Section 2: Process Data using a Lambda function and send to Kinesis Data Firehose

In this section, we create an Amazon Kinesis Data Firehose delivery stream and populate the delivery stream using a Lambda trigger which acts as a consumer for the Kinesis Data Stream with the historic dataset of taxi trips made in NYC. The Lambda function filters out the spurious data in the incoming events and then sends the clean events to a Kinesis Data Firehose Delivery Stream in batch mode (using the PutRecordBatch API). The Lambda trigger is configured both in standard (polling) and Enhanced FanOut mode to illustrate the differences between the two.

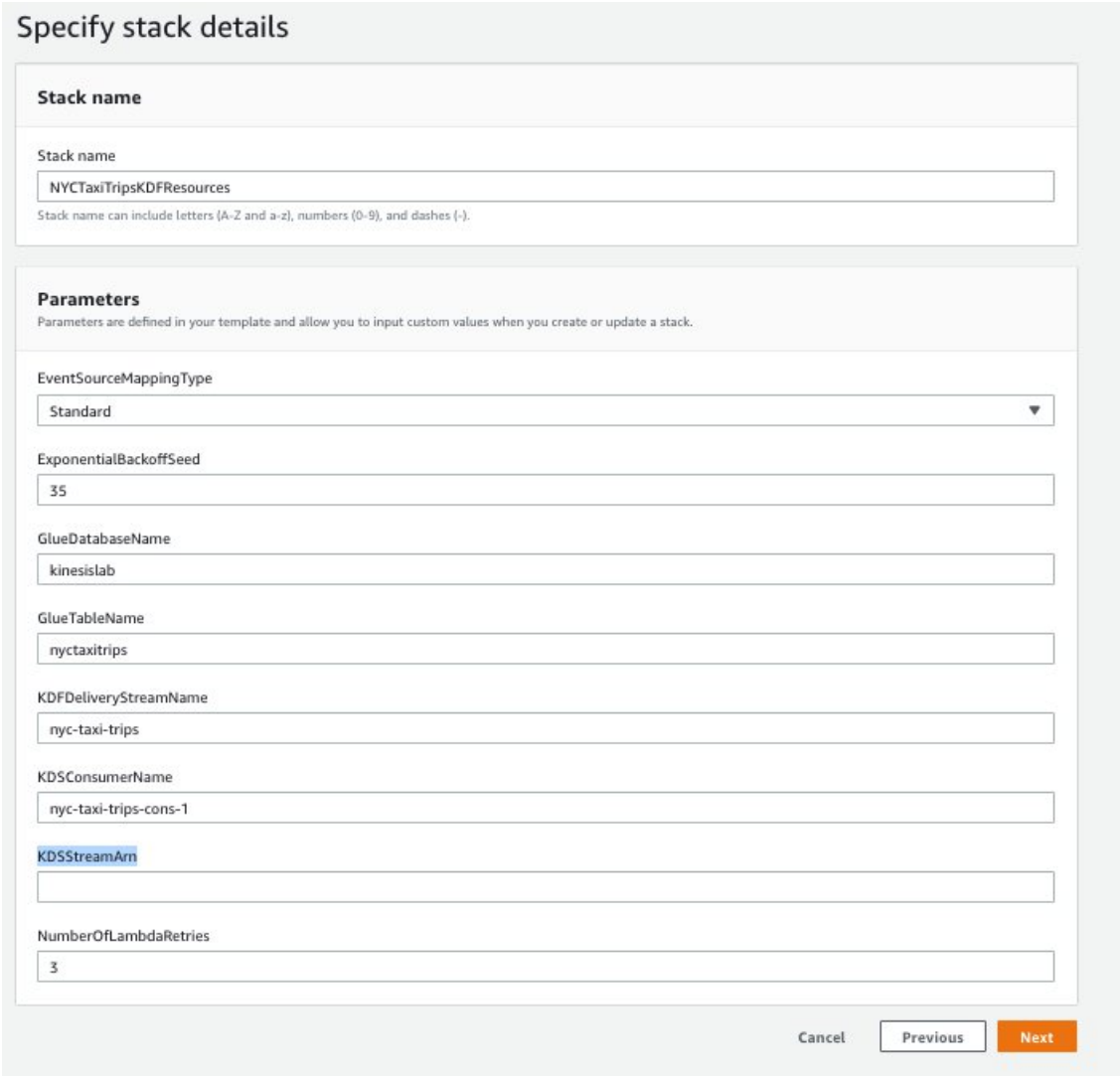
We have provided a Cloudformation template to spin up all the resources. If you prefer to set up the resources using the Console or the aws cli, please scroll down to that section in this guide.

Create Resources using a Cloudformation template

Follow this link (https://console.aws.amazon.com/cloudformation/home#/stacks/new?stackName=NYCTaxiTripsKDFResources&templateURL=https://s3.amazonaws.com/shausma-public/public/cfn-templates/kinesis-analytics-workshop/kinesis-firehose-infrastructure.yml) to execute the CloudFormation template.

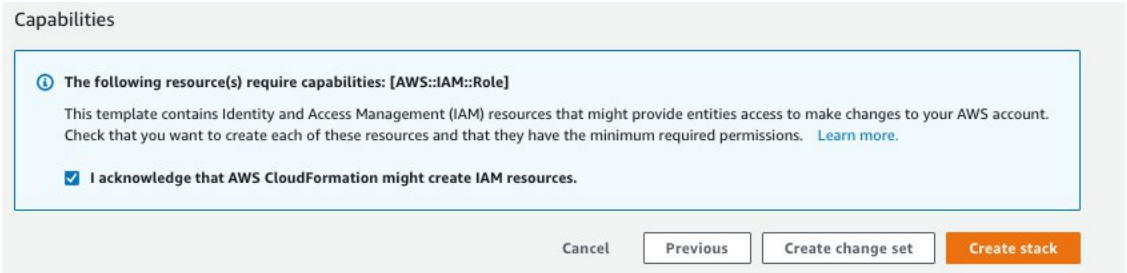
Click “Next” on the first screen.

In the next screen, copy the Stream arn of the Kinesis Data Stream from Section 1 and paste it in the “KDSStreamArn” box. Accept all other defaults and click “Next”

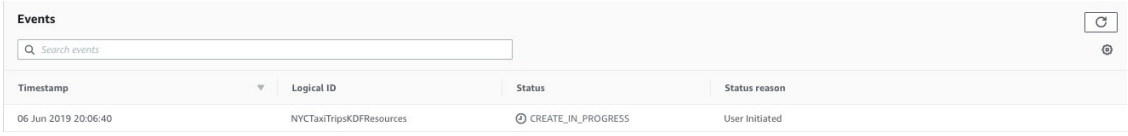


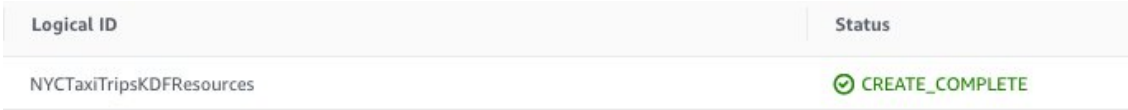
Click “Next” on the next screen.

In the next screen, scroll down and click on the checkbox in the “Capabilities” section and click “Create stack”



Wait for the stack to get created





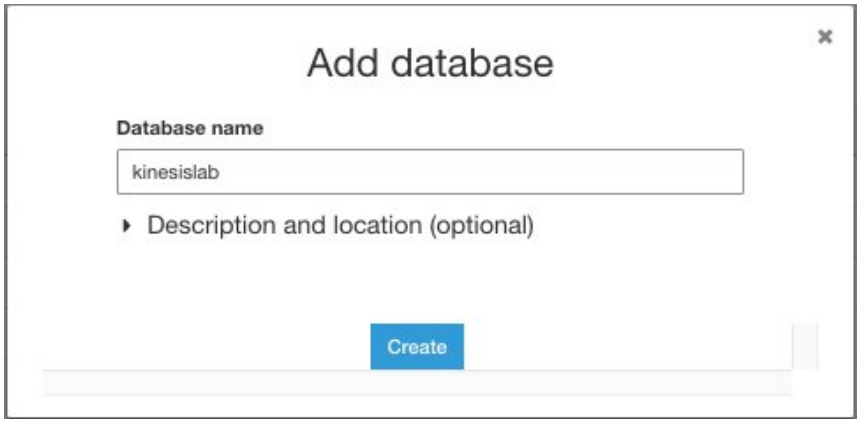
Once the stack compltes creation, look into the resources in the outputs tab, and go to the respective consoles and look at the resources created. Go to the "When all resources are created" section in this guide.

Create Resources using the Console and AWS CLI

Before we create the Kinesis Data Firehose delivery stream, we need to create some prerequisites that the delivery stream references.

Create the Glue database:

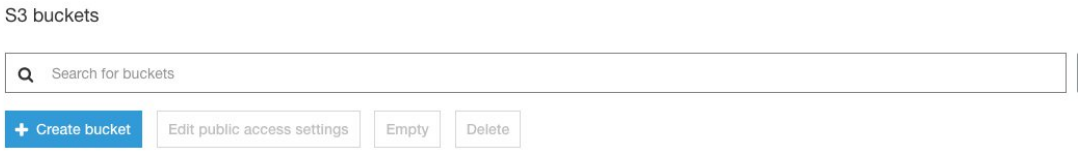
Go to the Glue console, click on "Databases" in the left pane and then click on "Add database". Type in a name for the database and click on "Create". This database will be used later to create an external table using the Athena console to provide a schema for data format conversion when creating a delivery stream using the Kinesis Data Firehose console.



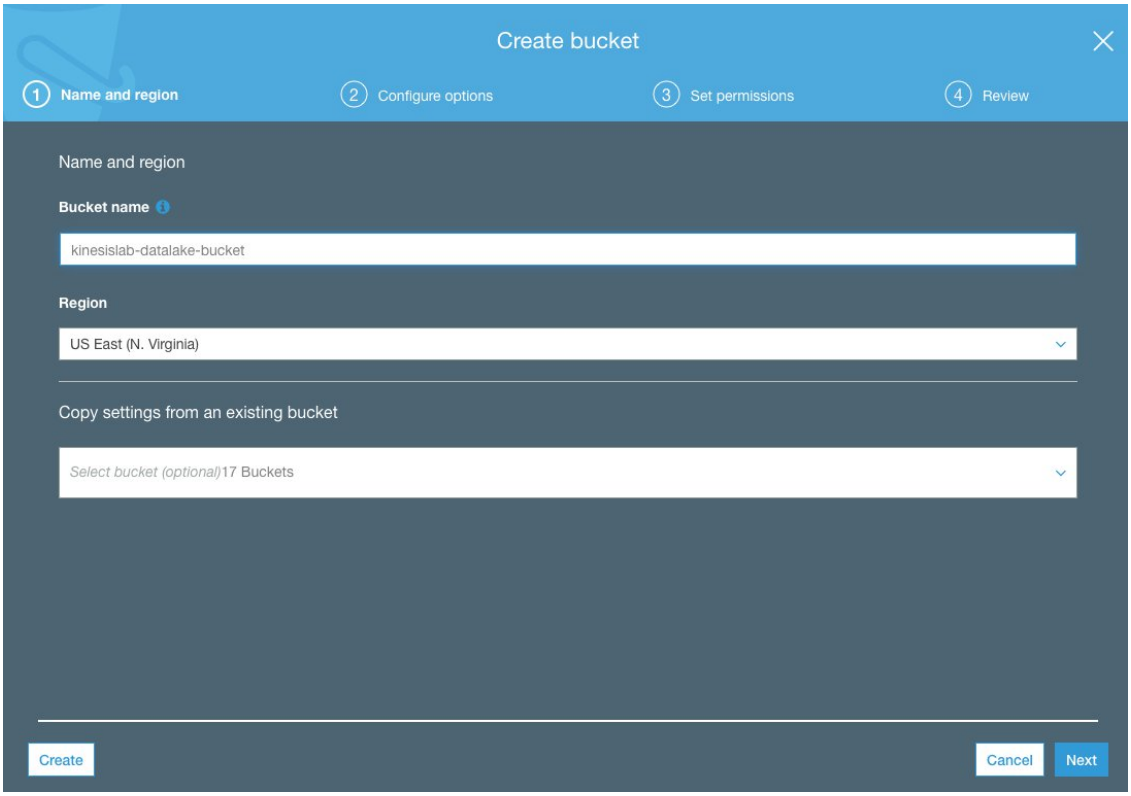
AWS CLI: aws glue create-database --database-input '{"Name": "kinesislab"}'

Create the S3 bucket

Go to the S3 console, click on "Create bucket". This bucket will be used to store the data delivered to S3 via the Kinesis Data Firehose delivery stream.



Type in the Bucket name, specify the Region and click on "Create".



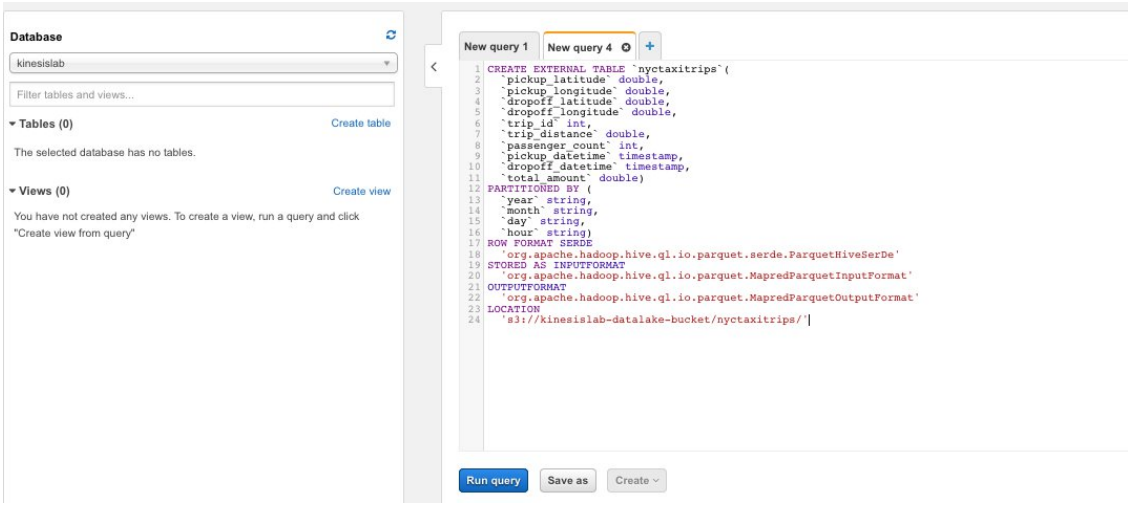
AWS CLI: aws s3 mb s3://kinesislab-datalake-bucket --region us-east-1

Create the external table that Kinesis Data Firehose will use as a schema for data format conversion

Go to the Athena console, in the left pane, under "Database", click on the dropdown and select the database that you created earlier through the Glue console. Paste the following sql statement in the query window and click on "Run query". Remember to replace the S3 bucket name with the name of the bucket you created earlier through the S3 console.

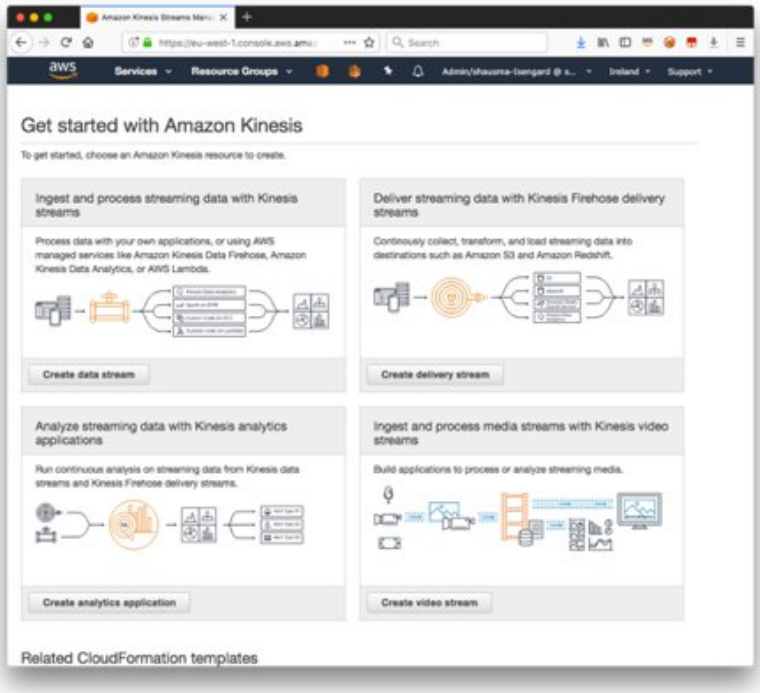
```
CREATE EXTERNAL TABLE `nyctaxitrips`(
  `pickup_latitude` double,
  `pickup_longitude` double,
  `dropoff_latitude` double,
  `dropoff_longitude` double,
  `trip_id` int,
  `trip_distance` double,
  `passenger_count` int,
  `pickup_datetime` timestamp,
```

```
  `dropoff_datetime` timestamp,
  `total_amount` double)
 PARTITIONED BY (
  `year` string,
  `month` string,
  `day` string,
  `hour` string)
 ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSe
rDe'
 STORED AS INPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputF
ormat'
 OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutput
Format'
 LOCATION
  's3://kinesislab-datalake-bucket/nyctaxitrips/'
```

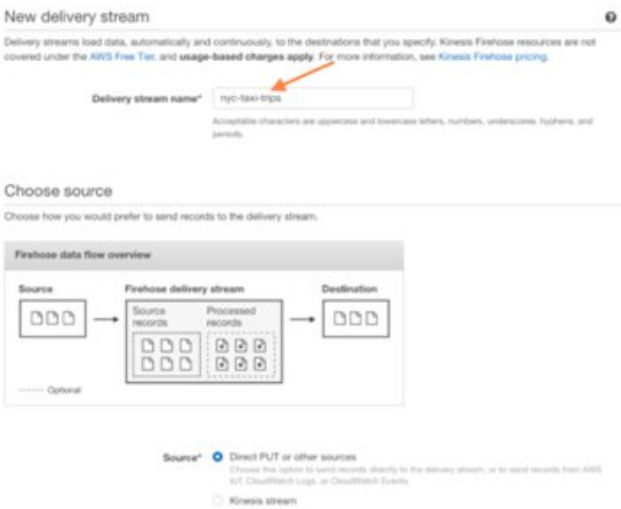


Create Kinesis Data Firehose delivery Stream

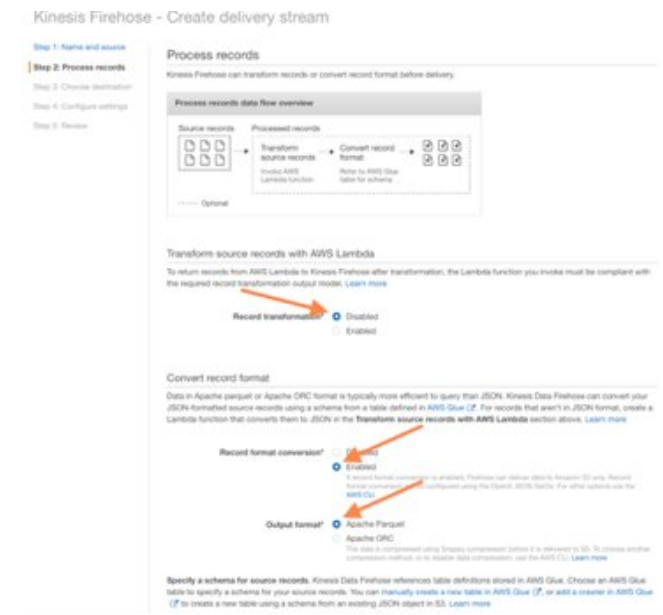
Navigate to the Amazon Kinesis services and press **Get Started** when prompted (you may not need to complete this, if you have already used Amazon Kinesis). Select **Create delivery stream** to navigate to the Amazon Kinesis Data Firehose service.



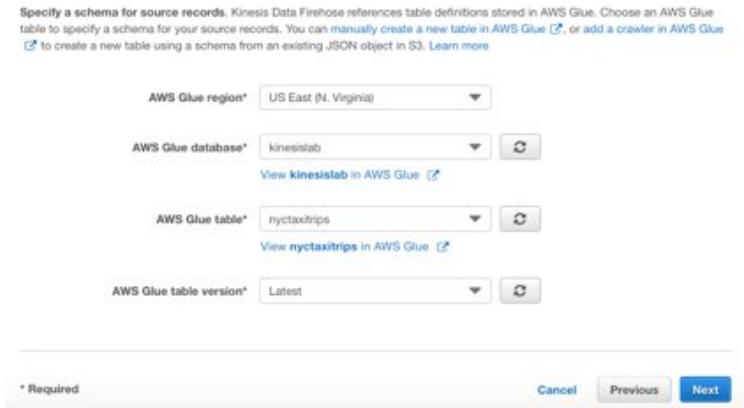
Enter a unique name for the Delivery stream name, eg, *nyc-taxi-trips.* For "Source", choose "Direct PUT or other sources" as a lambda function would be used to feed events to the Kinesis Data Firehose delivery stream. Click **Next.**



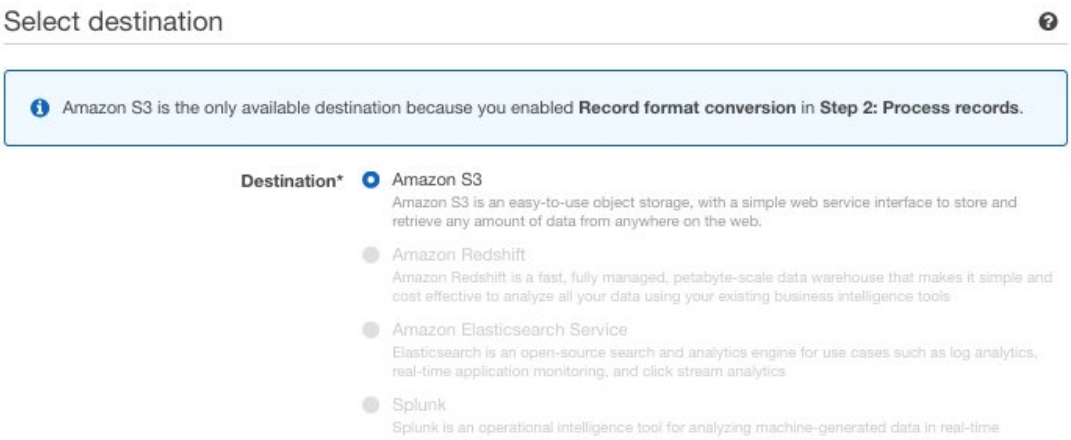
Choose "Record Transformation" "as "Disabled" and "Record format conversion" as "Enabled" and choose "Output format" as "Apache Parquet".



Choose the schema to use for data format conversion as illustrated in the screenshot below.



Select the S3 destination. I chose "S3".



Click on the dropdown next to "S3 bucket" and select the bucket that you created earlier through the S3 console. For "S3 prefix", copy and paste the following:

nyctaxitrips/year=!{timestamp:YYYY}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/

For "S3 error prefix" copy and paste the following:

nyctaxitripserror/!{firehose:error-output-type}/year=!{timestamp:YYYY}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/

For "Source record S3 backup", click on "Disabled". Click on "Next".

S3 bucket* kinesislab-datalake-bucket Create new

View **kinesislab-datalake-bucket** in S3 console

S3 prefix

By default, Kinesis Data Firehose appends the prefix "YYYY/MM/DD/HH" (in UTC) to the data it delivers to Amazon S3. You can override this default by specifying a custom prefix that includes expressions that are evaluated at runtime.

If your custom prefix doesn't include expressions, Kinesis Data Firehose uses your prefix and appends "YYYY/MM/DD/HH". If your custom prefix includes a Firehose random string or timestamp expression, Kinesis Data Firehose doesn't append "YYYY/MM/DD/HH".Learn more

Prefix

```
nyctaxitrips/year=!{timestamp:YYYY}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/
```

S3 error prefix

You can specify an S3 bucket prefix to be used in error conditions. This prefix can include expressions for Kinesis Data Firehose to evaluate at runtime. Learn more about the rules for specifying prefix expressions

Error prefix

```
nyctaxitripserror/!{firehose:error-output-type}/year=!{timestamp:YYYY}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/
```

S3 backup

Enabling source record backup ensures that source records can be recovered if record processing transformation does not produce the desired results. Learn more

Source record S3 backup* Disabled / Enabled

* Required Cancel Previous Next

Specify the **buffering hints** for the Amazon S3 destination. Type in 128 MB and 300 seconds.

S3 buffer conditions

Firehose buffers incoming records before delivering them to your S3 bucket. Record delivery will be triggered once either of these conditions has been satisfied. Learn more

Because of the high compression ratios that you typically get when converting JSON records to Parquet or ORC format, we recommend a buffer size value of 128 MB. Specifying a smaller buffer size can result in the delivery of very small S3 objects, which are less efficient to query.

Buffer size* 128 MB

Specify a buffer size between 64-128 MB

Buffer interval* 300 seconds

Specify a buffer interval between 60-900 seconds

Keep the default settings for "S3 compression and encryption".

S3 compression and encryption

Firehose can compress records before delivering them to your S3 bucket. Compressed records can also be encrypted in the S3 bucket using a KMS master key. Learn more

You don't need to enable compression. Snappy compression was enabled automatically when you chose **Record format conversion** in **Step 2: Process records**. To choose another compression method, or to disable data compression, use the AWS CLI. Learn more

S3 compression*
- Disabled
- GZIP
- Snappy
- Zip

S3 encryption*
- Disabled
- Enabled

Choose "Enabled" for "Error logging".

Error logging

Firehose can log record delivery errors to CloudWatch Logs. If enabled, a CloudWatch log group and corresponding log streams are created on your behalf. Learn more

Error logging* Disabled
Enabled

Tags (optional)

You can add tags to organize your AWS resources, track costs, and control access. Learn more

Key | Value - *optional*

Enter key | *Enter value* | **Remove tag**

Add tag

You can add 49 more tag(s)

Specify the IAM role that you want Kinesis Data Firehose to assume to access resources on your behalf. Choose either **Create new** or **Choose** to display a new screen. Choose **Create a new IAM role**, name the role, and then choose **Allow**.

IAM role

Firehose uses an IAM role to access your specified resources, such as the S3 bucket and KMS key. Learn more

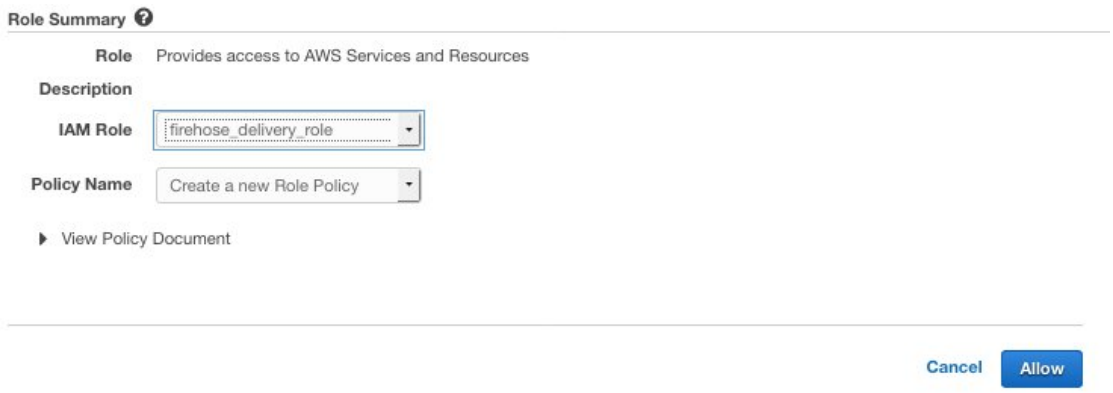
IAM role* **Create new or choose**

IAM role

Firehose uses an IAM role to access your specified resources, such as the S3 bucket and KMS key. Learn more

IAM role* **Create new or choose**

Choose **Create Delivery Stream**.



Copy and paste the json below into a file and use it to create the delivery stream using the AWS CLI.
AWS CLI:

- aws firehose create-delivery-stream --cli-input-json file://createdeliverystream.json .

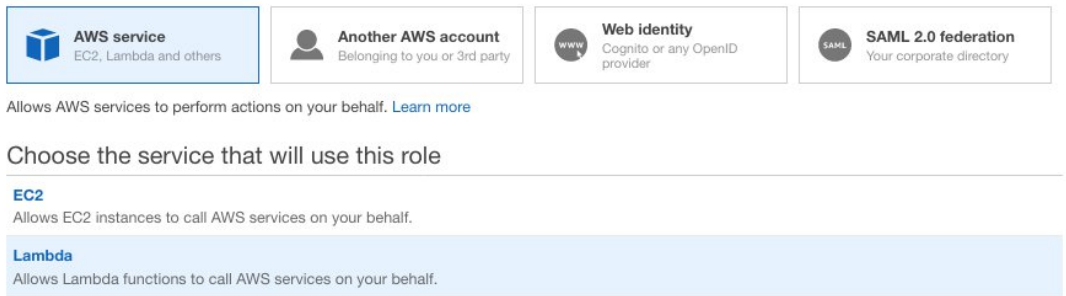
createdeliverystream.json

```
{
    "DeliveryStreamName": "nyc-taxi-trips",
    "DeliveryStreamType": "DirectPut",
    "ExtendedS3DestinationConfiguration": {
        "RoleARN": "arn:aws:iam::<account-
id>:role/firehose_delivery_role",
        "BucketARN": "arn:aws:s3:::kinesislab-datalake-
bucket",
        "Prefix":
"nyctaxitrips/year=!{timestamp:YYYY}/month=!{timestamp:MM}/
day=!{timestamp:dd}/hour=!{timestamp:HH}/",
        "ErrorOutputPrefix":
"nyctaxitripserror/!{firehose:error-output-
type}/year=!{timestamp:YYYY}/month=!{timestamp:MM}/day=!{ti
mestamp:dd}/hour=!{timestamp:HH}/",
        "BufferingHints": {
            "SizeInMBs": 128,
            "IntervalInSeconds": 300
        },
        "CompressionFormat": "UNCOMPRESSED",
        "EncryptionConfiguration": {
            "NoEncryptionConfig": "NoEncryption"
        },
        "CloudWatchLoggingOptions": {
            "Enabled": true,
```

```
            "LogGroupName": "KDF-NYCTaxiTrips",
            "LogStreamName": "S3Delivery"
        },
        "S3BackupMode": "Disabled",
        "DataFormatConversionConfiguration": {
            "SchemaConfiguration": {
                "RoleARN": "arn:aws:iam::<account-
id>:role/firehose_delivery_role",
                "DatabaseName": "kinesislab",
                "TableName": "nyctaxitrips",
                "Region": "us-east-1",
                "VersionId": "LATEST"
            },
            "InputFormatConfiguration": {
                "Deserializer": {
                    "OpenXJsonSerDe": {}
                }
            },
            "OutputFormatConfiguration": {
                "Serializer": {
                    "ParquetSerDe": {}
                }
            },
            "Enabled": true
        }
    }
}
```

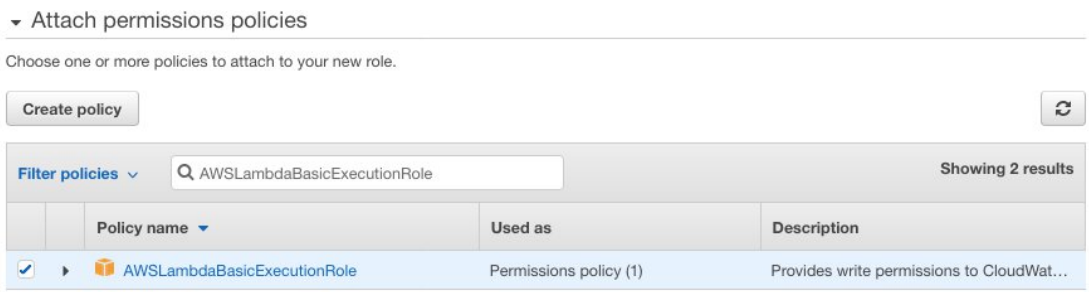
Create the IAM role to use with the Lambda function

Go to the IAM console, click on "Roles" and the click "Create role". Click on "AWS service" and then click on "Lambda". Change the name of the Kinesis Data Stream if you created a stream named different than “nyc-taxi-trips”.

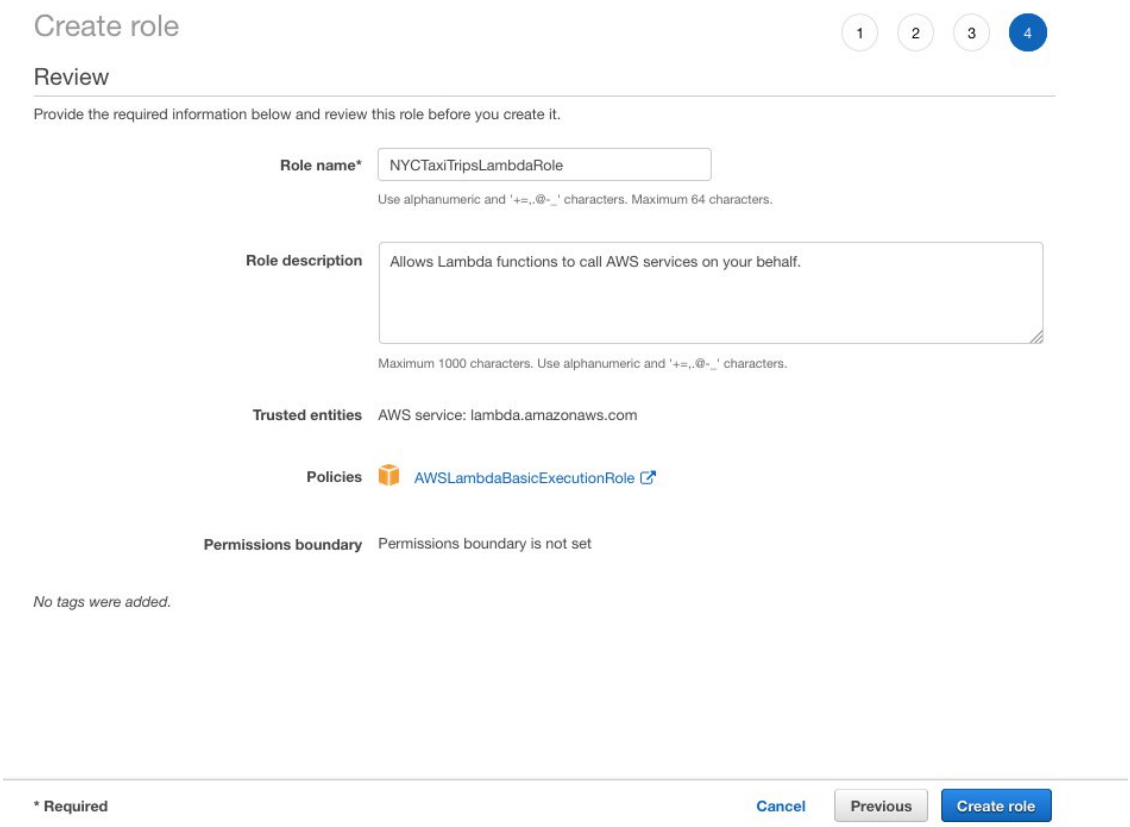


Click on "Next:Permissions"

In the search box next to "Filter policies", type "AWSLambdaBasicExecutionRole" and select the checkbox next to "AWSLambdaBasicExecutionRole". Click on "Next:Tags" and add tags if you wish. Click "Next:Review"



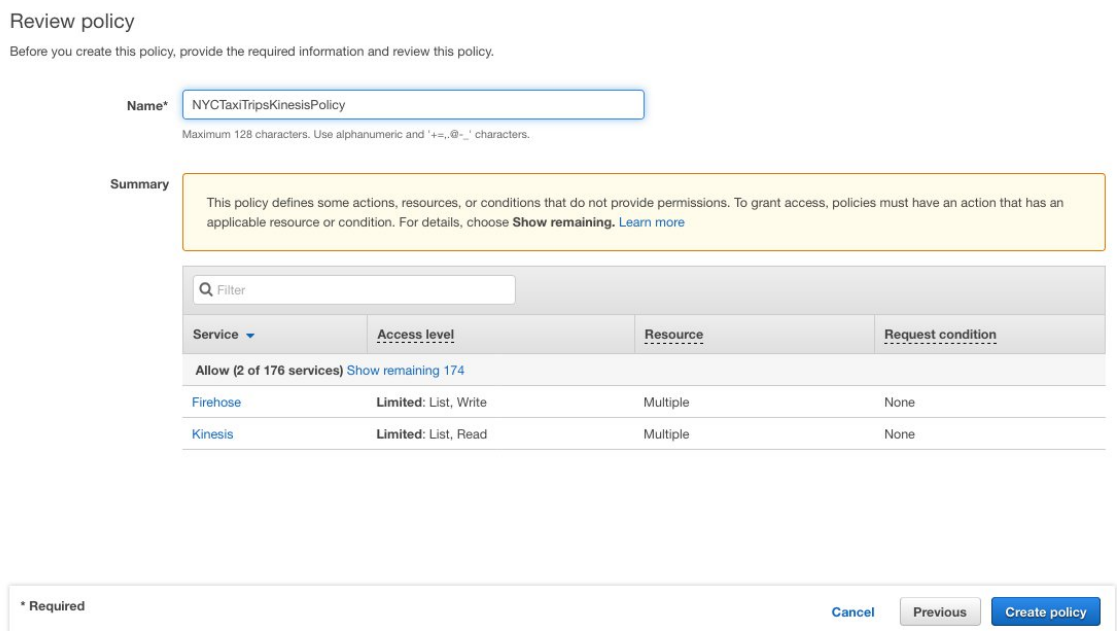
Fill in the details as show below and click on "Create role".



Click on the newly create role. Then click on "Add inline policy". Click on the "JSON" tab. Then copy and paste the json from the KinesisPolicy.json file below (remember

to change the account-id to your account id and the stream name and delivery stream name to your streams). Then click on "Review policy".

Fill in the details as shown below and click on "Create policy".



The Lambda role is now created.

AWS CLI:

TrustPolicyForLambda.json

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Principal": {
            "Service": "lambda.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
    }]
}
```

- aws iam create-role --role-name NYCTaxiTripsLambdaRole --assume-role-policy-document file://TrustPolicyForLambda.json

- aws iam attach-role-policy --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole --role-name NYCTaxiTripsLambdaRole
- aws iam put-role-policy --role-name NYCTaxiTripsLambdaRole --policy-name NYCTaxiTripsKinesisPolicy --policy-document file://KinesisPolicy.json

KinesisPolicy.json:

```
{
    "Version": "2012-10-17",
    "Statement": [{
            "Sid": "Effect: ",
            "Effect": "Allow",
            "Action": [
                "kinesis:GetShardIterator",
                "kinesis:GetRecords",
                "firehose:PutRecordBatch",
                "kinesis:DescribeStream"
            ],
            "Resource": [
                "arn:aws:kinesis:us-east-
1:<accountid>:stream/nyc-taxi-trips",
                "arn:aws:firehose:us-east-1:
:<accountid>:deliverystream/nyc-taxi-trips"
            ]
        },
        {
            "Sid": "KinesisPerm2",
            "Effect": "Allow",
            "Action": [
                "kinesis:ListStreams",
                "kinesis:SubscribeToShard",
                "kinesis:DescribeStreamSummary",
                "firehose:ListDeliveryStreams"
            ],
            "Resource": "*"
        },
        {
            "Sid": "KinesisPerm3",
            "Effect": "Allow",
            "Action": "kinesis:ListShards",
            "Resource": [
                "arn:aws:kinesis:us-east-1:
:<accountid>:stream/nyc-taxi-trips"
            ]
        }
```

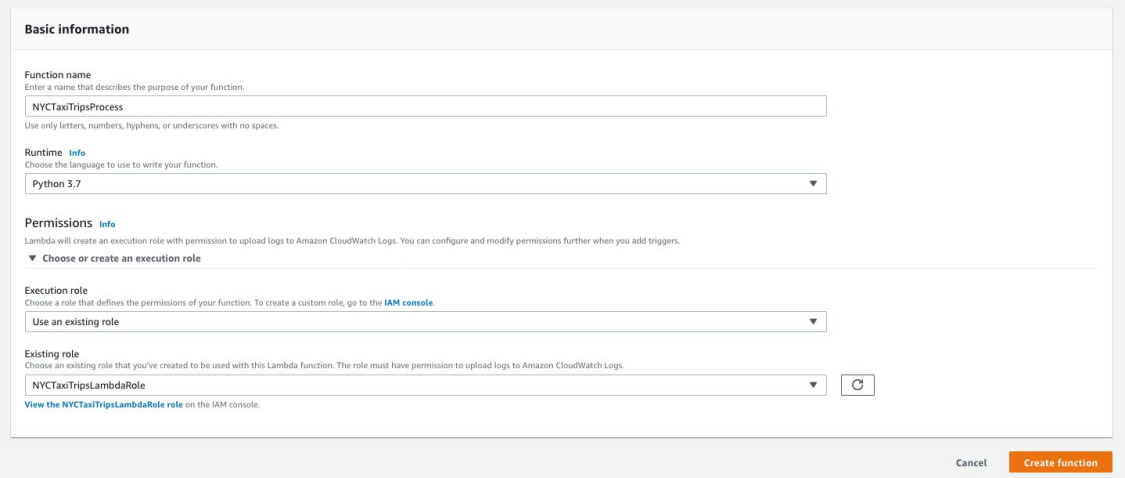
```
        ]
}
```

Create the Lambda function to process records from the Kinesis Data Stream

The Lambda function does a few things:

1. It inspects the incoming message for unclean records with missing fields and filters them out.
2. It tries to send the clean records to Kinesis Data Firehose.
3. If it receives a throttling error, it determines if all the records received failed or some records failed.
    a. If all records failed, it raises an exception, so the Lambda service can retry with the same payload (the service keeps retrying with the same payload until it receives a success). It also logs the corresponding trip ids so you can check to see if they actually made it through to the S3 bucket.
    b. If some records failed, it retries based on the configured retries environment variable with exponential backoff. If any of the retries are successful, it returns a success. If none of the retries are successful, it raises an exception so the Lambda service can retry with the same payload. Note that in this case, there could be duplicate records sent. You can increase the number of retries or save the records somewhere to process later and move on as alternate strategies to prevent duplicates.

Go to the Lambda console and click on "Create function". Select "Author from scratch" and fill in the details as show below and click on "Create function".



Copy and Paste the following code in the "Function code" window overwriting the template code. Then click "Save".

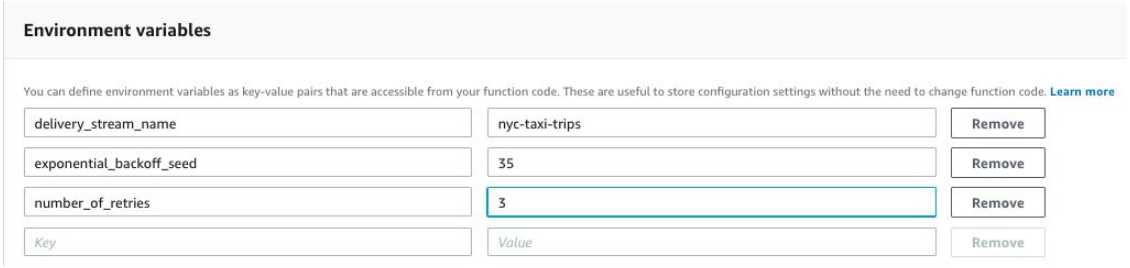
```
import base64
from datetime import datetime
import time
import json
import boto3
import random
import uuid
import os

client = boto3.client('firehose')


def check_data(data):
    payload = json.loads(data)

    if payload['type'] == "trip" and
payload['pickup_longitude'] != 0 and
payload['pickup_latitude'] != 0 and
payload['dropoff_latitude'] != 0 and
payload['dropoff_longitude'] != 0:
        return True
    else:
        return False

def gen_retry_output_list(resp, outputRecList):
    recCount = 0
    retryOutputList = []
    for respRec in resp['RequestResponses']:
        try:
            respError = respRec['ErrorCode']
            if respError == "ServiceUnavailableException":
                retryOutputList.append(outputRecList[recCou
nt])
        except KeyError:
            pass
        recCount += 1

    return retryOutputList

def get_sleep_time(retryCount, exponentialBackoff, seed):

    if (exponentialBackoff == True):
        return (2*(seed**retryCount))/1000
```

```
    else:
        return (500/1000)

def lambda_handler(event, context):
    print('Loading function' + ' '
+  datetime.now().strftime('%Y-%m-%dT%H:%M:%S.%fZ'))
    print('Processing {}
record(s).'.format(len(event['Records'])))
    output = {}
    outputRecList = []
    retryOutputList = []
    numRetries = int(os.environ['number_of_retries'])
    retryCount = 1
    eventRecords = len(event['Records'])
    tripIds = []
    deliveryStreamName = os.environ['delivery_stream_name']
    seed = int(os.environ['exponential_backoff_seed'])
    exponentialBackoff = False

    for record in event['Records']:
        recordData =
base64.b64decode(record['kinesis']['data'])
        recordDataJson = json.loads(recordData)
        if check_data(recordData):
            output['Data'] = recordData
            outputRecList.append(output)
            output = {}
            tripIds.append(recordDataJson['trip_id'])

    if len(outputRecList) > 0:

        resp =
client.put_record_batch(DeliveryStreamName=deliveryStreamNa
me,
                   Records=outputRecList
            )
    else:
        print("No records to send ...")
        return {
            'statusCode': 200,
            'body': json.dumps('Lambda successful!')
        }


    if resp['FailedPutCount'] != 0:
        print('Failed to process {}
record(s).'.format(resp['FailedPutCount']))
```

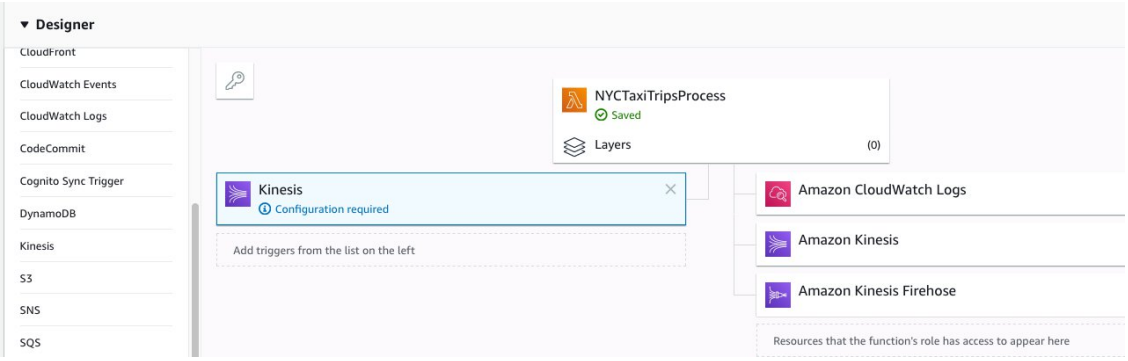
```
        if resp['FailedPutCount'] != eventRecords:

            while (retryCount <= numRetries):

                print("Retrying {} failed records up to {}
times with exponential
backoff...".format(resp['FailedPutCount'], numRetries -
(retryCount - 1)))
                retryOutputList =
gen_retry_output_list(resp, outputRecList)
                if len(retryOutputList) > 0:
                    exponentialBackoff = True
                print("Backing Off for {} seconds
...".format(get_sleep_time(retryCount, exponentialBackoff,
seed)))
                time.sleep(get_sleep_time(retryCount,
exponentialBackoff, seed))

                retryResp =
client.put_record_batch(DeliveryStreamName=deliveryStreamNa
me,
                    Records=retryOutputList
                )

                if retryResp['FailedPutCount'] == 0:
                    print("Retry successful after {} tries
...".format(retryCount))

                    return {
                            'statusCode': 200,
                            'body': json.dumps('Lambda
successful!')
                    }

                retryCount += 1
                outputRecList = retryOutputList
                retryOutputList = []
                resp = retryResp
                print(resp['RequestResponses'])

            print("All retries unsuccessful. Letting Lambda
retry but there could be duplicates ...")
            raise Exception("Records could not be sent.
Lambda to retry ...")
        else:
            print("Since all records failed, letting Lambda
```

```
retry..")
            print(tripIds)
            print(resp['RequestResponses'])
            raise Exception("Records could not be sent.
Lambda to retry ...")
    else:
         print("Records successfully sent ...")
         return {
            'statusCode': 200,
            'body': json.dumps('Lambda successful!')
        }
```

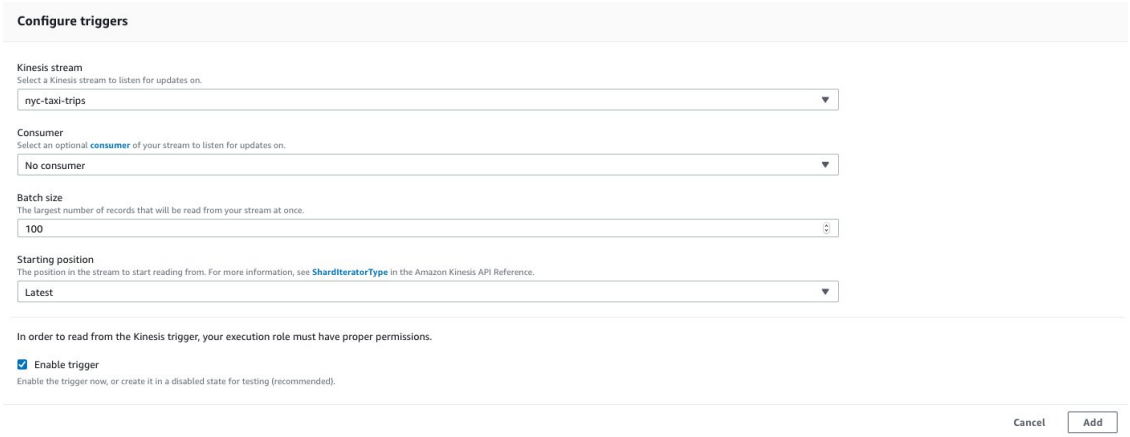
Scroll down to the “Environment variables” section and fill it out as show below:



Go to the Triggers pane in the top left of the page, scroll down and click on "Kinesis".



Scroll down to the "Configure triggers" section and fill in the details as shown below. First we will setup the trigger in "Standard" mode which is with "No consumer". Click "Add". Then scroll up and click "Save".

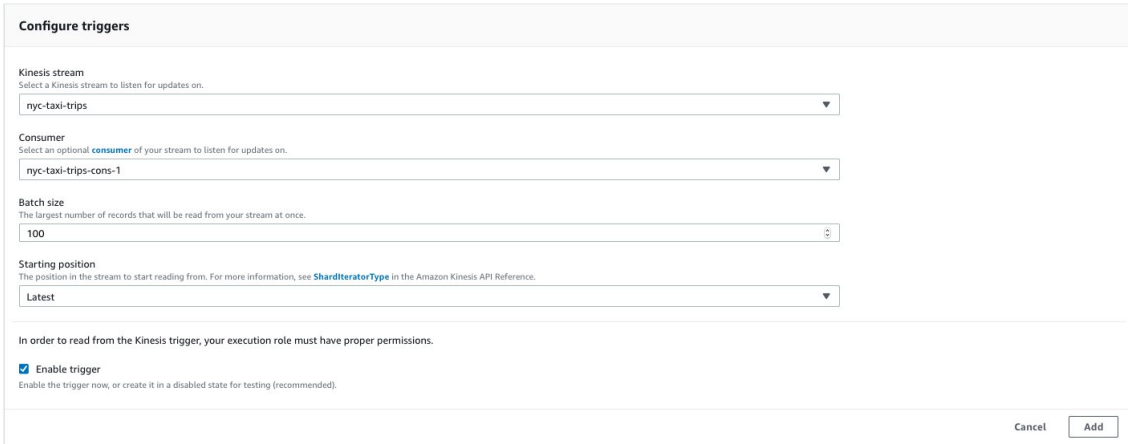


In order to setup the Lambda trigger in "Enhanced FanOut (EFO)" mode, first register a consumer using the AWS CLI. This cannot be done via the console.

AWS CLI:

- aws kinesis register-stream-consumer --stream-arn <your-kinesis-stream-arn> --consumer-name nyc-taxi-trips-cons-1

Then come back to the Lambda console, go to the Triggers pane in the top left of the page, scroll down and click on "Kinesis" as before. Scroll down to the "Configure triggers" section and fill in the details as shown below. In this case, click on the "Consumer" dropdown and select the consumer you just created in the AWS CLI. Click "Add". Then scroll up and click "Save".



AWS CLI:

- Save the Lambda function code in a “lambda_function.py” file and zip it up into a lambda_function.zip file using: zip lambda_function.zip lambda_function.py
- aws lambda create-function --function-name NYCTaxiTrips --runtime python3.7 --role arn:aws:iam::<account-id>:role/NYCTaxiTripsLambdaRole --handler lambda_function.lambda_handler --zip-file "fileb://lambda_function.zip"

For standard event source mapping:

AWS CLI:

- aws lambda create-event-source-mapping --event-source-arn <stream-arn> --function-name NYCTaxiTripsProcess --starting-position LATEST

To get the stream-arn use:

- aws kinesis describe-stream –stream-name nyc-taxi-trips

For EFO event source mapping:

AWS CLI:

- aws lambda create-event-source-mapping --event-source-arn <consumer-arn> --function-name NYCTaxiTripsProcess --starting-position LATEST

To get the consumer-arn use:

- aws kinesis list-stream-consumers --stream-arn <stream-arn for the nyc-taxi-trips stream>

The Lambda function that will process the records from the Kinesis Data stream is now created and the event source mapping is also created.

When all resources are created

Start sending data to the Kinesis Data Streams stream created in Section 1 as outlined in Section 1.

Once the Lambda function starts processing (note that it will process from the tip of the stream as the starting position is set to LATEST), the Kinesis Data Firehose delivery stream you created will ingest the records, buffer it, transform it to parquet and deliver it to the S3 destination under the prefix provided. Go to the S3 console and navigate to the bucket and prefix and locate the files.

You can also navigate to Amazon Cloudwatch Logs to look at the output of the Kinesis Data Streams trigger Lambda function. To do that, go to the Lambda console, click on the function you created, click on the “Monitoring” tab, then click on “View logs in CloudWatch”. This should take you to the Cloudwatch Logs console and you

can see the log streams. Click on any one of them and scroll down to observe the lambda function execution outputs. In particular, look for the instances of throttling errors received from Kinesis Data Firehose and how the function handles them.