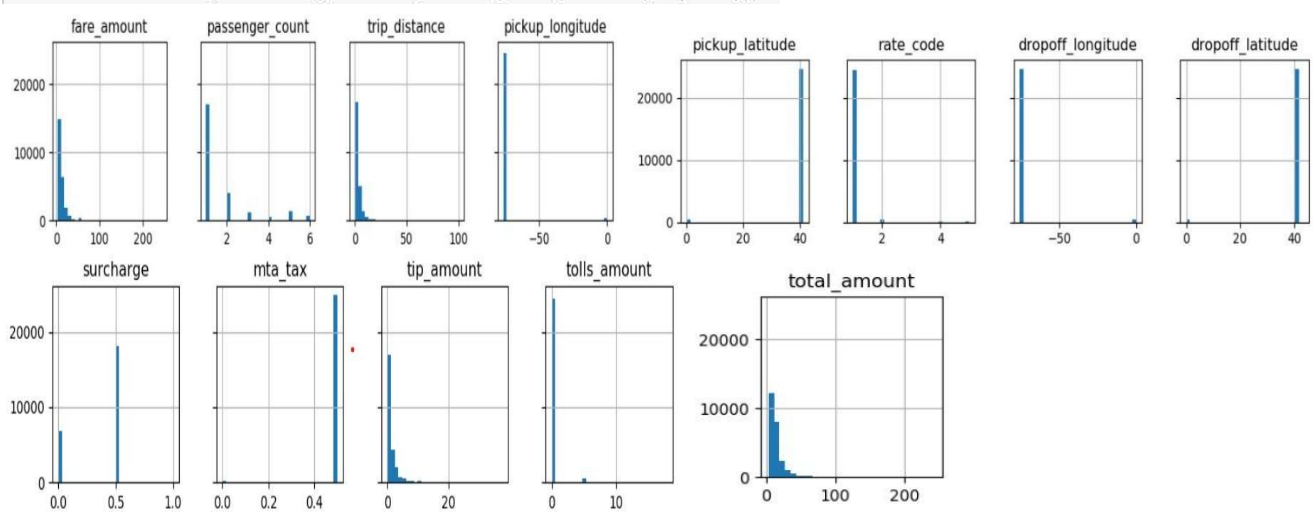
```
   fare_amount vendor_id pickup_datetime dropoff_datetime  passenger_count  \
0         18.0       CMT    01/11/12 1:18    01/11/12 1:35                1
1         10.0       CMT    01/11/12 1:18    01/11/12 1:28                1
2         35.5       CMT    01/11/12 1:18    01/11/12 2:16                1
3          5.5       CMT    01/11/12 1:18    01/11/12 1:22                1
4         10.5       CMT    01/11/12 1:18    01/11/12 1:27                1

   trip_distance  pickup_longitude  pickup_latitude  rate_code  \
0            5.4        -73.984519        40.779776          1
1            2.4        -73.996082        40.753302          1
2            5.4        -73.970535        40.799144          1
3            1.1        -73.956560        40.771124          1
4            2.7        -73.959062        40.771722          1

  store_and_fwd_flag  dropoff_longitude  dropoff_latitude payment_type  \
0                  N         -73.947342         40.764681          CRD
1                  N         -73.985783         40.727865          CSH
2                  N         -73.957026         40.770164          CSH
3                  N         -73.960994         40.757343          CRD
4                  N         -73.967998         40.800170          CRD

   surcharge  mta_tax  tip_amount  tolls_amount  total_amount
0        0.5      0.5        3.80           0.0         22.80
1        0.5      0.5        0.00           0.0         11.00
2        0.5      0.5        0.00           0.0         36.50
3        0.5      0.5        1.62           0.0          8.12
4        0.5      0.5        2.85           0.0         14.35
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24998 entries, 0 to 24997
Data columns (total 18 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   fare_amount         24998 non-null  float64
 1   vendor_id           24998 non-null  object
 2   pickup_datetime     24998 non-null  object
 3   dropoff_datetime    24998 non-null  object
 4   passenger_count     24998 non-null  int64
 5   trip_distance       24998 non-null  float64
 6   pickup_longitude    24998 non-null  float64
 7   pickup_latitude     24998 non-null  float64
 8   rate_code           24998 non-null  int64
 9   store_and_fwd_flag  13789 non-null  object
 10  dropoff_longitude   24998 non-null  float64
 11  dropoff_latitude    24998 non-null  float64
 12  payment_type        24998 non-null  object
 13  surcharge           24998 non-null  float64
 14  mta_tax             24998 non-null  float64
 15  tip_amount          24998 non-null  float64
 16  tolls_amount        24998 non-null  float64
 17  total_amount        24998 non-null  float64
dtypes: float64(11), int64(2), object(5)
memory usage: 3.4+ MB
```

- Frequency table for each categorical table

```
display(df.describe())
%matplotlib inline
hist = df.hist(bins=30, sharey=True, figsize=(10, 10))
```



- Data preprocessing:

We observed that the "store_and_fwd_flg" column has very low variability, with 98% of the values being "N" and only 2% being "Y." Therefore, this column is unlikely to have a significant impact on the target variable, which is the "fare_amount." Additionally, based on our knowledge of the domain, we know that the "payment_type" column is not related to the trip fare. Hence, we removed both of these features from the dataset.

```
df = df.drop(['payment_type', 'store_and_fwd_flag'], axis=1)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24998 entries, 0 to 24997
Data columns (total 16 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   fare_amount        24998 non-null  float64
 1   vendor_id          24998 non-null  object
 2   pickup_datetime    24998 non-null  object
 3   dropoff_datetime   24998 non-null  object
 4   passenger_count    24998 non-null  int64
 5   trip_distance      24998 non-null  float64
 6   pickup_longitude   24998 non-null  float64
 7   pickup_latitude    24998 non-null  float64
 8   rate_code          24998 non-null  int64
 9   dropoff_longitude  24998 non-null  float64
 10  dropoff_latitude   24998 non-null  float64
 11  surcharge          24998 non-null  float64
 12  mta_tax            24998 non-null  float64
 13  tip_amount         24998 non-null  float64
 14  tolls_amount       24998 non-null  float64
 15  total_amount       24998 non-null  float64
dtypes: float64(11), int64(2), object(3)
memory usage: 3.1+ MB
```

- We noticed that the dataset contains two features called "pickup_datetime" and "dropoff_datetime" which represent the start and end times of a taxi ride. We know that the fare amount of a taxi ride is influenced by the duration of the trip therefore we created a new feature that calculates the ride duration using these two features.

```
df['dropoff_datetime']= pd.to_datetime(df['dropoff_datetime'])
df['pickup_datetime']= pd.to_datetime(df['pickup_datetime'])
df['journey_time'] = (df['dropoff_datetime'] - df['pickup_datetime'])
df['journey_time'] = df['journey_time'].dt.total_seconds()
df['journey_time']
```

```
0        1020.0
1         600.0
2        3480.0
3         240.0
4         540.0
          ...
24993     540.0
24994     420.0
24995     600.0
24996    1200.0
24997     420.0
Name: journey_time, Length: 24998, dtype: float64
```

- Now, after creating 'journey_time feature' we dropped 'pickup_datetime' and 'dropoff_datetime' features.

```
df = df.drop(['dropoff_datetime', 'pickup_datetime'], axis=1)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24998 entries, 0 to 24997
Data columns (total 15 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   fare_amount        24998 non-null  float64
 1   vendor_id          24998 non-null  object
 2   passenger_count    24998 non-null  int64
 3   trip_distance      24998 non-null  float64
 4   pickup_longitude   24998 non-null  float64
 5   pickup_latitude    24998 non-null  float64
 6   rate_code          24998 non-null  int64
 7   dropoff_longitude  24998 non-null  float64
 8   dropoff_latitude   24998 non-null  float64
 9   surcharge          24998 non-null  float64
 10  mta_tax            24998 non-null  float64
 11  tip_amount         24998 non-null  float64
 12  tolls_amount       24998 non-null  float64
 13  total_amount       24998 non-null  float64
 14  journey_time       24998 non-null  float64
dtypes: float64(12), int64(2), object(1)
memory usage: 2.9+ MB
```

- vedor_id is a categorical feature so we changed it to float as need for Liner learner algorithm.

```
df = pd.get_dummies(df, dtype=float)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24998 entries, 0 to 24997
Data columns (total 16 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   fare_amount        24998 non-null  float64
 1   passenger_count    24998 non-null  int64
 2   trip_distance      24998 non-null  float64
 3   pickup_longitude   24998 non-null  float64
 4   pickup_latitude    24998 non-null  float64
 5   rate_code          24998 non-null  int64
 6   dropoff_longitude  24998 non-null  float64
 7   dropoff_latitude   24998 non-null  float64
 8   surcharge          24998 non-null  float64
 9   mta_tax            24998 non-null  float64
 10  tip_amount         24998 non-null  float64
 11  tolls_amount       24998 non-null  float64
 12  total_amount       24998 non-null  float64
 13  journey_time       24998 non-null  float64
 14  vendor_id_CMT      24998 non-null  float64
 15  vendor_id_VTS      24998 non-null  float64
dtypes: float64(14), int64(2)
memory usage: 3.1 MB
```

- Train, Test and Validation split

```
import numpy as np

train_data, validation_data, test_data = np.split(df.sample(frac=1, random_state=1729), [int(0.7 * len(df)), int(0.9 * len(df))]
train_data.to_csv('train.csv', header=False, index=False)
validation_data.to_csv('validation.csv', header=False, index=False)
test_data.to_csv('test.csv', header=False, index=False)
```

- Used the Boto3 library to upload three CSV files ("train.csv", "validation.csv", and "test.csv") to an S3 bucket specified by the "data_bucket" variable and in specific prefixes within that bucket specified by "data_prefix".

```
boto3.Session().resource('s3').Bucket(data_bucket).Object(os.path.join(data_prefix, 'train/train.csv')).
upload_file('train.csv')

boto3.Session().resource('s3').Bucket(data_bucket).Object(os.path.join(data_prefix, 'validation/validation.csv
upload_file('validation.csv')

boto3.Session().resource('s3').Bucket(data_bucket).Object(os.path.join(data_prefix, 'test/test.csv')).
upload_file('test.csv')
```

- We set up the data channels and the algorithm to work together by creating "sagemaker.session.s3_input" objects from data channels. The objects are put into a dictionary that the algorithm can easily consume. We also specify "text/csv" as the "content_type" for the pre-processed files in the "data_bucket". We create two channels one for training and another for validation. Testing samples we created above will be used for prediction.

```
s3_train_data = f"s3://{data_bucket}/{data_prefix}/train"

s3_validation_data = f"s3://{data_bucket}/{data_prefix}/validation"

s3_test_data = f"s3://{data_bucket}/{data_prefix}/test"

output_location = f"s3://{output_bucket}/{output_prefix}/output"


train_data = sagemaker.inputs.TrainingInput(
    s3_train_data,
    distribution="FullyReplicated",
    content_type="text/csv",
    s3_data_type="S3Prefix",
    record_wrapping=None,
    compression=None,
)
validation_data = sagemaker.inputs.TrainingInput(
    s3_validation_data,
    distribution="FullyReplicated",
    content_type="text/csv",
    s3_data_type="S3Prefix",
    record_wrapping=None,
    compression=None,
)
```

- Retrieve image for the Linear Learner Algorithm according to the region

Linear Learner Algorithm:

```
from sagemaker.image_uris import retrieve

container = retrieve("linear-learner", boto3.Session().region_name, version="1")
```

- We create an estimator, which is configured to use the Linear Learner container image. We set the training parameters and hyperparameters configuration for the estimator to define how the model will be trained.

```
%%time
import boto3
import sagemaker
from time import gmtime, strftime

sess = sagemaker.Session()

job_name = "DEMO-linear-learner-taxifare-regression-" + strftime("%H-%M-%S", gmtime())
print("Training job", job_name)

linear = sagemaker.estimator.Estimator(
    container,
    role,
    input_mode="File",
    instance_count=1,
    instance_type="ml.m4.xlarge",
    output_path=output_location,
    sagemaker_session=sess,
)

linear.set_hyperparameters(
    epochs=16,
    wd=0.01,
    loss="absolute_loss",
    predictor_type="regressor",
    normalize_data=True,
    optimizer="adam",
    mini_batch_size=1000,
    lr_scheduler_step=100,
    lr_scheduler_factor=0.99,
    lr_scheduler_minimum_lr=0.0001,
    learning_rate=0.1,
)
```

- After creating the Estimator classes, the instances we requested are provisioned and configured with the necessary libraries. Then, the data is downloaded from our channels into the instance. Subsequently, the training process commences, and during this process, the data logs display various losses, including Mean Average Precision (mAP) on the validation data, for every iteration of the dataset.

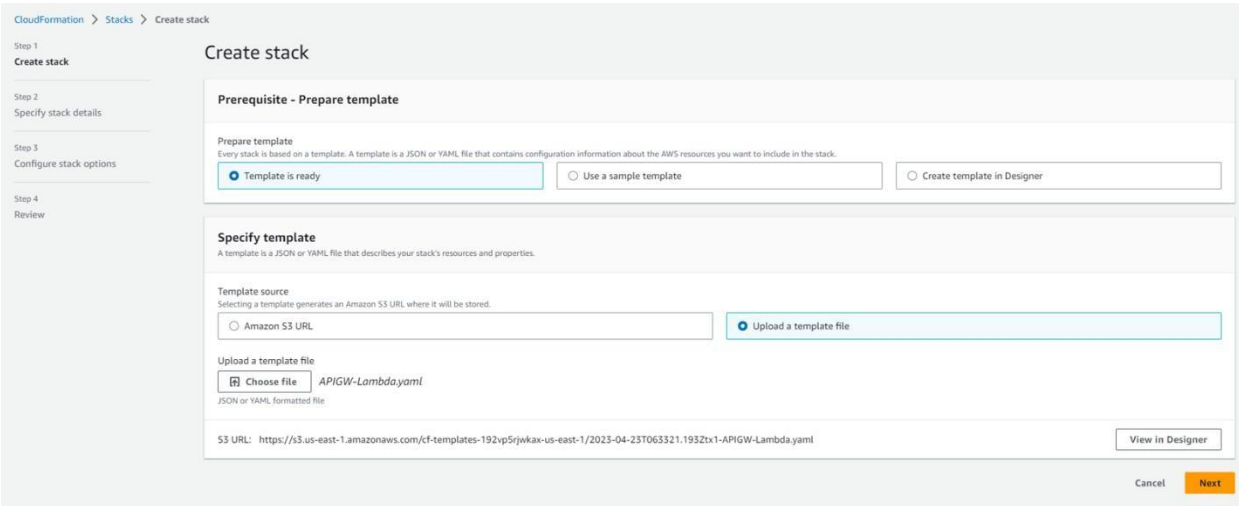
```
%%time
linear.fit(inputs={"train": train_data, "validation": validation_data}, job_name=job_name)
```

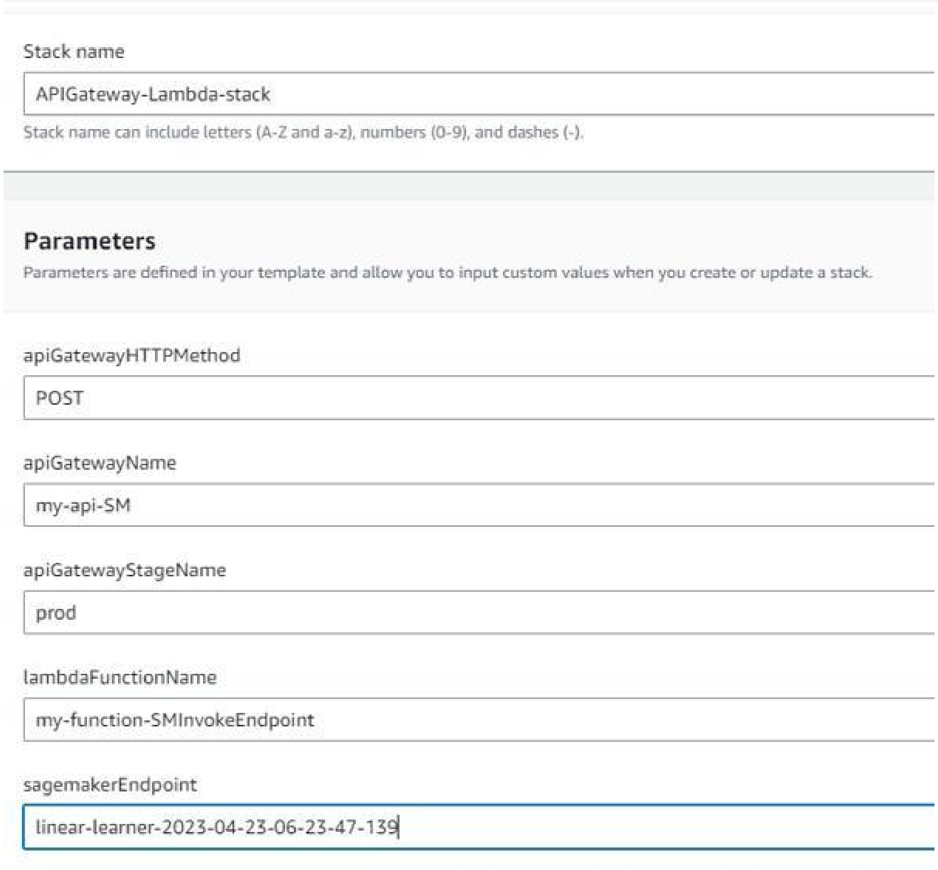
- Host the model

After the training process is complete we deploy the trained model on Amazon SageMaker real-time hosted endpoint. This will help to generate predictions from the model. We did not host the model on the same type of instance used for training because training is computationally expensive task that has different computation and memory requirements from hosting. Hence, we can choose a different instance type for hosting the model. We trained our model on an ml.m4.xlarge instance, but we used ml.c4.xlarge to host the model because this instance is a less expensive cpu instance.

```
linear_predictor = linear.deploy(initial_instance_count=1, instance_type="ml.c4.xlarge")

print(f"\ncreated endpoint: {linear_predictor.endpoint_name}")
```

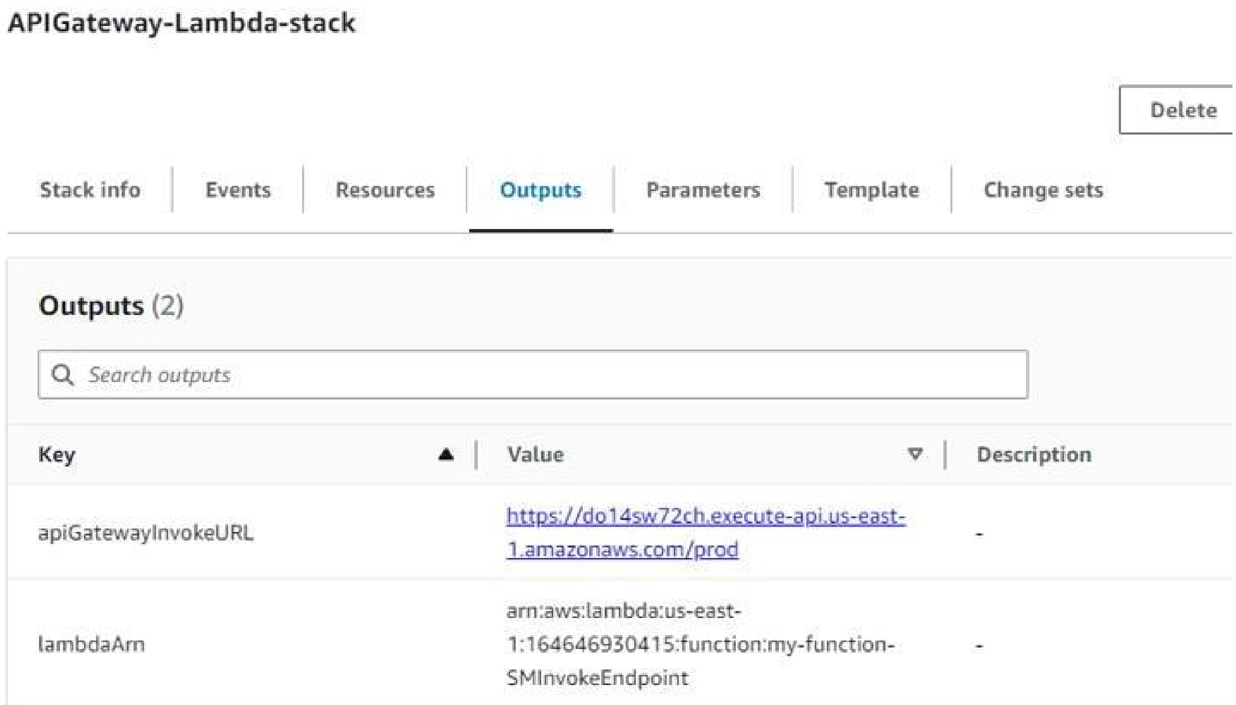
7) Creating Stack in Cloud Formation





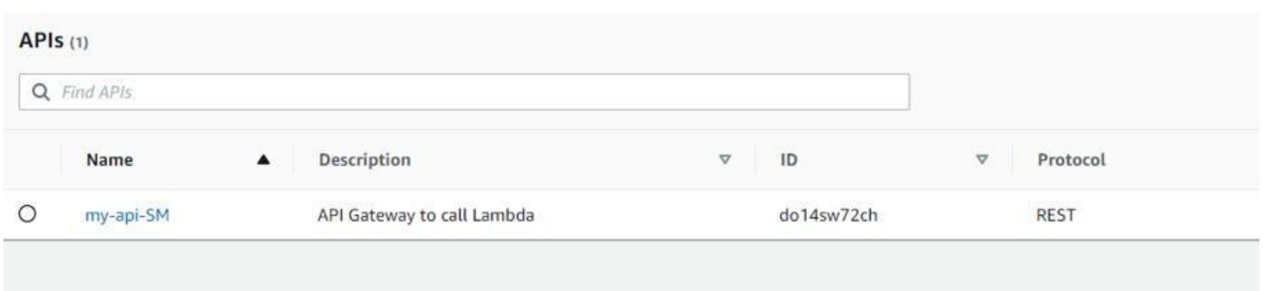
- This is done to create an API Gateway and Lambda Function.
- In the sage maker endpoint the created python .ipynb file is entered.
- The API gateway attributes are declared

- A Lambda function has been created to call a SageMaker endpoint and make predictions for taxi fares. This Lambda function is then called by API Gateway to obtain the predicted fare for streaming data that is being sent from Kinesis Data Analytics.



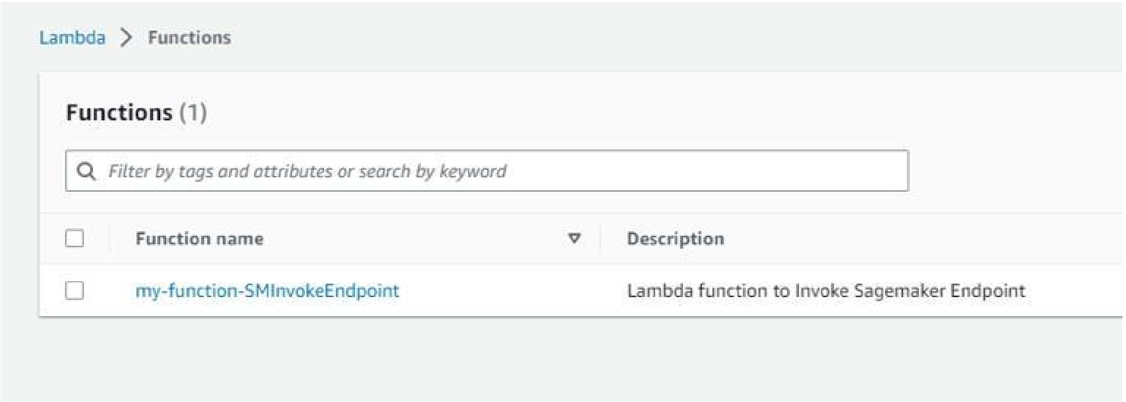
The created API gateway and lambda function has been initialized.

8) Verify the created API gateway function and lambda function



enable AWS Kinesis Data Analytics Apache Flink application to invoke the API Gateway endpoint available for the Lambda function,

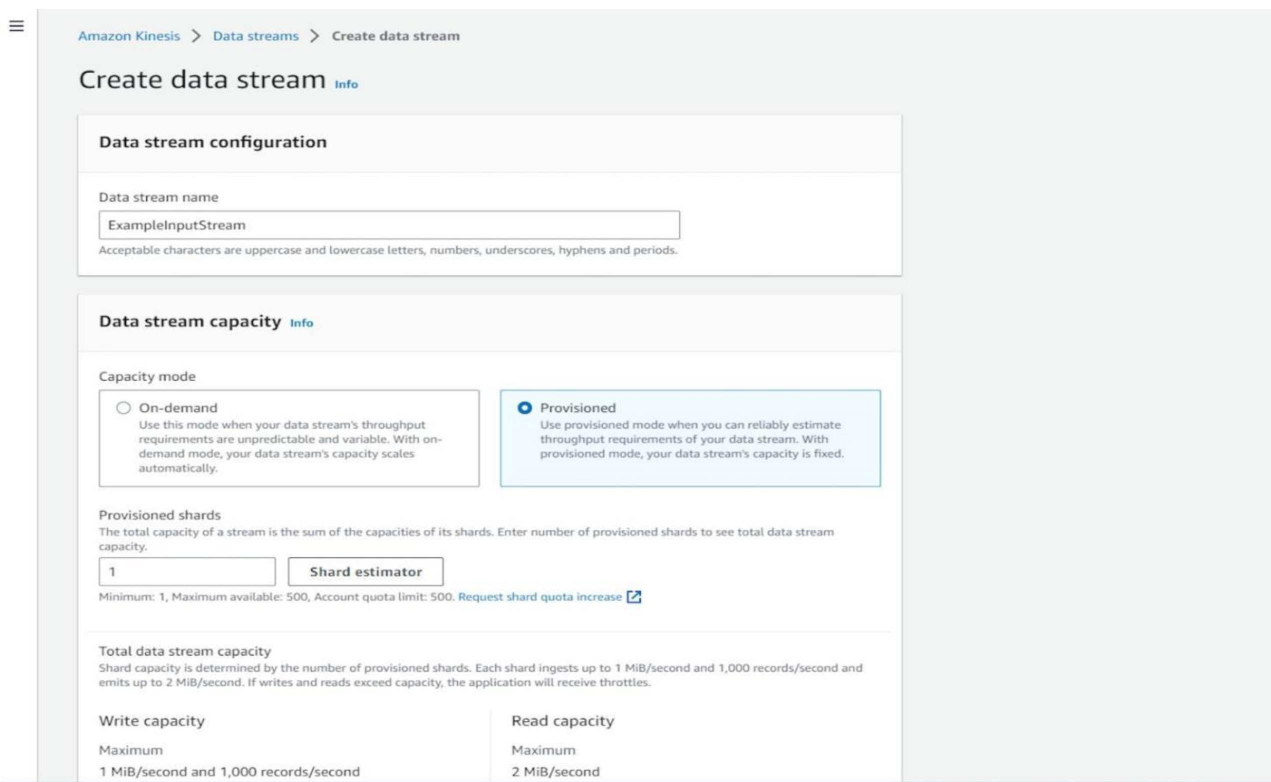
- Accessing the AWS API Gateway Service console and navigate to the API Gateway instance that corresponds to the Lambda function.
- The invoke URL for the 'prod' stage is saved for later use. This URL will be used to invoke the API Gateway endpoint from the Kinesis Data Analytics application as an ASync I/O endpoint.



Checking the created API and Lambda functions in their respective application window.

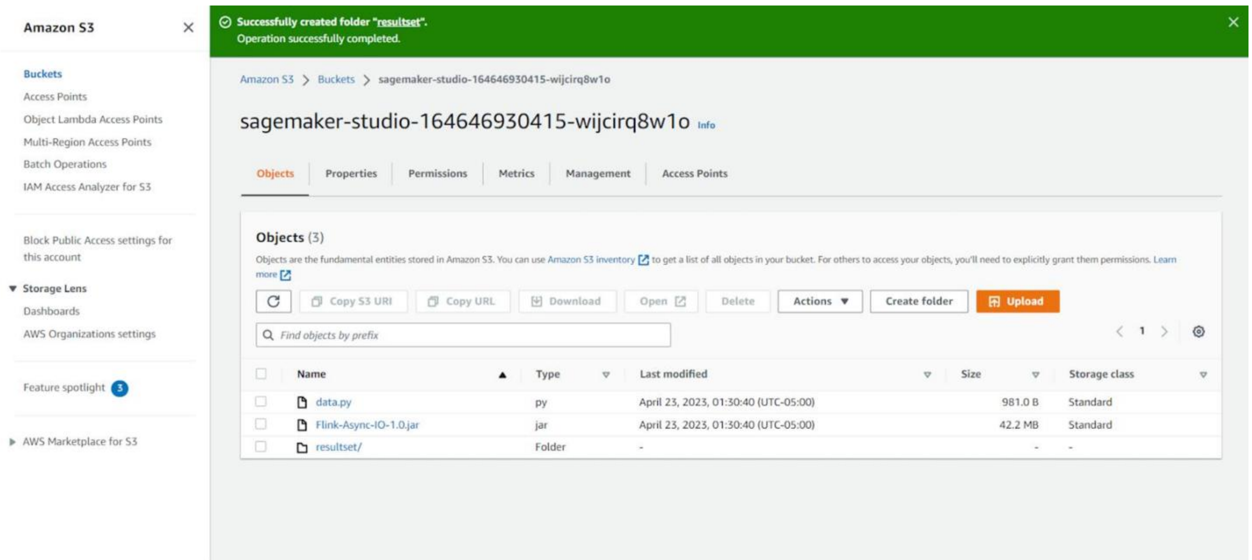
9) CREATING KINESIS DATA STREAM:

A Kinesis data stream is created in the Amazon Kinesis Data Stream Service in the AWS console. The Data stream's name is set as "ExampleInput Stream" and the data stream capacity is set as provisioned. The number of Provsioned shards is set as 1.

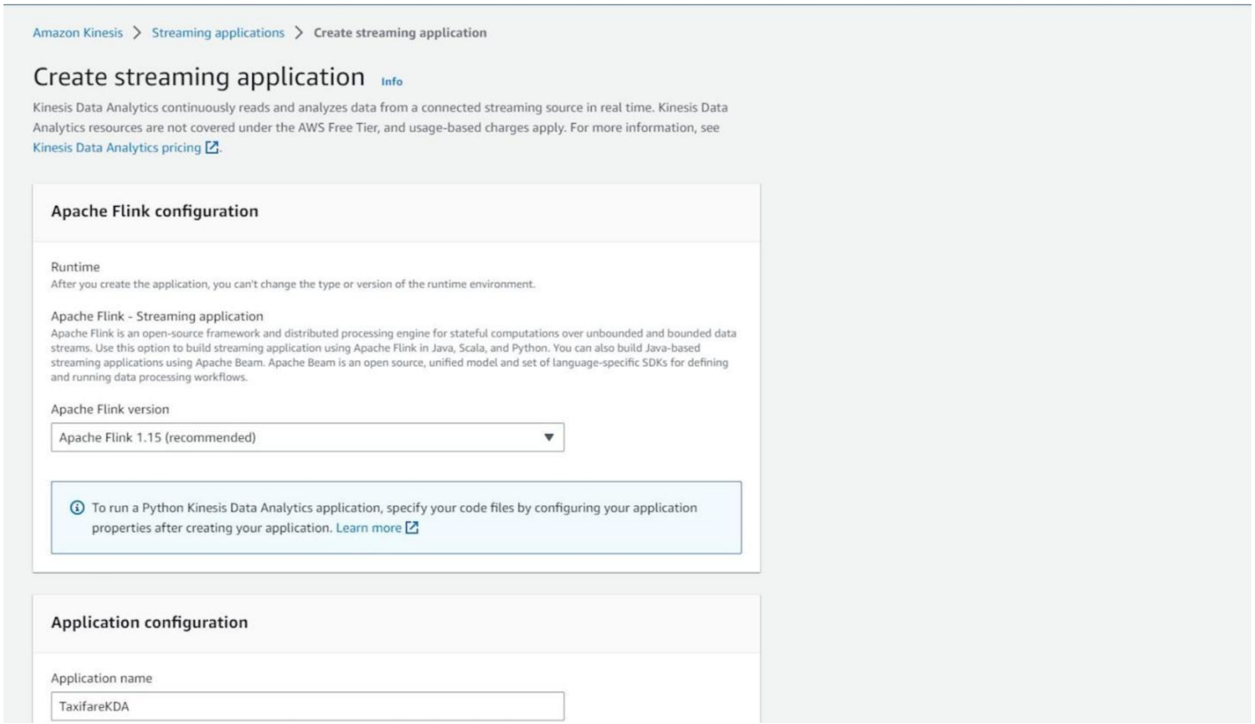


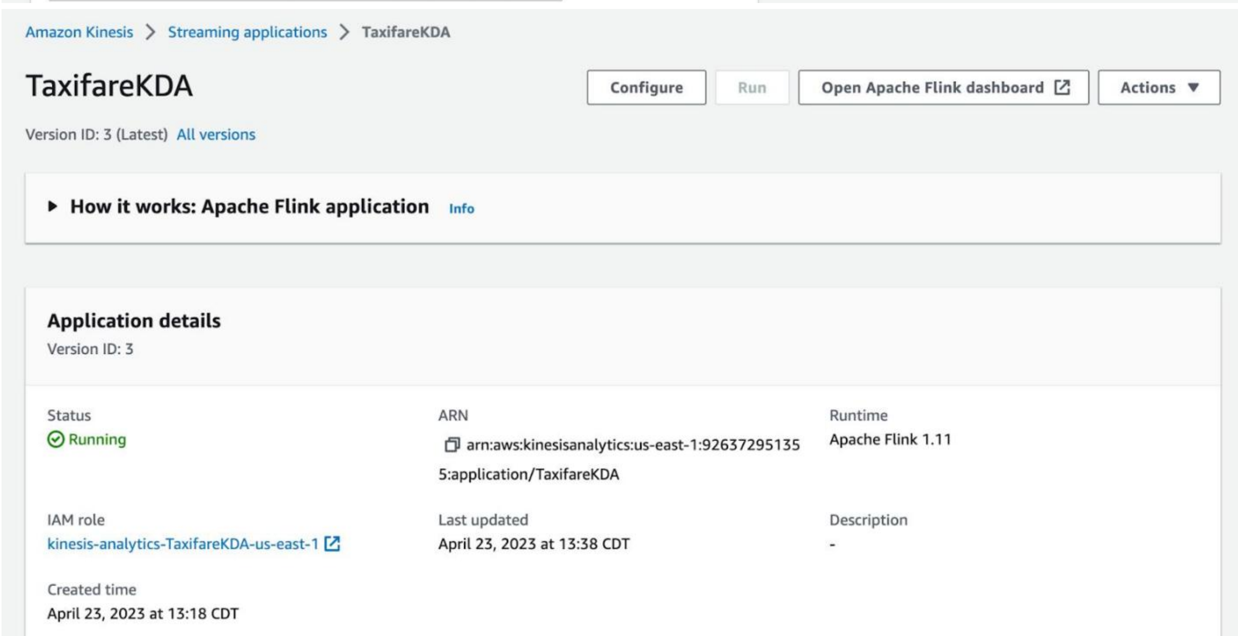
10) Set up Kinesis data analytics(Apache Flink application)

A folder “resultset” is create in the sagemaker studio bucket in the S3 console. The reason why the S3 bucket is created before creating the Kinesis Data Analytics Application (KDA) is because the KDA would store all the inference results in the S3 bucket. The inference results are generated by sagemaker endpoint and returned to the KDA in asynchronous input/output manner. The path of the “resulset” folder will be passed to the KDA for storing the results. The jar file for this Apache Flink application is accessible in s3 bucket.

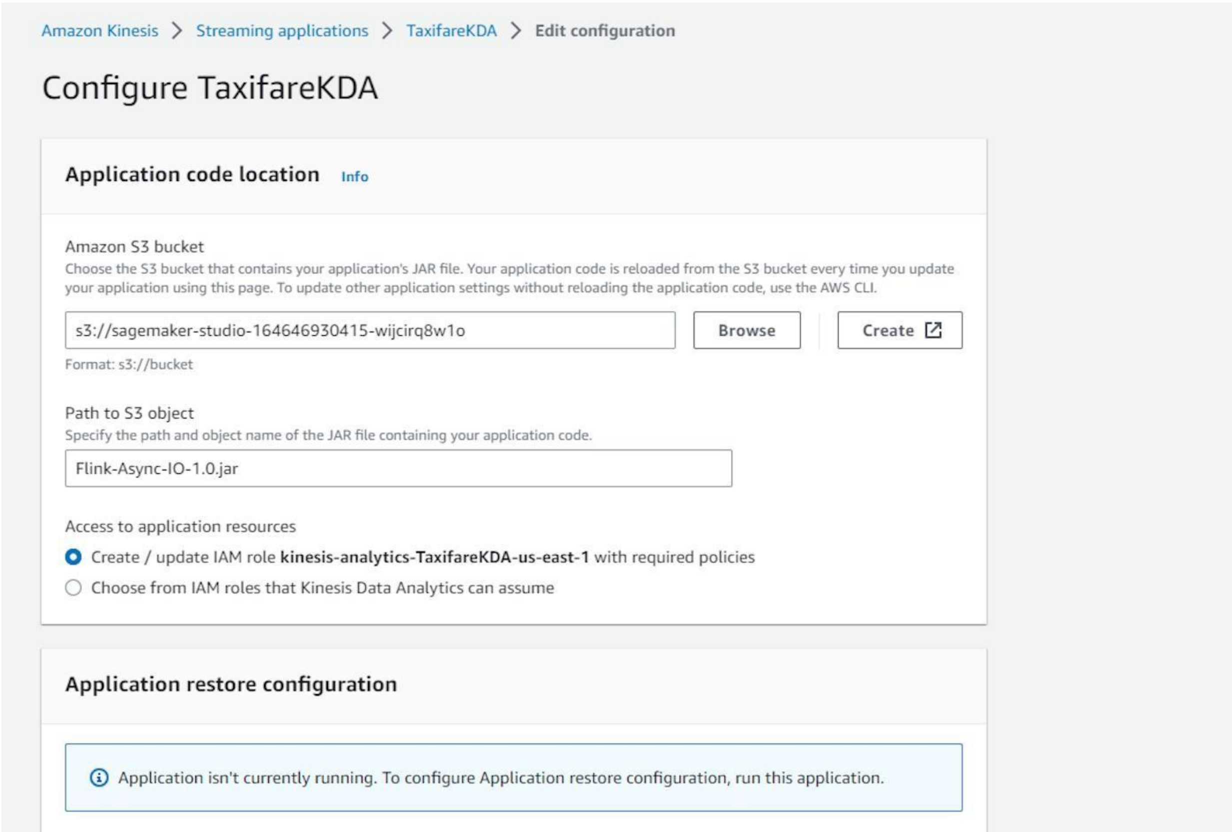


The Apache Flink java application is created. Apache Flink will make use of its Async I/O operator feature to bring in the prediction of the taxi fares in the real- time incoming data stream. The API Gateway is triggered asynchronously for each data record to get the taxi fare prediction. The predictions are then stored in the S3 bucket with the corresponding data record. The apache flink version is chosen as 1.15, which is the recommended version. The name of the application is given as “TaxifareKDA”. The template is chosen as “development” because of its low costs.





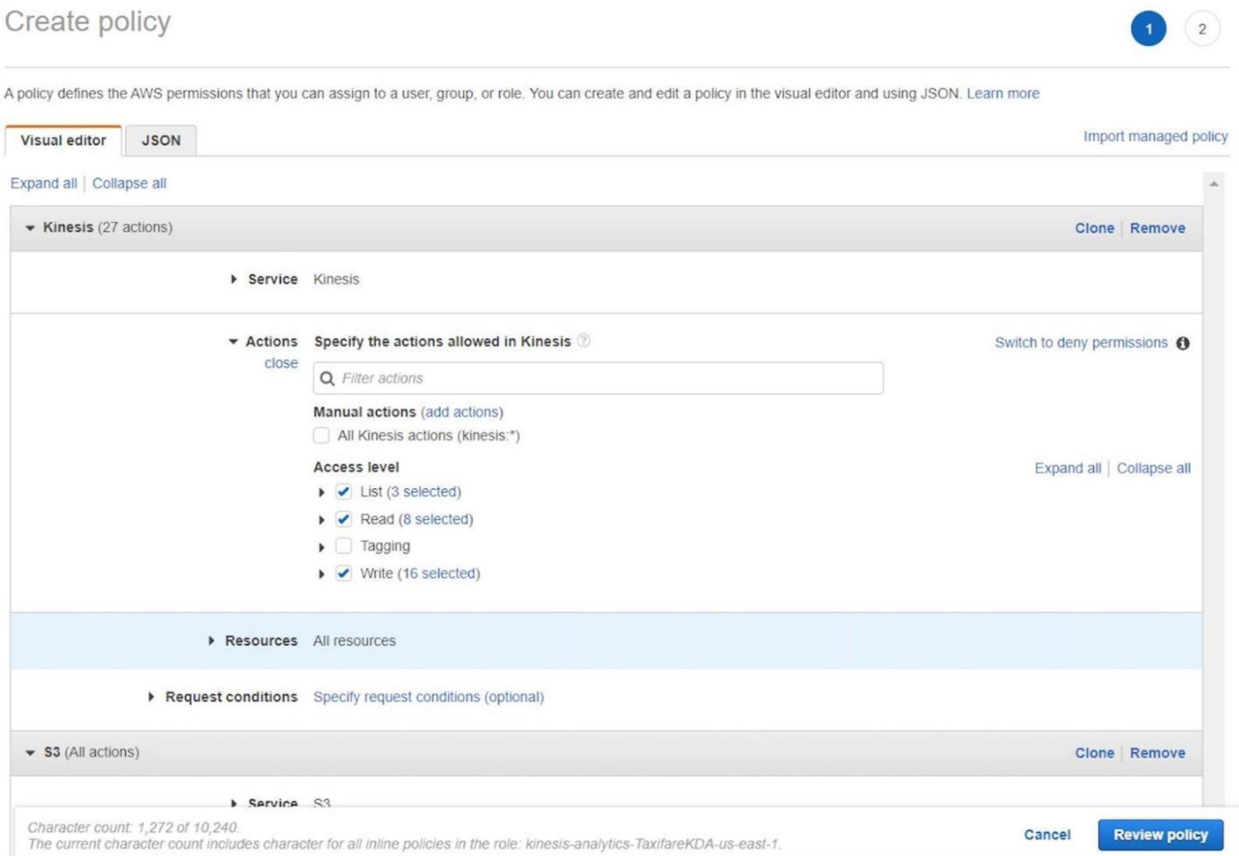
The next step is to configure the application “TaxifareKDA”. Under Amazon S3 Bucket, the name of the S3 bucket is specified and under Path to S3 Object, the jar file name is provided.

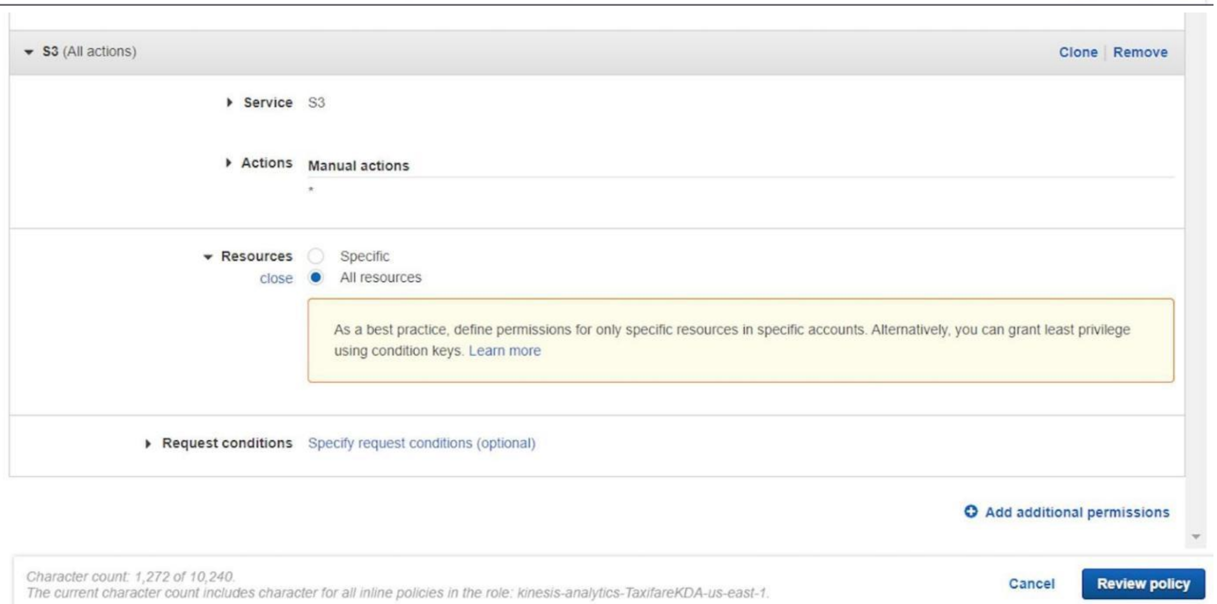


In the runtime properties section the Group ID has been created as FlinkApplicationProperties and a total of four key- value pairs are created. The names of the key and their corresponding values are provided in their respective columns.

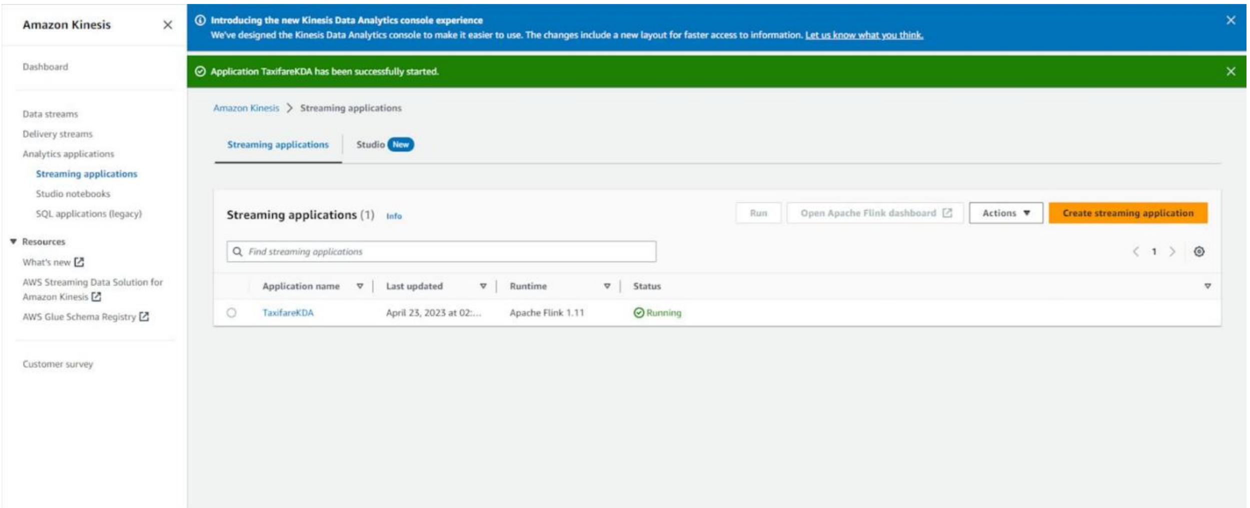
11) Give suitable IAM policies and run

The current permissons of the KDA application does not have the permissions to List and read. It also does not have the permissions for the S3 bucket to list or read. So some more IAM policies are to be added to this application. To give the permissions under “create inline policies” , the kinesis service is selected and once it is selected, under “access levels”, List and Read is selected. Now, for S3 all actions and resources are selected.

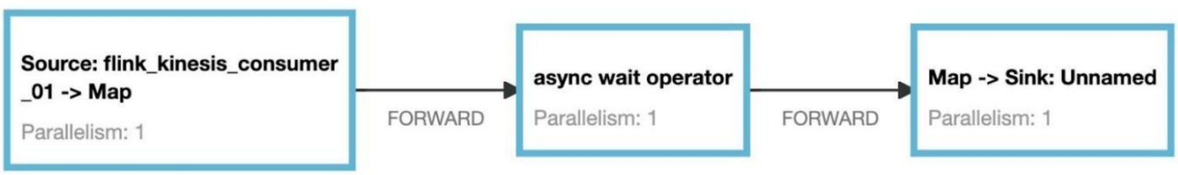




The next step is to create the policy by giving it a name. The kinesis application is then run by selecting “Run Without Snapshot” option. After running it, the application is successfully started.

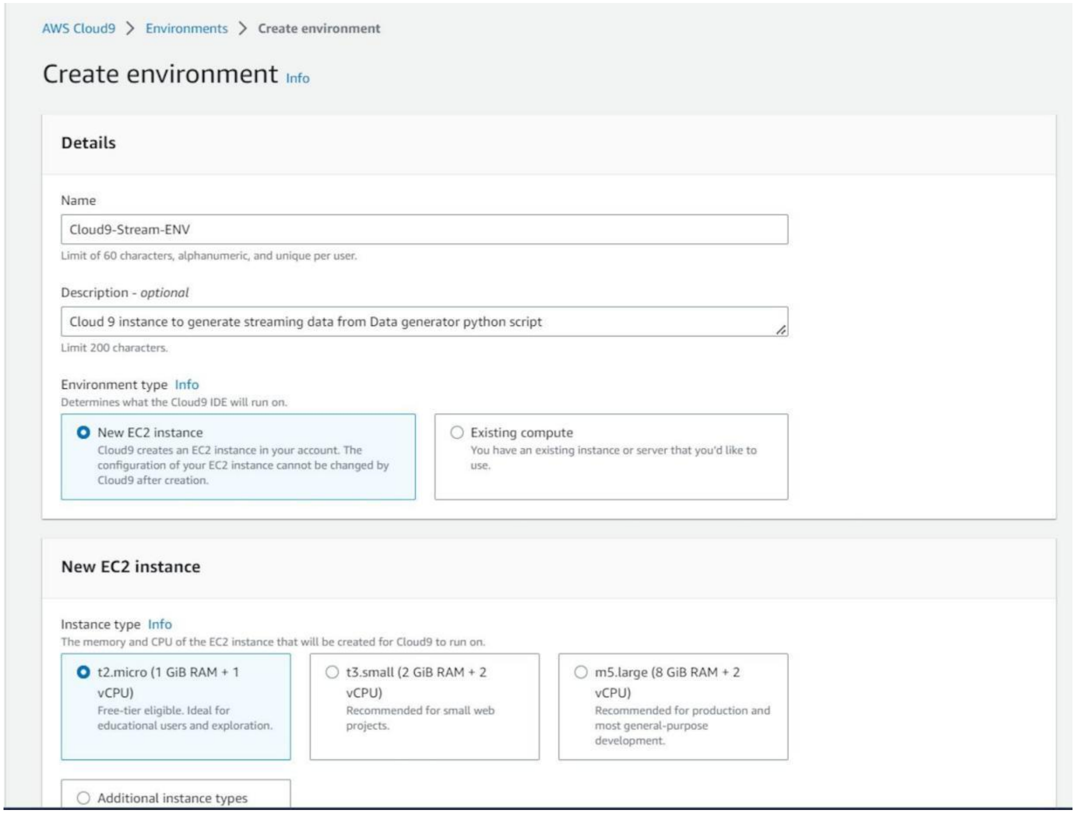


The application graph is given below. The application graph is a visual representation of the data flow consisting of the operators and intermediate results.



In the S3 bucket , we have the python code (data.py) for data generation, which would generate the streaming data. It randomly picks and send the streaming data to the kinesis data stream that was created before. The kinesis data analytics application will take in the streaming data once the streaming data reaches the data stream. The API Gateway is then called and then the resulting dataset which contains the taxi fare predicted values would be stored in the S3 bucket “resultset”.

The cloud 9 environment is created by selecting all the default values.



A new fie is created and the data.py file is copied to file and then it is saved.

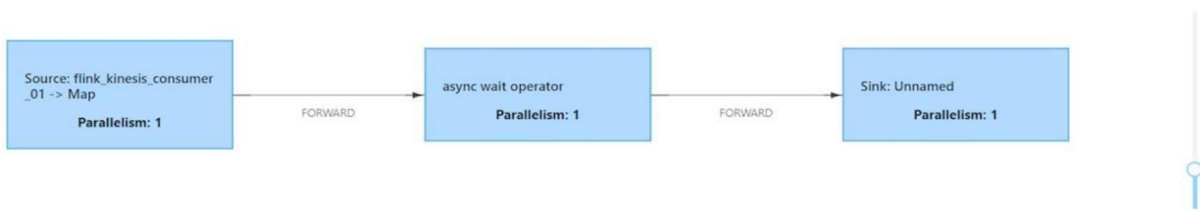
Welcome × | data.py ×

```python
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",
    "1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0",
    "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))
```

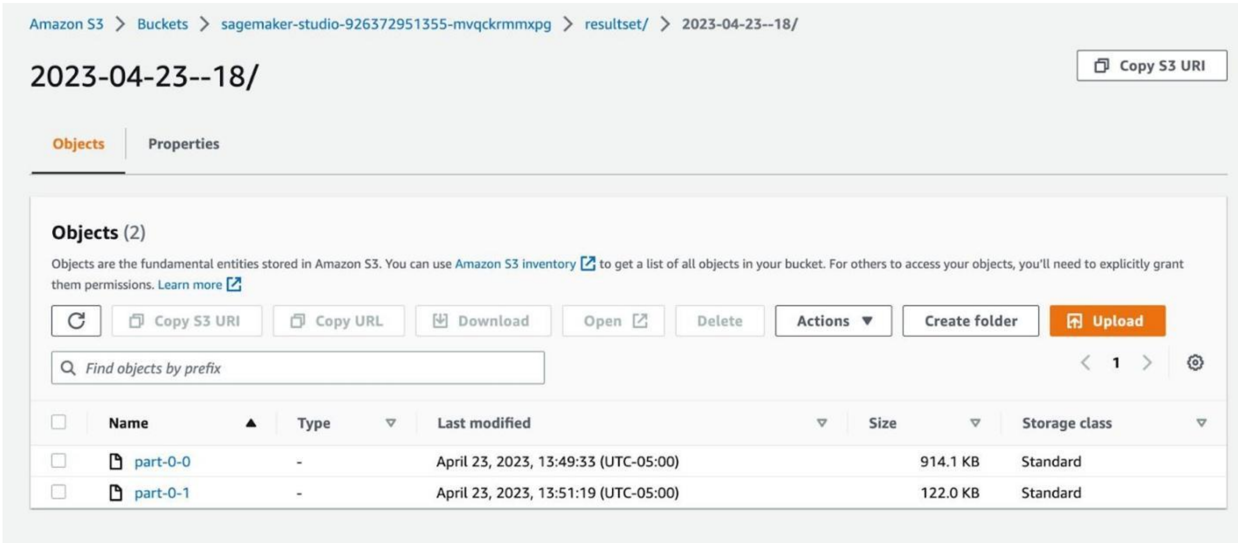
The code is executed in the terminal and after execution streaming data is generated randomly. The generation of streaming data is stopped after a while.

```
python3 - "ip-172-31-9-39 ×   Immediate   ×   ⊕
ec2-user:~/environment $ python data.py
1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0
1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0
2,13.6,-73.98812,40.748923,1,-73.9038550000001,40.887425,0.5,0.5,0.0,0.0,38.5,1320.0,1.0,0.0
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
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
2,1.5,-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,-73.9038550000001,40.887425,0.5,0.5,0.0,0.0,38.5,1320.0,1.0,0.0
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,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0
1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0
1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0
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,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0
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
2,1.5,-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
```

The next step is to choose configuration to add the code in the apache flink application. The details about each of the three jobs is shown below.

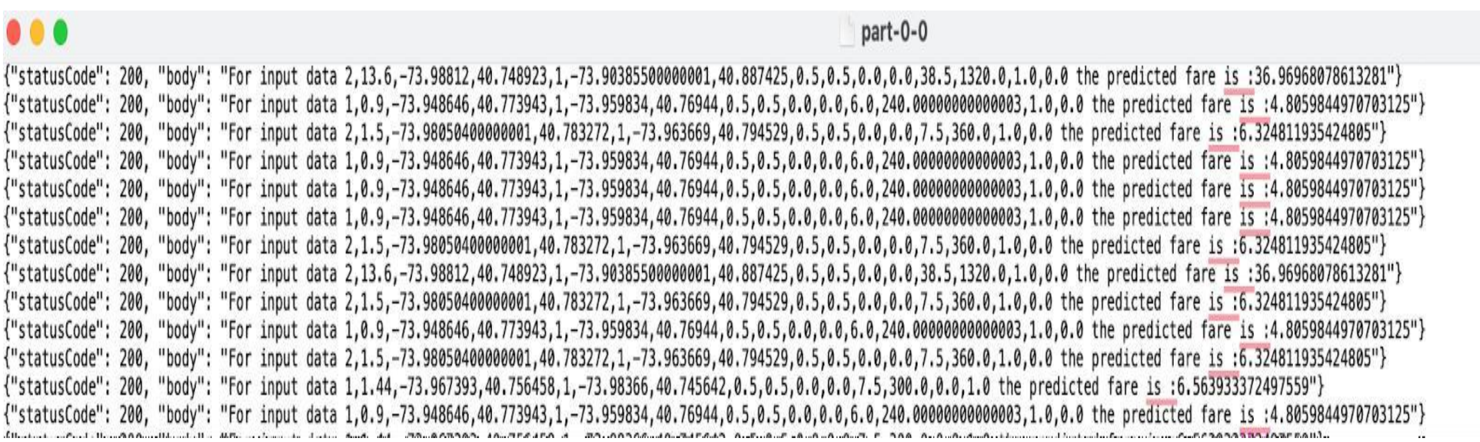


Name	Status	Bytes Received	Records Received	Bytes Sent	Records Sent	Parall	Tasks
Source: flink_kinesis_consumer_01 -> Map	RUNNING	0 B	0	0 B	0	1	1
async wait operator	RUNNING	300 B	0	0 B	0	1	1
Sink: Unnamed	RUNNING	300 B	0	0 B	0	1	1

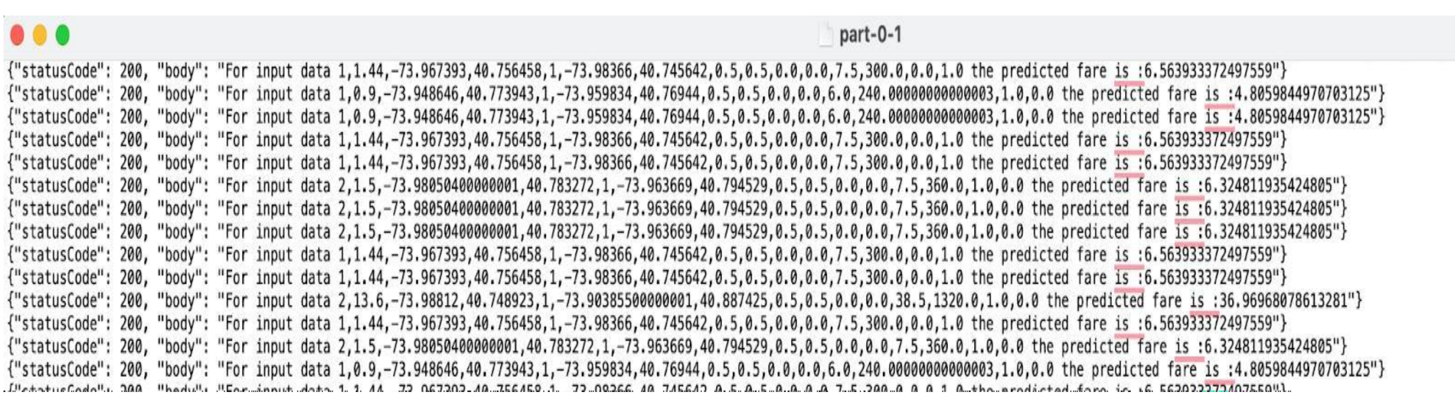


Output

The predicted fare for streaming data are computed from the model deployed at the end point and the results is being stored in s3 bucket.


part-0-0

{"statusCode": 200, "body": "For input data 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 the predicted fare is :36.96968078613281"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.324811935424805"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.324811935424805"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :36.96968078613281"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.324811935424805"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.324811935424805"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.563933372497559"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}


part-0-1

{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.563933372497559"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.563933372497559"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.563933372497559"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.324811935424805"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.324811935424805"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.324811935424805"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.563933372497559"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.563933372497559"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :36.96968078613281"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.563933372497559"}
{"statusCode": 200, "body": "For input data 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 the predicted fare is :6.324811935424805"}
{"statusCode": 200, "body": "For input data 1,0.9,-73.948646,40.773943,1,-73.959834,40.76944,0.5,0.5,0.0,0.0,6.0,240.00000000000003,1.0,0.0 the predicted fare is :4.8059844970703125"}

CONCLUSION AND FUTURE WORK:

SageMaker provides a managed environment for developing and deploying ML models, while Kinesis allows for the real-time streaming of data. By integrating the two services, businesses can create an end-to-end pipeline that ingests streaming data, performs real-time inference on the data using an ML model, and provides the results for further processing or analysis.

The use cases for this pipeline are numerous, from predicting customer churn to detecting fraud in real-time transactions. By leveraging the power of real-time ML inference on streaming data, businesses can make faster and more informed decisions, leading to better outcomes.

Overall, the combination of AWS SageMaker and Kinesis provides a highly scalable and reliable platform for real-time ML inference on streaming data, and is a valuable tool for businesses across a wide range of industries.

REFERENCES:

[1] Hassan, Fawzya & Shaheen, Masoud. (2020). Predicting Diabetes from Health-based Streaming Data using Social Media, Machine Learning and Stream Processing Technologies. International Journal of Engineering Research and Technology. 13. 1957. 10.37624/IJERT/13.8.2020.1957-1967.
[2] Hassan, Fawzya & Shaheen, Masoud & Sahal, Radhya. (2020). Real-Time Healthcare Monitoring System using Online Machine Learning and Spark Streaming. International Journal of Advanced Computer Science and Applications. 11. 10.14569/IJACSA.2020.0110977.
[3] Martín, Cristian & Langendoerfer, Peter & Doerwald, Pouya & Díaz, Manuel & Rubio, Bartolomé. (2020). Kafka-ML: connecting the data stream with ML/AI frameworks.
[4] https://catalog.us-east-1.prod.workshops.aws/
[5]https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[6]https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html
[7] https://docs.aws.amazon.com/streams/latest/dev/introduction.html
[8] https://docs.aws.amazon.com/kinesisanalytics/latest/dev/what-is.html
[9] https://aws.amazon.com/api-gateway/
[10] https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
[11] https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html