Cloud Computing Final Project

Create a data processing pipeline with AWS Kinesis

Chih-Hua Chang

January 20, 2023

1 Introduction

1.1 Objective

The objective of the project is to build a data processing pipeline with Amazon Kinesis.
The National Oceanic and Atmospheric Administration (NOAA) Climate Data contains weather records in the US. In the project, the daily weather parameters in "GHCND" data set including precipitation, snowfall, and min/max temperature, in Maryland state from October 1, 2021 to October 31, 2021 will be requested from NOAA Climate Data REST API. The records will be extract to a JSON file which contains the date/time, station, location, datatype, and value information. The records will be insert to AWS Kinesis by a EC2 producer and retrieved by a Lambda consumer. The Lambda consumer will remove records from Kinesis, and send the records to two DynamoDB tables, Precipitation table and Temperature table, based on the datatype in the records. The information in the DynamoDB table will include the timestamp, the weather station, and the value. The information in DynamoDB can be retrieved by location, and sorted by timestamp value.

1.2 Services Introduction

The project leverages four Amazon Web Services in total, including Amazon Elastic Compute Cloud, Amazon Kinesis, Amazon Lambda, and Amazon DynamoDB, to create the data processing pipeline. Below is a brief description of each service and how they will be applied in the project.

1.2.1 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud, which is also named Amazon EC2, provides scalable computing capacity (virtual machines) in the Cloud. In the project, EC2 will be used as a producer to send/receive NOAA REST API requests/ responses via HTTP/HTTPs, parse and build JSON files, and put JSON files to Amazon Kinesis.

1.2.2 Amazon Kinesis

Amazon Kinesis is a service for real-time data streaming. It will allow us to process and analyze data as it arrives and respond instantly. In the project, the weather records will be sent to the Kinesis Data Stream by EC2 and the records will be processed and sent to the DynamoDB by Lambda.

1.2.3 Amazon Lambda

Amazon Lambda is a serverless, event-driven compute service that enables us to run code virtually without provisioning or managing servers. In the project, Lambda will be triggered by Kinesis, and the records will be sent to DynamoDB with the Lambda Consumer Code.

1.2.4 Amazon DynamoDB

Amazon DynamoDB is a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale. In our project, the records processed by lambda will be sent to two DynamoDB tables based on the datatype of the records, including temperature and precipitation. The partition key is the stationid and the sort key is the date.

1.3 Data Processing Pipeline

In the project, the weather records from NOAA Climate Data Online REST API will be requested by the EC2 producer and be sent to the Kinesis Data Stream. In the meantime, the Lambda consumer will be triggered by Kinesis and put the records from the Kinesis Data Stream to DynamoDB based on the datatype.

Figure 1 is the AWS Data Processing Pipeline flow chart.

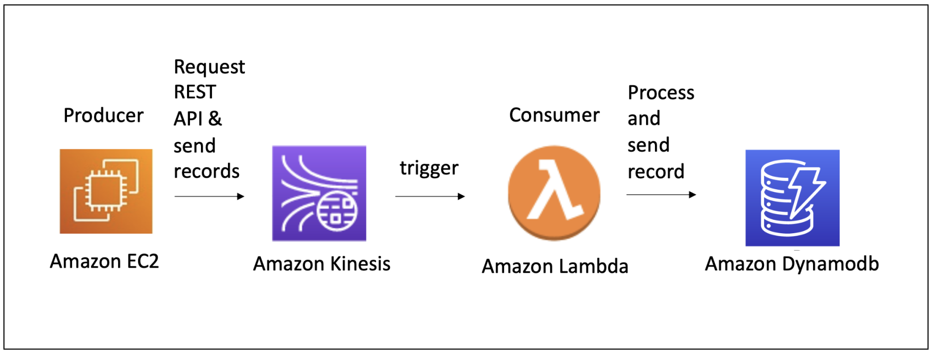


Figure 1: Data processing pipeline

2 Services Configuration

In this section, I am going to run through all the services and the configuration settings details with images references.

2.1 Create Kinesis

Create a Kinesis Data Stream, which will receive and temporarily hold the records sent from the EC2. The name of the Kinesis Data Stream is "maryland_weather_record", and the shard is provisioned to 1.

Figure 2 is the Kinesis configuration information.

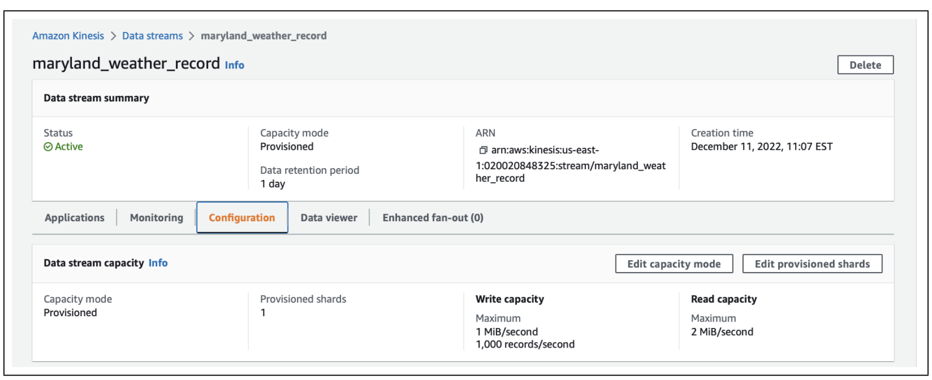


Figure 2: Kinesis Configuration

2.2 Create Dynamodb

Create two Dynamodb tables, one for the precipitation and snow records, and the other for the temperature records.

The first Dynamodb table is named "Precipitation". The partition key is "stationid" and the sort key is "date".

Figure 3 is the configuration of the "Precipitation" DynamoDB table.

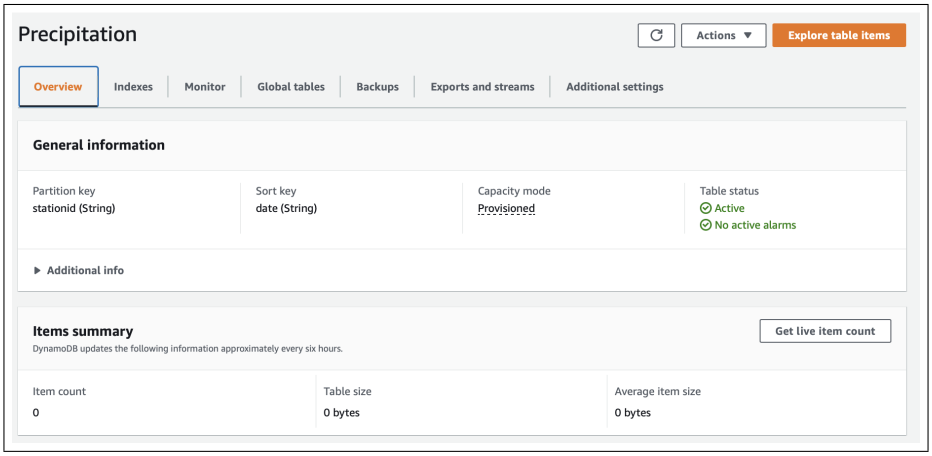


Figure 3: Dynamodb Precipitation Table Cinfiguration

The second Dynamodb table is named "Temperature". The partition key is "stationid" and the sort key is "date".

Figure 3 is the configuration of the "Temperature" DynamoDB table.

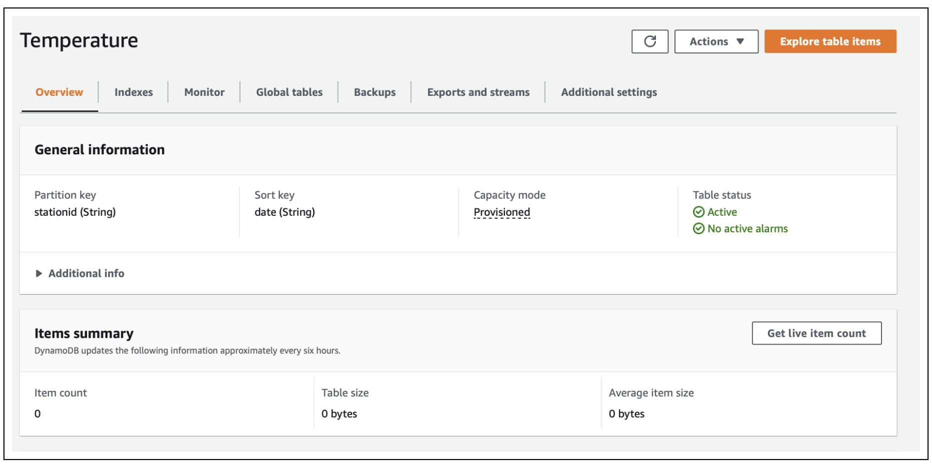


Figure 4: Dynamodb Temperature Table Cinfiguration

2.3 Create IAM Policies

To ensure the AWS security best practice, I will apply the least-privilege permissions for the EC2 service and the Lambda service. As EC2 needs to send records to the Kinesis, the "ec2_producer_policy" should include "kinesis:PutRecord" action and the resource should be the Kinesis Data Stream we created, which is the "maryland_weather_record".

Figure 5 is the EC2 producer policy in JSON format.

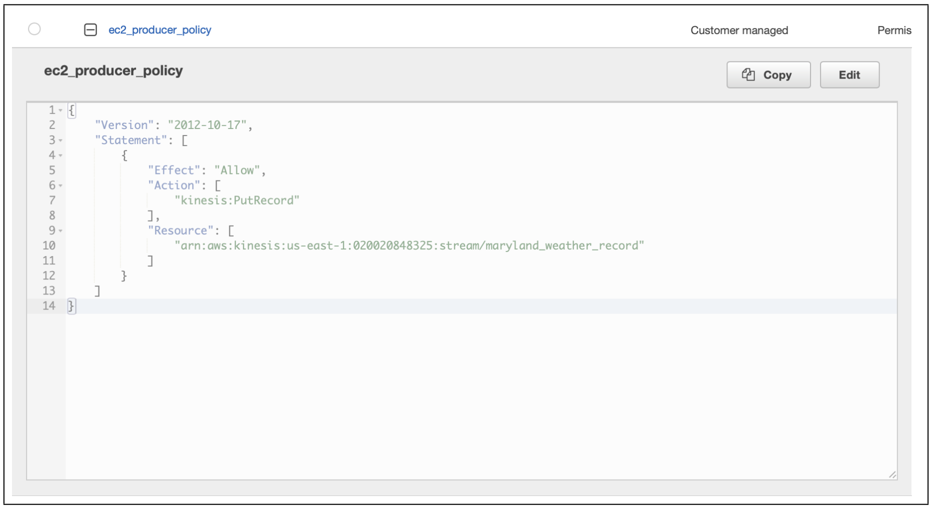


Figure 5: EC2 Producer Policy

The Lambda needs to access Kinesis Data Stream to get the records and put the records to DynamoDB. So, The "lambda_consumer_policy" should include "dynamodb:UpdateItem" action for the DynamoDB tables created previously and include "kinesis:DescribeStream", "kinesis:GetRecords", and "kinesis:GetShardIterator" action for the "maryland_weather_record" Kinesis Data Stream.
Figure 6 is the Lambda consumer policy in JSON format.

Figure 6: Lambda Consumer Policy

2.4 Create IAM Roles

Create two roles, one for the EC2 producer and the other for the Lambda consumer.
For the EC2 producer, the role should contain the policy that authorizes EC2 to put records in the Kinesis Data Stream.
The EC2 producer role is named "ec2_producer_role" with attached "ec2_producer_policy" policy.

Figure 7 shows the EC2 producer role.

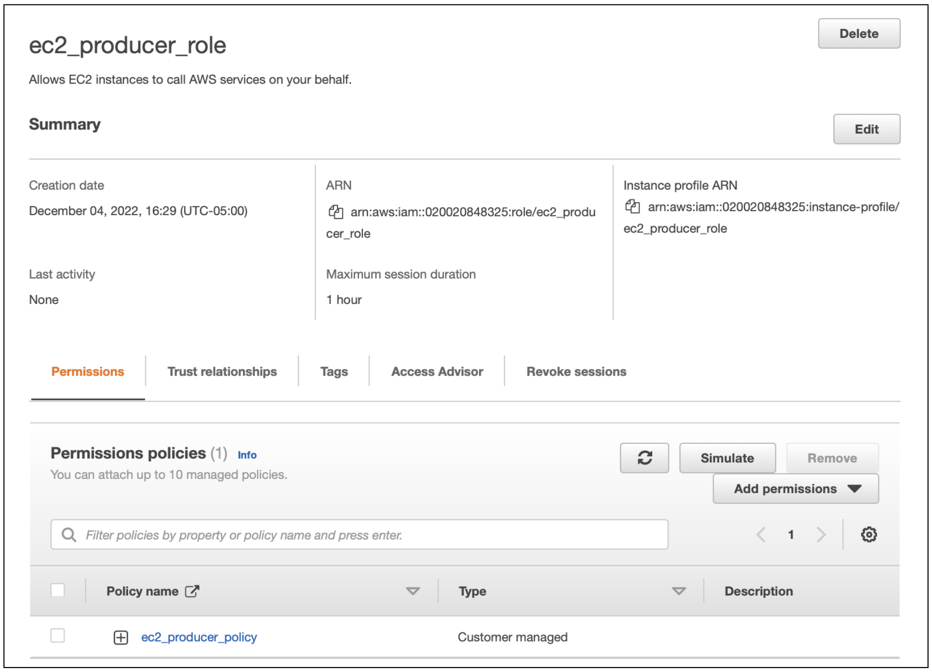


Figure 7: EC2 producer role

For the Lambda consumer, the role should contain the policy that authorizes Lambda to access and get records from Kinesis. Also, a policy for Lambda to and put records to Dynamodb.
Also, a "LambdaBasicExecutionRole" is needed for the lambda to upload logs to CloudWatch. The Lambda consumer role is named "lambda_consumer_role" with attached "lambda_consumer_policy" and "LambdaBasicExecutionRole" policy.

Figure 8 shows the Lambda Consumer role.

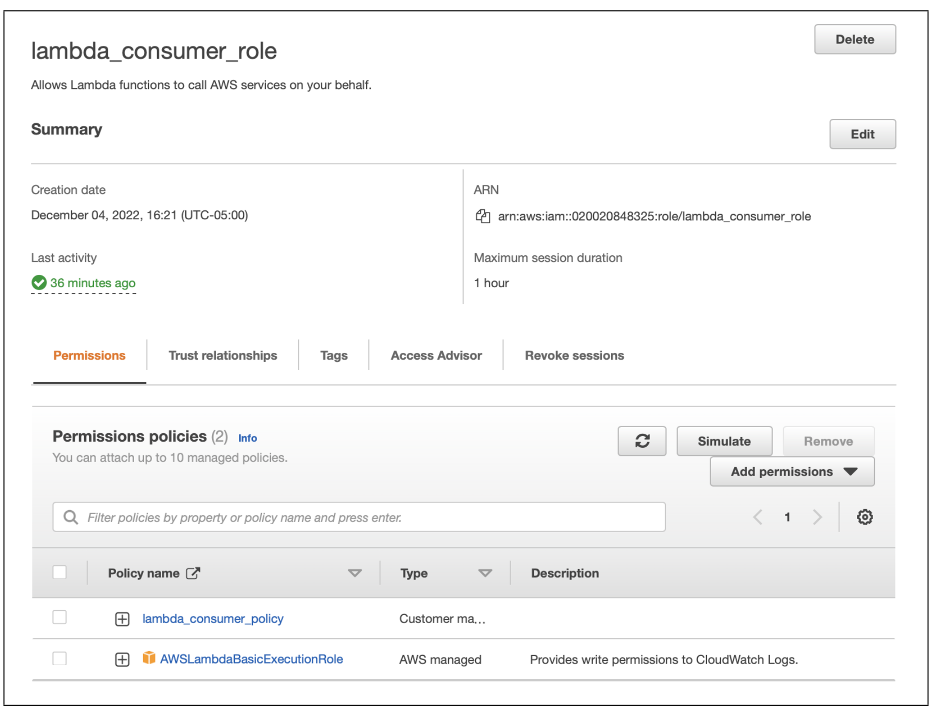


Figure 8: Lambda consumer role

2.5 Create Lambda

Create Lambda, which will be triggered by Kinesis, and put the records from Kinesis to Dynamodb. The name of our lambda is "lambda_consumer", and select "python 3.9" in the "Runtime" section. Assign "lambda_consumer_role" for the execution role.

Figure 9 shows the Lambda configuration settings.

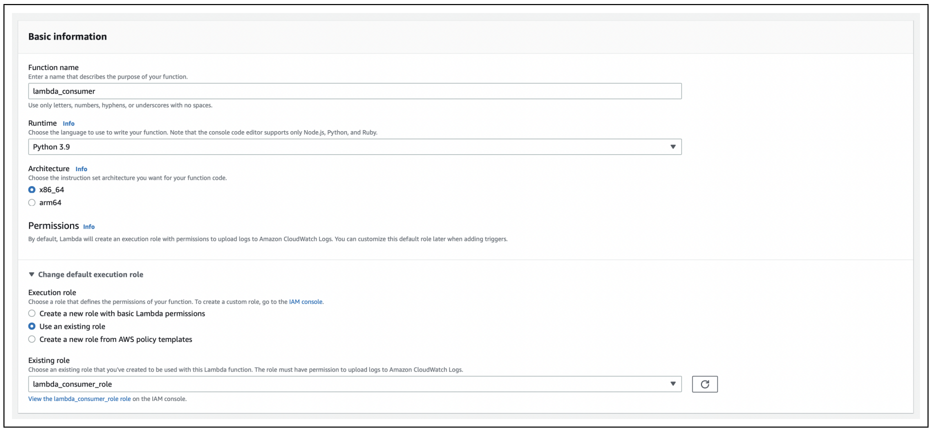


Figure 9: Lambda Configuration Setting

In the "Code Source" section, paste the "lambda_consumer_code" that could get the records in the Kinesis Data Stream and send the records to Dynamodb tables.

Figure 10 shows the Lambda consumer code.

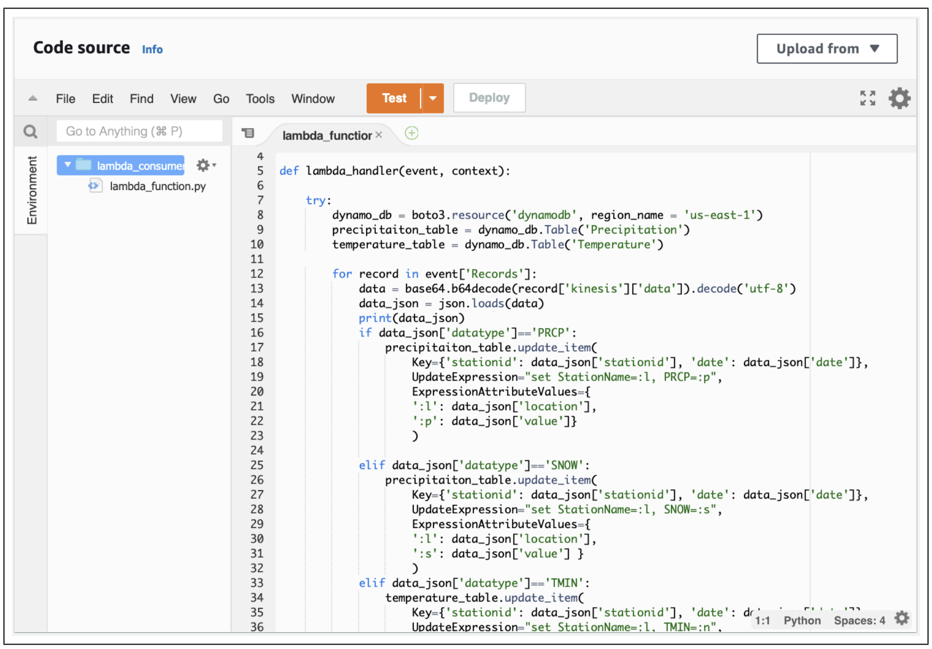


Figure 10: Lambda Consumer Code

Add trigger to the Lambda Consumer, select Kinesis and select "maryland_weather_record" in the "Kinesis stream" option. The batch size is set to 50.

Figure 11 shows the Lambda trigger settings.

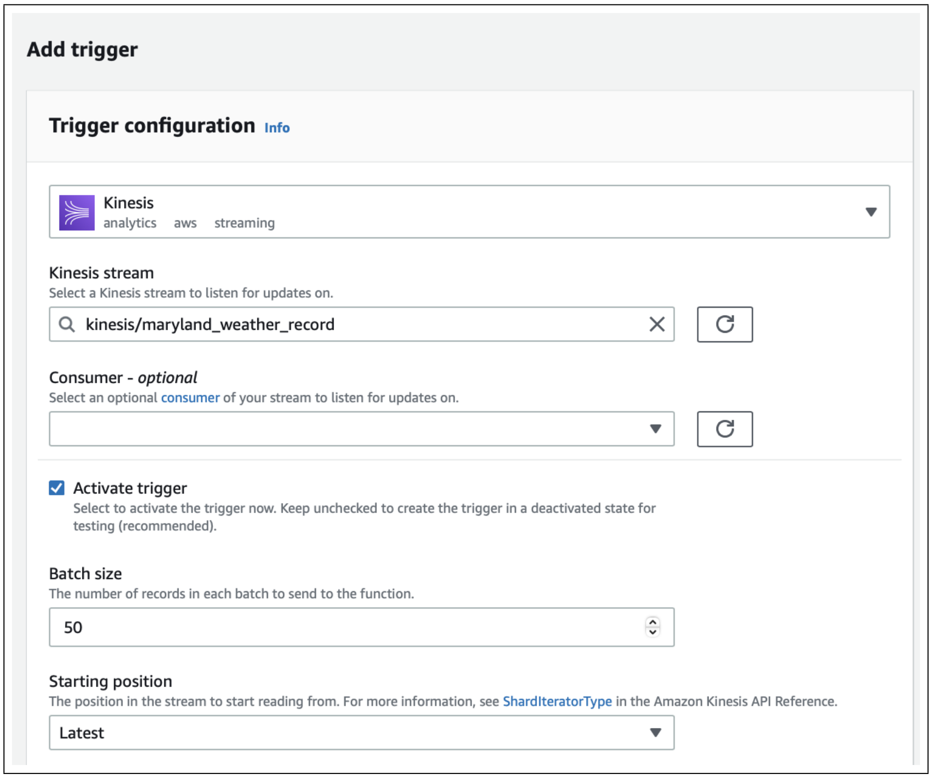


Figure 11: Lambda Trigger settings

Figure 12 shows the Lambda general configuration.

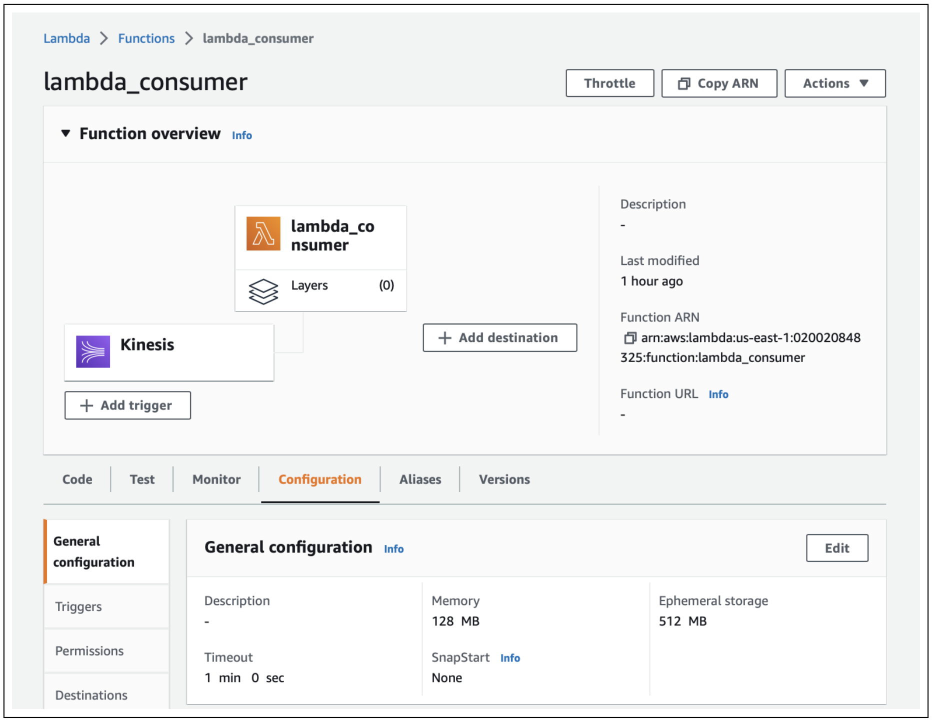


Figure 12: Lambda Configuration

Figure 13 shows the Lambda trigger configuration.

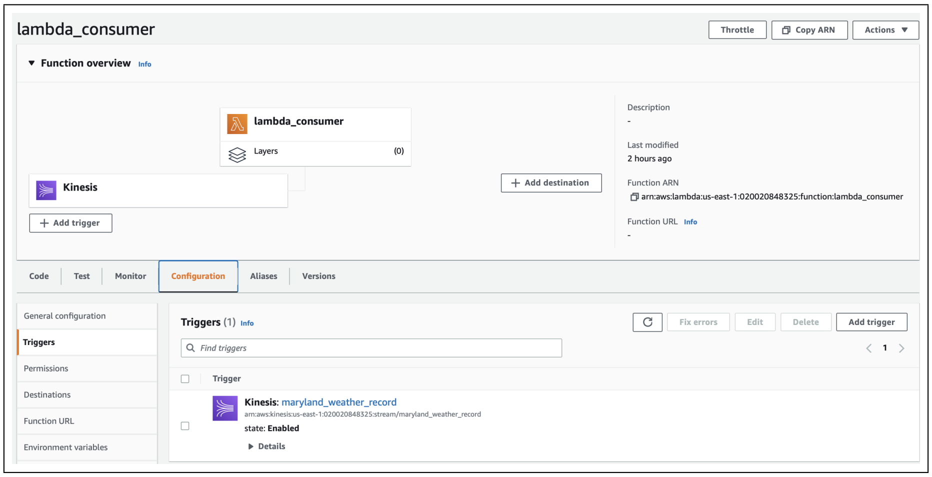


Figure 13: Lambda Trigger

2.6 Create EC2 Instance

Next, create EC2 instance as a producer to put records into the Kinesis Data Stream. The name of the EC2 is ec2_producer, with ”CentOS 7 (x86_64) - with Updates HVM” AMI in AWS marketplace. The instance type is set to ”t2.micro”, and the key pair is the ”data650.pem” file.
In the ”Network setting” section, click ”create security group” and allow SSH traffic from anywhere. In the ”Advance detail” section, select ”ec2_producer_role” for the ”IAM instance profile”.

Figure 14 shows the EC2 name and AMI selection.

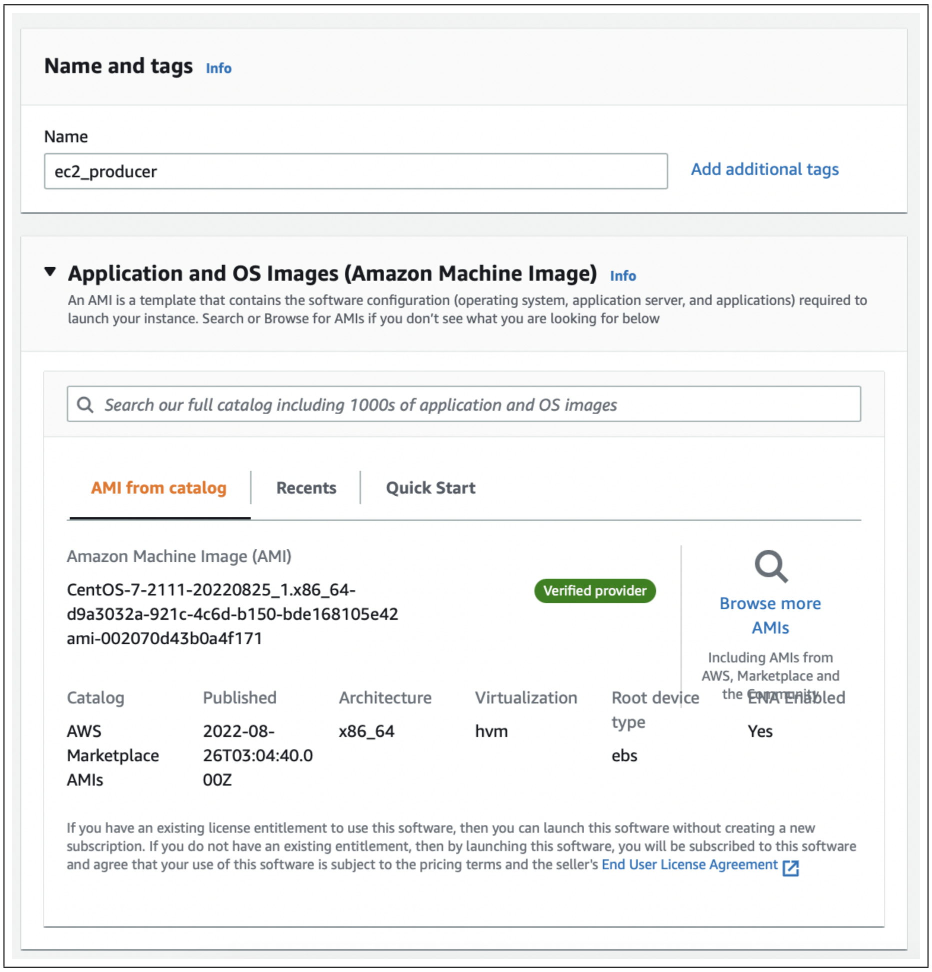


Figure 14: EC2 Name and AMI

Figure 15 shows the EC2 instance type and key pair selection.

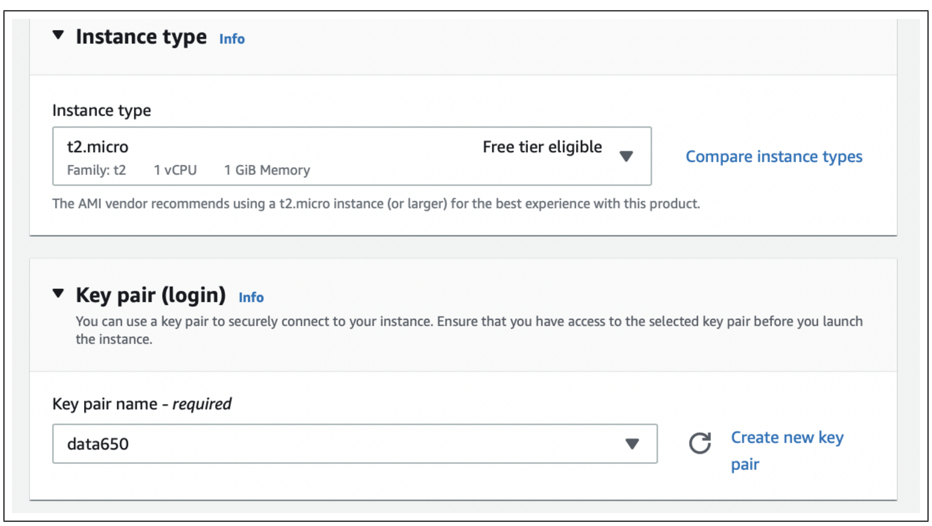


Figure 15: EC2 Instance type and Key pair

Figure 16 shows the EC2 security group settings.

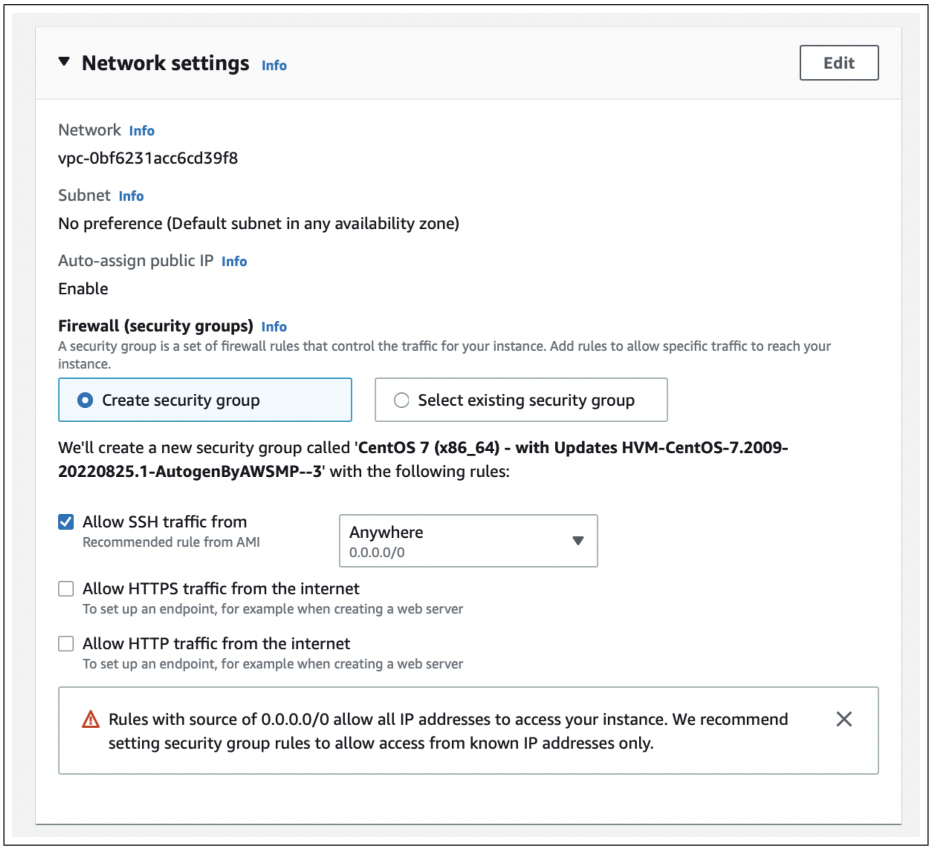


Figure 16: EC2 Security group

Figure 17 shows the EC2 role selection.

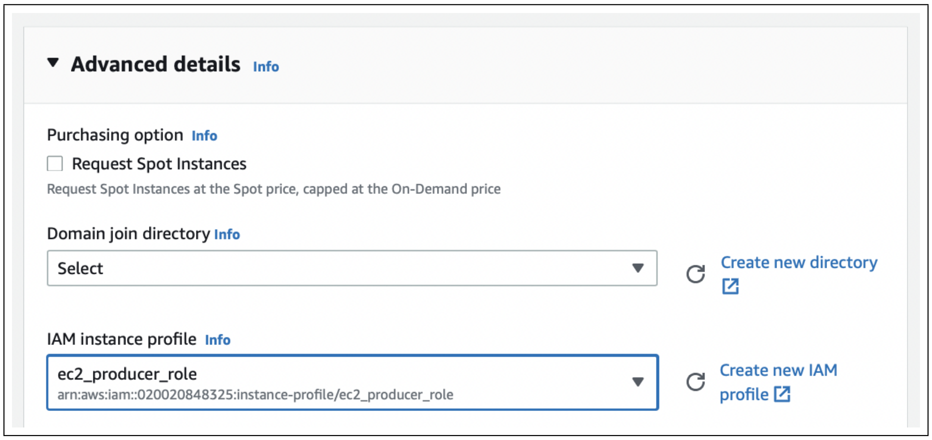


Figure 17: EC2 Role

Figure 18 shows the EC2 configuration.

Figure 18: EC2 Configuration

3 Test the Pipeline

To test the pipeline, first SSH in our EC2 instance with our key pair and the EC2 puclic ip address, and install nano and python3 with the code "sudo yum install nano" and "sudo yum install python3". Then, create a nano file, paste the "ec2_producer_code" in the file and saved the file as "ec2_producer_code.py. Install the boto3 library and the requests library for the code to run. Finally, run the "ec2_producer_code.py" file with python3. Every time the record is put to the kinesis, the terminal will print out the number of records. The total number of the records is 10670.

4 Monitoring the result

To check whether the records are put into the Kinesis, click the Kinesis monitoring dashboard.

Figure 19 shows that in the GetRecords-sum section, there are 6388 records in the first peak, and 4282 records in the second stage. There are 10670 records in total that flows into the Kinesis Data Stream.

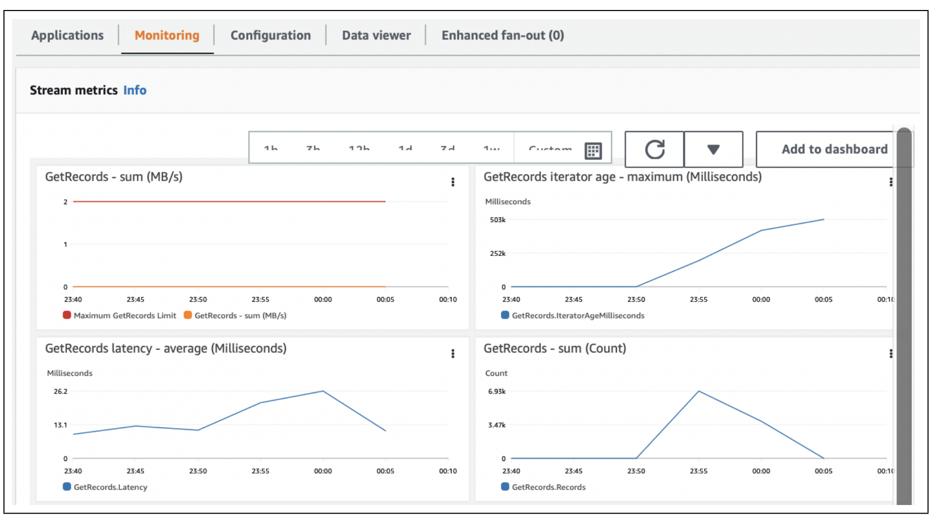


Figure 19: Kinesis Monitoring 1

Figure 20 shows the the GetRecord Success rate and PutRecord Success rate are both 1, which means all the desired records are sent to Kinesis Data Stream successfully.

.

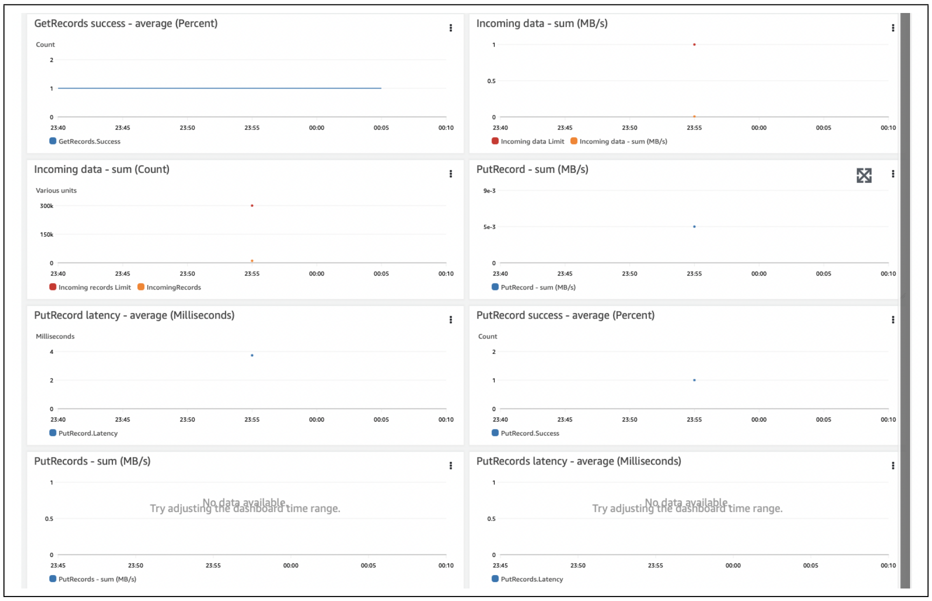
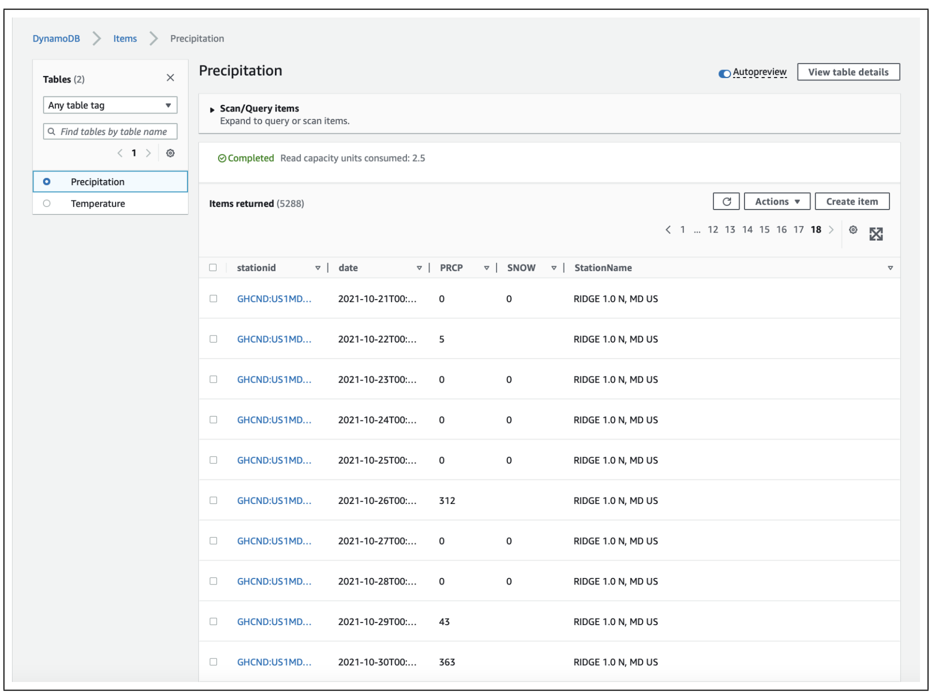


Figure 20: Kinesis Monitoring 2

To ensure the data are put into the DynamoDB table, click the "view item details" and see if the records are successfully put into the tables.

Figure 21 shows that there are 5,288 records in the Precipitation table. The Precipitation table includes the information of station id, station name, date, PRCP value, and SNOW value.



stationid	date	PRCP	SNOW	StationName
GHCND:US1MD...	2021-10-21T00:...	0	0	RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-22T00:...	5		RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-23T00:...	0	0	RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-24T00:...	0	0	RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-25T00:...	0	0	RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-26T00:...	312		RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-27T00:...	0	0	RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-28T00:...	0	0	RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-29T00:...	43		RIDGE 1.0 N, MD US
GHCND:US1MD...	2021-10-30T00:...	363		RIDGE 1.0 N, MD US

Figure 21: Dynamodb Precipitation Table

Figure 22 shows that the there are 934 records in the Temperature table. The Temperature table includes the information of station id, station name, date, TMAX value, and TMIN value.

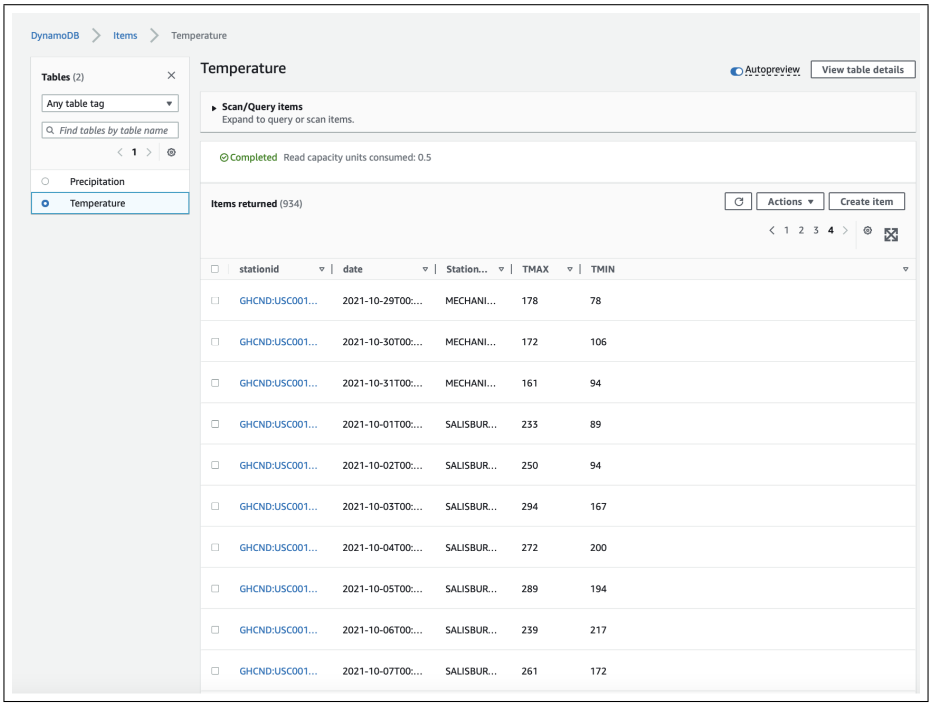


Figure 22: Dynamodb Temperature table

4.1 Appendix

- EC2 producer code:

The EC2 producer code sends requests through the NOAA Climate Data Rest API and parses the received JSON files to look for the targeted information. The targeted information is saved as a JSON file then sent to the Kinesis.

```python
import boto3
import json
import requests
from datetime import datetime
import json
import time

# connect to AWS Kinesis
my_stream_name = 'maryland_weather_record'
kinesis_client = boto3.client('kinesis',
                              region_name='us-east-1',
                              aws_access_key_id='',
                              aws_secret_access_key=''
                             )

#request data from weather rest API
token = ''
header = dict(token=token)
location_id = 24 # Maryland id
url = 'https://www.ncdc.noaa.gov/cdo-web/api/v2/stations?locationid=FIPS:' + \
        str(location_id) + '&limit=1000'
r = requests.get(url, headers=header)
d = json.loads(r.text)

#filter the stations that has data in our desire date range
station_id = []
station_name = []
for item in d['results']:
    min_date = datetime.strptime(item['mindate'], "%Y-%m-%d")
    max_date = datetime.strptime(item['maxdate'], "%Y-%m-%d")

    start_date = datetime.strptime('2020-10-01', "%Y-%m-%d")
    end_date = datetime.strptime('2020-10-31', "%Y-%m-%d")

    if (max_date >= end_date) and (min_date <= start_date):
        station_id.append(item['id'])
        station_name.append(item['name'])
```

Figure 23: EC2 producer code 1

```python
#retrieve data from the filtered stations list
count = 0
for i in range(len(station_id)):
    sid = station_id[i]
    sname = station_name[i]
    url = 'https://www.ncdc.noaa.gov/cdo-web/api/v2/data?datasetid=GHCND' + \
          '&limit=1000&stationid=' + sid + '&startdate=2021-10-01&enddate=2021-10-31'
    r = requests.get(url, headers={'token':token})
    d = json.loads(r.text)
    # the token only accept 5 requests per second
    time.sleep(0.3)
    try:
        for item in d['results']:
            if item['datatype']=='PRCP' or item['datatype']=='SNOW' or \
                item['datatype']=='TMAX' or item['datatype']=='TMIN':
                json_data = {
                                'date': item['date'],
                                'stationid': sid,
                                'location': sname,
                                'datatype': item['datatype'],
                                'value': item['value']
                            }
                time.sleep(0.01)
                put_response = kinesis_client.put_record(
                    StreamName=my_stream_name,
                    Data=json.dumps(json_data),
                    PartitionKey='stationid')

                count += 1
                print(count)
    except:
        # station id is empty
        continue
```

Figure 24: EC2 producer code 2

- Lambda consumer code:

  The Lambda consumer code separates the records by the "datatype" of the records. The records are separated into four categories, including "PRCP", "SNOW", "TMAX", and "TMIN". If the datatype is "SNOW" or "PRCP, the records will be sent to the "Precipitation" DynamoDB table and if the datatype is "TMAX" or "TMIN", then the records will be sent to the "Temperature" table.

  Using the update_item function will keep the records that have the same partition key and sort key, so that the records with the same stationid and date but different datatype won't be overwritten.

```python
import json
import boto3
import base64

def lambda_handler(event, context):

    try:
        dynamo_db = boto3.resource('dynamodb', region_name = 'us-east-1')
        precipitaiton_table = dynamo_db.Table('Precipitation')
        temperature_table = dynamo_db.Table('Temperature')

        for record in event['Records']:
            data = base64.b64decode(record['kinesis']['data']).decode('utf-8')
            data_json = json.loads(data)
            print(data_json)
            if data_json['datatype']=='PRCP':
                precipitaiton_table.update_item(
                    Key={'stationid': data_json['stationid'], 'date': data_json['date']},
                    UpdateExpression="set StationName=:l, PRCP=:p",
                    ExpressionAttributeValues={
                    ':l': data_json['location'],
                    ':p': data_json['value']}
                    )

            elif data_json['datatype']=='SNOW':
                precipitaiton_table.update_item(
                    Key={'stationid': data_json['stationid'], 'date': data_json['date']},
                    UpdateExpression="set StationName=:l, SNOW=:s",
                    ExpressionAttributeValues={
                    ':l': data_json['location'],
                    ':s': data_json['value'] }
                    )
```

Figure 25: Lambda consumer code 1

```python
                    )
            elif data_json['datatype']=='TMIN':
                temperature_table.update_item(
                    Key={'stationid': data_json['stationid'], 'date': data_json['date']},
                    UpdateExpression="set StationName=:l, TMIN=:n",
                    ExpressionAttributeValues={
                    ':l': data_json['location'],
                    ':n': data_json['value'] }
                    )

            elif data_json['datatype']=='TMAX':
                temperature_table.update_item(
                    Key={'stationid': data_json['stationid'], 'date': data_json['date']},
                    UpdateExpression="set StationName=:l, TMAX=:x",
                    ExpressionAttributeValues={
                    ':l': data_json['location'],
                    ':x': data_json['value'] }
                    )



    except Exception as e:
            print(str(e))
```

Figure 26: Lambda consumer code 2