I. Introduction

The project aims to build a pipeline that acquires weather data, processes it in real-time using AWS Kinesis, and store it in a DynamoDB table. The weather data entails information about the daily measurement for precipitation (in mm), snowfall (in mm), and minimum and maximum temperature (in Fahrenheit) for Maryland, United States between October 1, 2021, and October 31, 2021.

The project workflow broadly involves acquiring data from National Oceanic and Atmospheric Administration's (NOAA's) REST API [1], inserting data into the AWS Kinesis data stream in real-time, extracting data from the stream, and importing it into DynamoDB tables. The project will be tested based on performing queries on the imported data, contained in the DynamoDB tables.

II. Background

The section describes the AWS services that were utilized during project implementation.

A. Amazon Kinesis

Amazon Kinesis is a managed service that enables real-time processing of streaming data at scale. With Kinesis, we can ingest real-time data such as video, audio, application logs, website clickstreams, and IoT telemetry data for machine learning, analytics, and other applications.

To create a real-time data stream using Kinesis, we will need to first create a Kinesis stream. This is a durable, scalable, and highly available data stream that stores data records for a configurable amount of time. We can then use one of the Kinesis producers (such as the Kinesis Producer Library or the Kinesis Data Streams API) to put data records into the stream. Once the data records are in the stream, we can use one of the Kinesis consumers (such as the Kinesis Consumer Library or the Kinesis Data Streams API) to read and process the data in real-time. For example, use Kinesis to feed real-time data into a machine learning model for real-time predictions, or into a real-time analytics platform for real-time analysis.

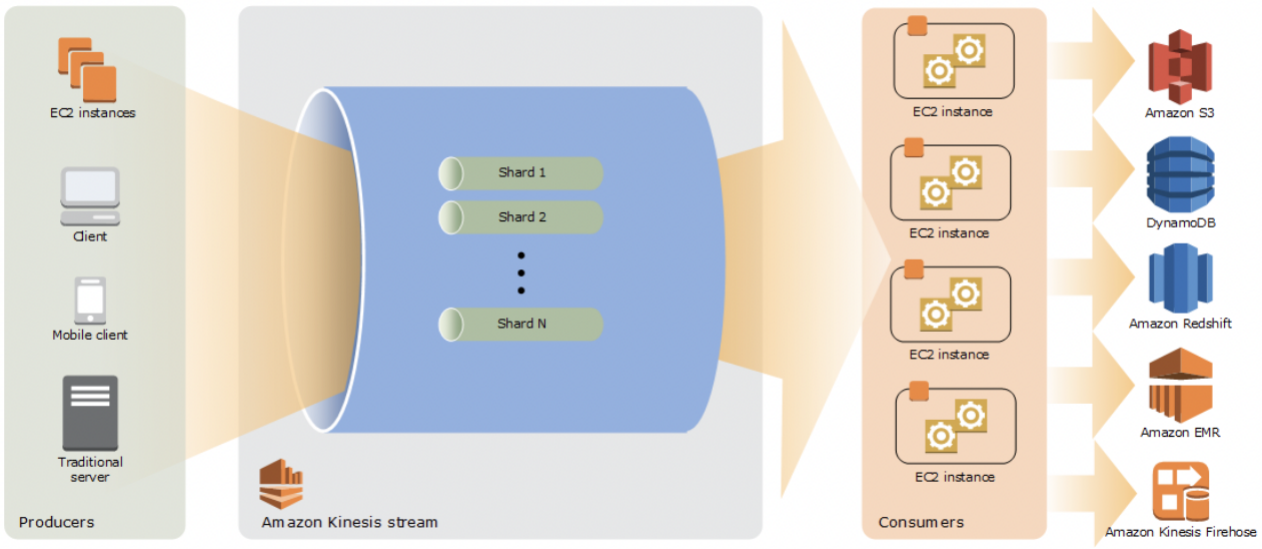

Fig. 1: Kinesis Data-Streams High Level Architecture

Overall, Amazon Kinesis provides a powerful and scalable solution for real-time data streaming and processing. The definition of real-time entails obtaining insights from the data in seconds instead of waiting for hours.

Amazon Kinesis is structured around four services, out of which we will be picking **Amazon Kinesis Data Stream** for the project. Figure 1 shows a high-level architecture of the kinesis data streams. The following points describe the kinesis data stream and its components.

- A Kinesis data stream is a set of shards.

Base URL

```
https://www.ncei.noaa.gov/cdo-web/api/v2/{endpoint}
```

Fig. 2: Base URL to extract data from NOAA's REST API

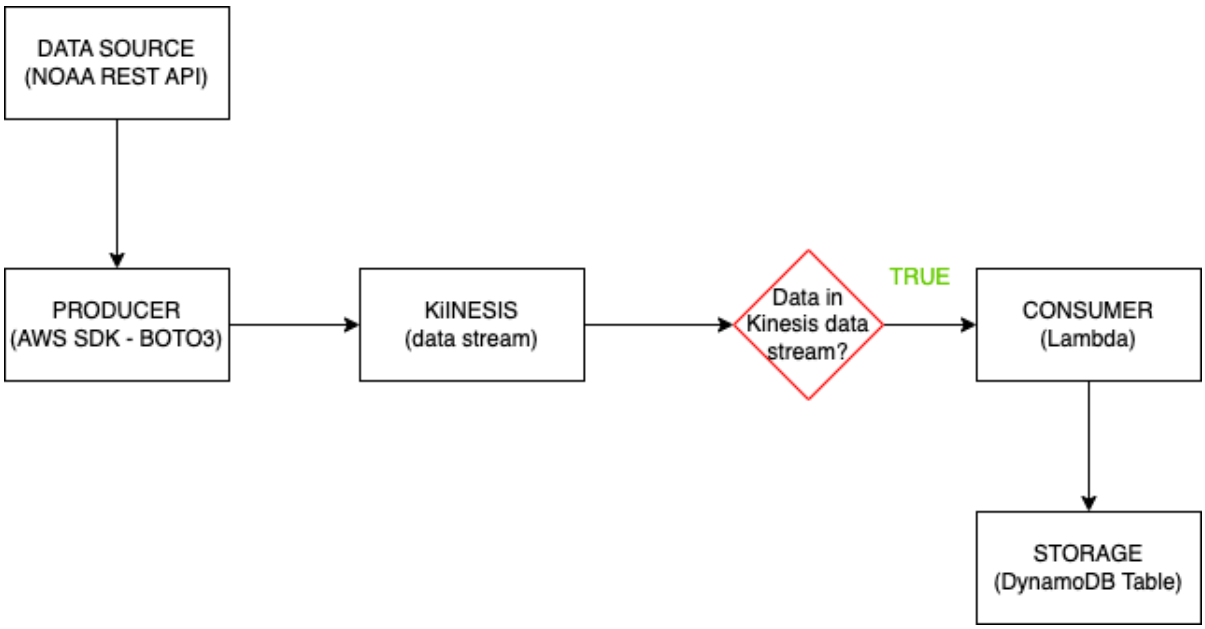

Fig. 3: Project Workflow

- A shard is a sequence of data records, uniquely identified by a partition key. Hence, all data records in a shard will have the same partition key. Each shard can support up to 1000 records per second for writes and 5 records per second for reads.
- Each data record in a shard is identified by a sequence number, unique within a shard.
- A producer is required to put data into a data stream. For the project, we will be using AWS SDK for Python—**Boto3**, as a producer.
- A consumer is required to extract data from a data stream. For the project, we will be using AWS **Lambda** service as a consumer, which gets triggered when there are data records in the data stream.

B. Amazon Cloud9

Amazon Cloud9 is an integrated development environment (IDE) offered by Amazon Web Services (AWS). It is a cloud-based IDE that allows developers to write, run, and debug their code in a web browser.

Cloud9 comes with a built-in terminal, code editor, and debugger, making it easy to develop and run your code. It also integrates with other AWS services, such as Amazon S3 and Amazon EC2, allowing you to easily access and manage your resources from within the IDE.

C. AWS Identity and Access Management (IAM)

AWS Identity and Access Management (IAM) is a web service that helps securely control access to AWS resources. IAM enables to manage users and their permissions in AWS.

An IAM role is a type of IAM identity that we can create and assign to users or AWS services. Roles are similar to users, but they do not have their own set of credentials (such as a password or access keys). Instead, you grant permissions to a role and then delegate those permissions to users or services that assume the role. For example,

we can create an IAM role that allows access to an Amazon S3 bucket, and then grant that role to an AWS Lambda function. When the Lambda function is executed, it will assume the IAM role and be able to access the S3 bucket.

An IAM policy is a document that specifies which actions a user or role can perform, on which resources, and under what conditions. WE attach IAM policies to users, roles, or groups to define their permissions. For example, we can create an IAM policy that allows a user to read and write to an Amazon S3 bucket, and attach that policy to the user. The user will then be able to access the S3 bucket according to the permissions specified in the policy.

D. Amazon DynamoDB

Amazon DynamoDB is a fully managed, NoSQL database service offered by Amazon Web Services (AWS). It provides fast and predictable performance with seamless scalability.

DynamoDB is designed to be highly flexible and scalable, with the ability to handle massive amounts of data and high levels of traffic. It uses a key-value data model, which allows developers to store, retrieve, and update data using simple APIs.

E. Amazon Lambda

Amazon Lambda is a serverless compute service offered by Amazon Web Services (AWS). It allows us to run code in response to events, such as changes to data in an Amazon S3 bucket or an Amazon DynamoDB table.

With Lambda, we don't need to worry about managing servers or scaling your application. Lambda automatically scales the application in response to incoming traffic, and we only pay for the compute time that you consume.

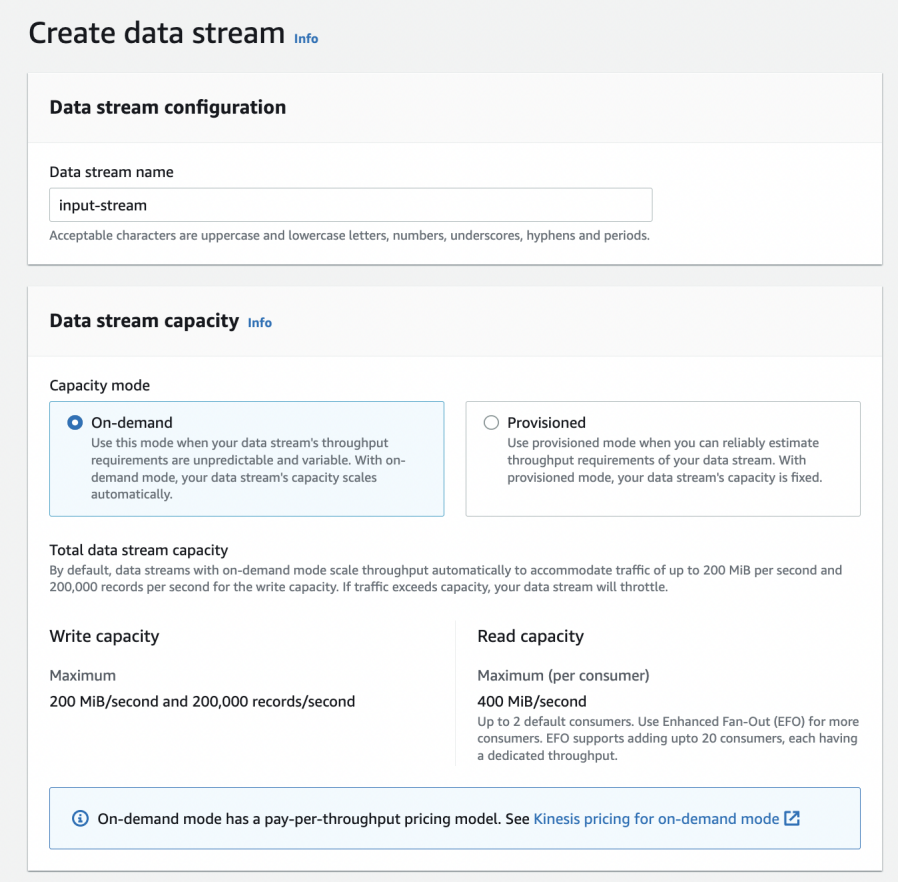

Fig. 4: Step 2: Create AWS Kinesis data stream

F. Amazon CloudWatch

Amazon CloudWatch is a monitoring service for Amazon Web Services (AWS) that provides data and insights to help manage and optimize applications and infrastructure. CloudWatch collects and tracks metrics, logs, and events,

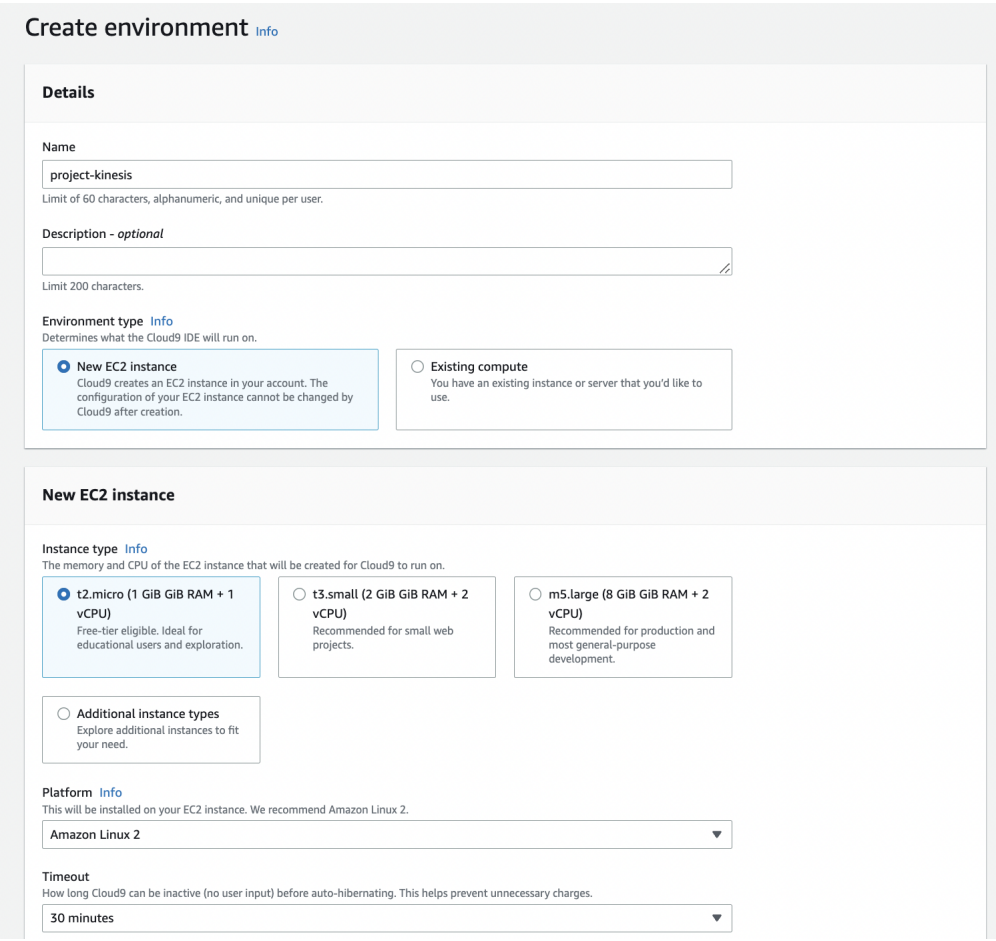

Fig. 5: Step 4: Create IDE using AWS Cloud9

allowing us to monitor the resources and applications in real-time. Some of the key metrics that CloudWatch can collect and monitor include:

- CPU utilization and network traffic
- Memory and disk usage
- Request count and latency
- Errors and status codes

III. Data Description

For the project, we will be using the weather data available via the National Oceanic and Atmospheric Administration's (NOAA's) website via REST API. The service gives access to accumulated climate data across the globe with an emphasis on the continental United States. To access the web services, a token needs to be obtained from the token request page [2].

To obtain the daily weather parameters, we will be using the Global Historical Climatology Network- Daily (GHCND)[3] dataset. Each record represents available information for a particular station on a particular day. The following are the core attributes of the dataset:

- *Date* is the year of the record(4 digits) followed by the month (2 digits) and day (2 digits).
- *Station* is the station identification code (17 characters).
- *Station_name* is the name of the station.
- *PRCP* is the measurement of the precipitation value in mm.
- *SNOW* is the measurement of the snowfall in mm.
- *TMIN* is the minimum temperature in Fahrenheit.
- *TMAX* is the maximum temperature in Fahrenheit.

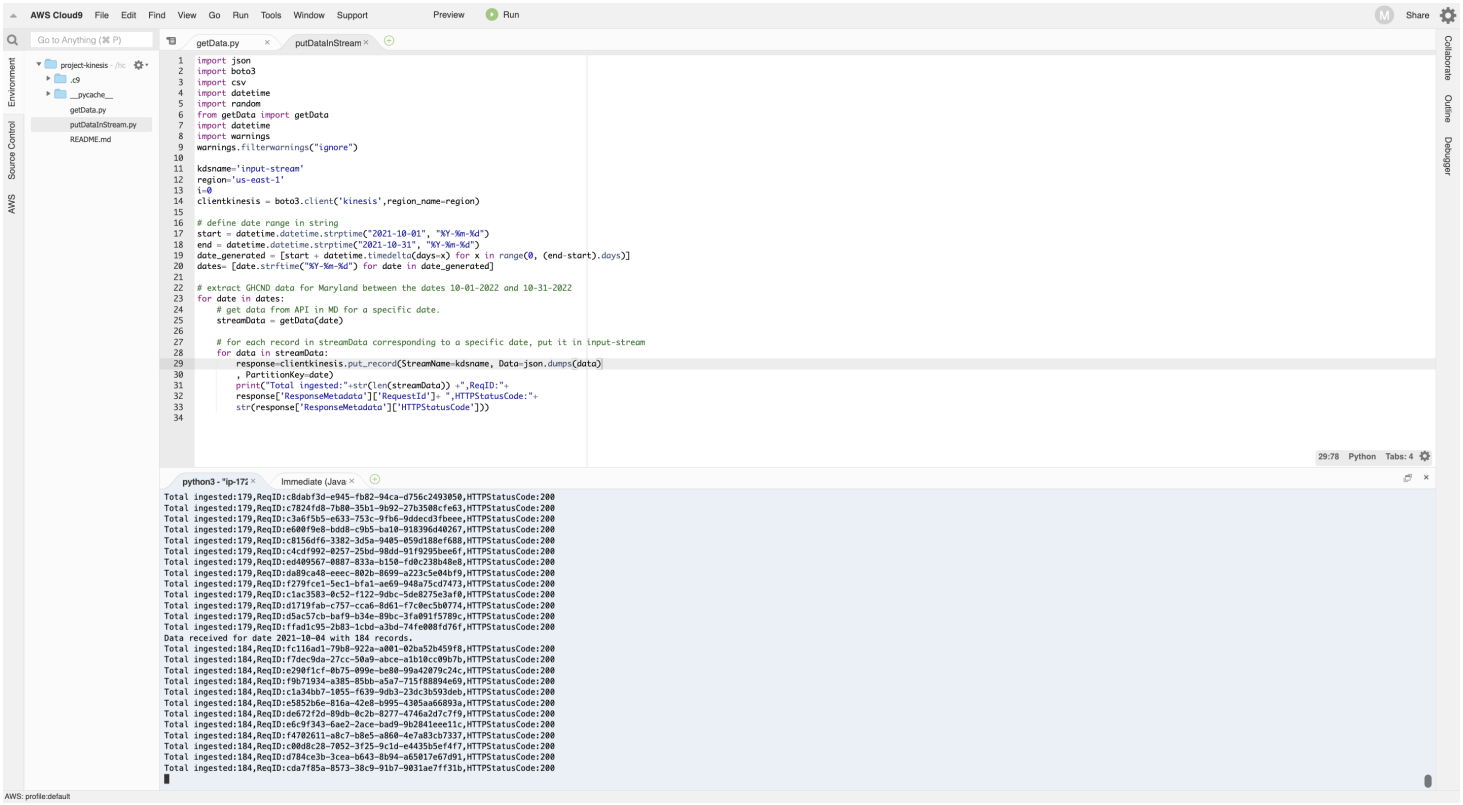

Fig. 6: Step 4: SDK producer on Cloud9

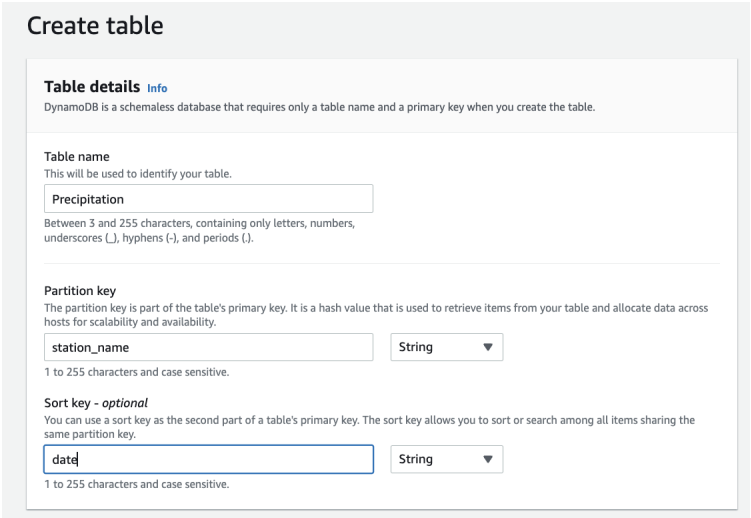

Fig. 7: Step 5: Create DynamoDB table with name Precipitation

The REST API has various endpoints to extract different types of information such as the name of the dataset, data categories, station names, and the data itself. We will use a combination of different endpoints to get data in batches, according to the date, and put together the core attributes as a single record in the kinesis data stream.

IV. Project Workflow

The section describes the workflow of the project in a broader context, as described in Figure 3. We will begin by extracting data using NOAA's REST API and reformat the structure of the data. This is followed by creating a producer to put the formatted data into the AWS Kinesis data stream. A lambda function is created as a consumer

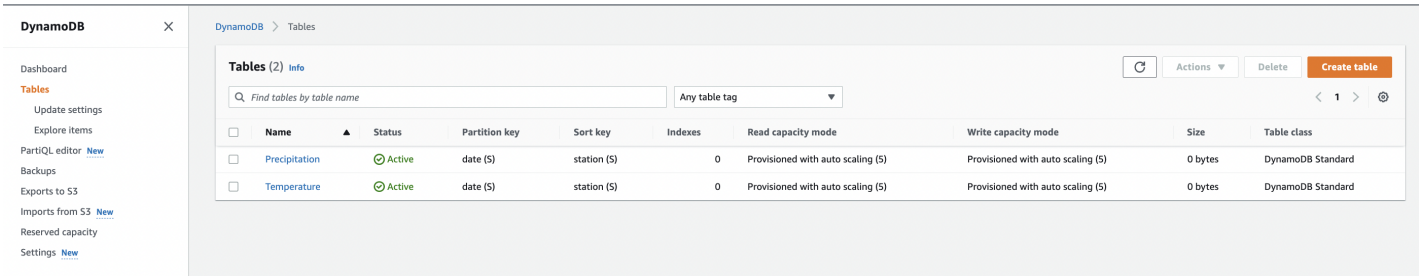

Fig. 8: Step 5: Screenshot of DynamoDB tables

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:UpdateTable",
                "dynamodb:DescribeTable",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:354077325797:table/Precipitation",
                "arn:aws:dynamodb:us-east-1:354077325797:table/Temperature"
            ],
            "Effect": "Allow"
        }
    ]
}
```

Fig. 9: Step 6: Customer managed policy in JSON format

that gets triggered on an event— presence of data in the kinesis data stream. The consumer extracts the data from the stream and stores it in a DynamoDB table.

V. Procedure

Following the guideline presented in Figure 3, the section describes the procedure in 7 steps.

A. Step 1: Extract data using NOAA's REST API

To extract weather parameters using the REST API, we use the base URL mentioned in Figure 2. As mentioned in Section III, we need two sets of information:

- Names of all weather stations in Maryland.
- Day-wise weather parameters for Maryland between October 1, 2021, and October 31, 2021.

To extract the former set of information, we use the endpoint— $stations?locationid = FIPS:24\&limit = 1000$. The endpoint extracts information about the stations using Federal Information Processing Standard (FIPS) code, which uniquely identifies the states and counties in the United States. The code for Maryland is 24. This returns 938 stations in Maryland.

To extract the data, we create *getData.py* file, containing a *getData* function that takes a date as an input. For the range of dates between October 1, 2021, and October 31, 2021, the function performs the following tasks:

- Extracts the station names in Maryland and creates a mapping between the station names and station (identification code) using the endpoint mentioned in Figure 2.

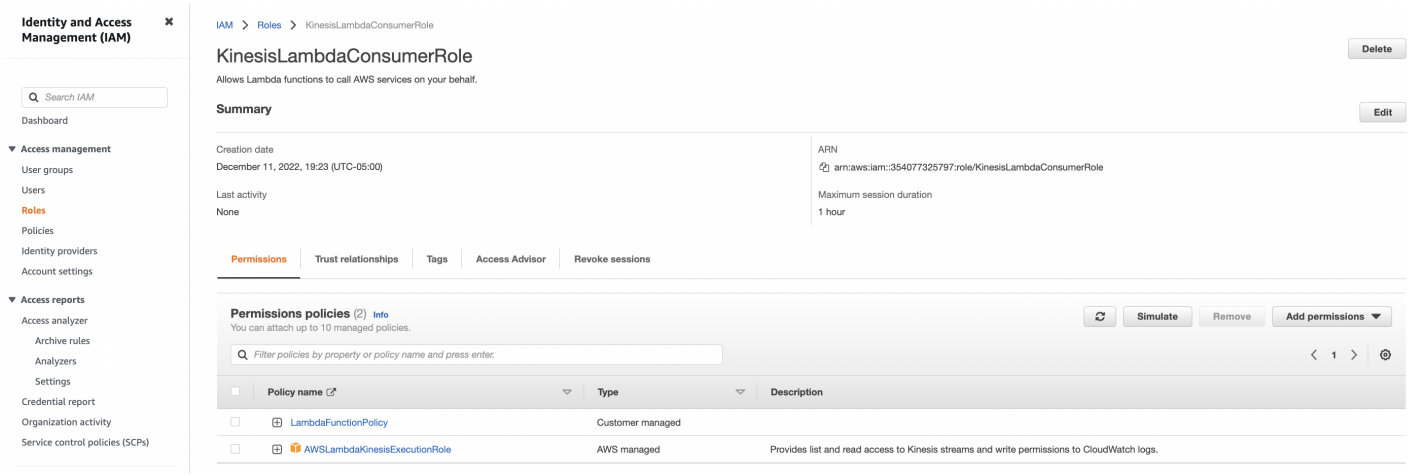

Fig. 10: Step 6: Screenshot of the IAM role with the appropriate policies

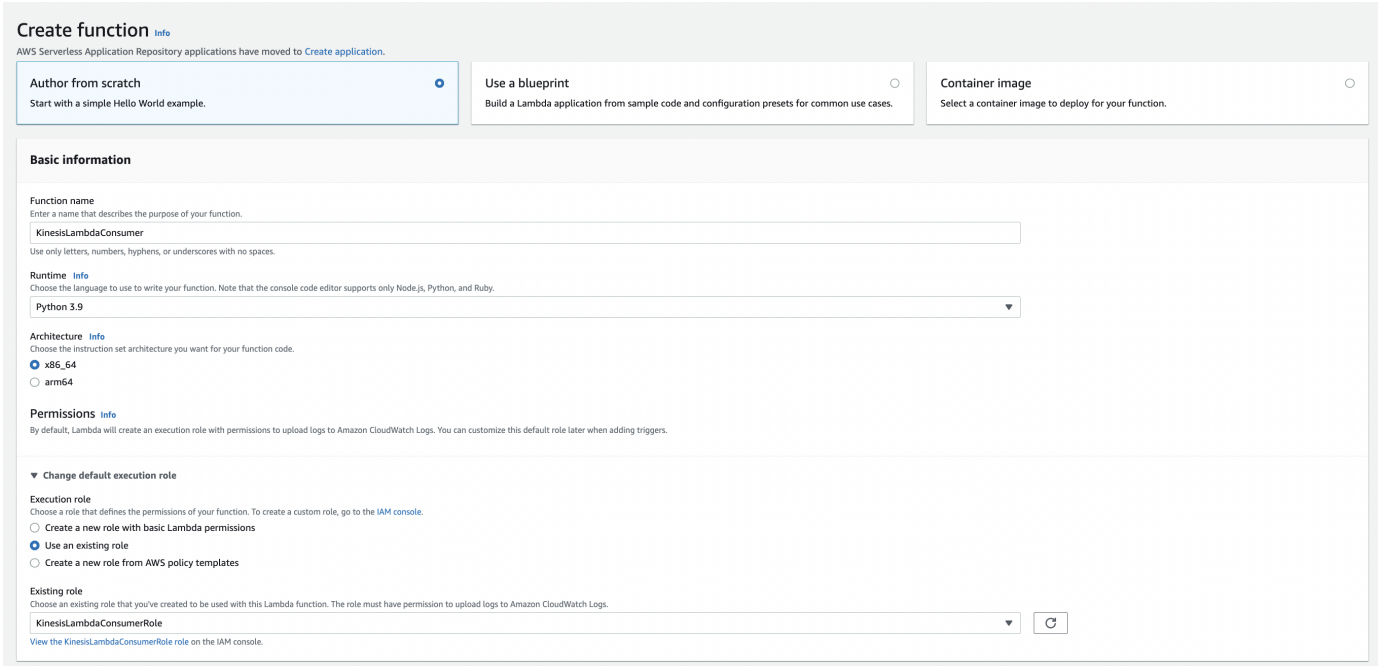

Fig. 11: Step 7: Creating Lambda function

- For a given date, obtain data using the endpoint— $data?datasetid = GHCND\&locationid = FIPS:24\&startdate = " + date + "\&enddate = " + date + "\& = date\&limit = " + str(limit)$. Here, date and limit are variables containing the date and the limit for records to obtain via the API, respectively.
- Process the extracted data from the REST API to create an array of python dictionaries, *streamData*, where each dictionary represents a record with the following keys— date, station, station_name, PRCP, SNOW, TMIN, and TMAX.

For more information on the process of extracting data from NOAA's REST API and to create *streamData*, refer Appendix A.

B. Step 2: Create Kinesis data stream

In this step, we will create a data stream by using the AWS Kinesis service. Simply go to **Kinesis** service on AWS and click on 'Create data stream'. The stream is named as **input-stream** and **on-demand** capacity mode is used, as shown in Figure 4. The mode allows automatically scaling the data stream capacity. By default, the retention period for the data in the stream is one day.

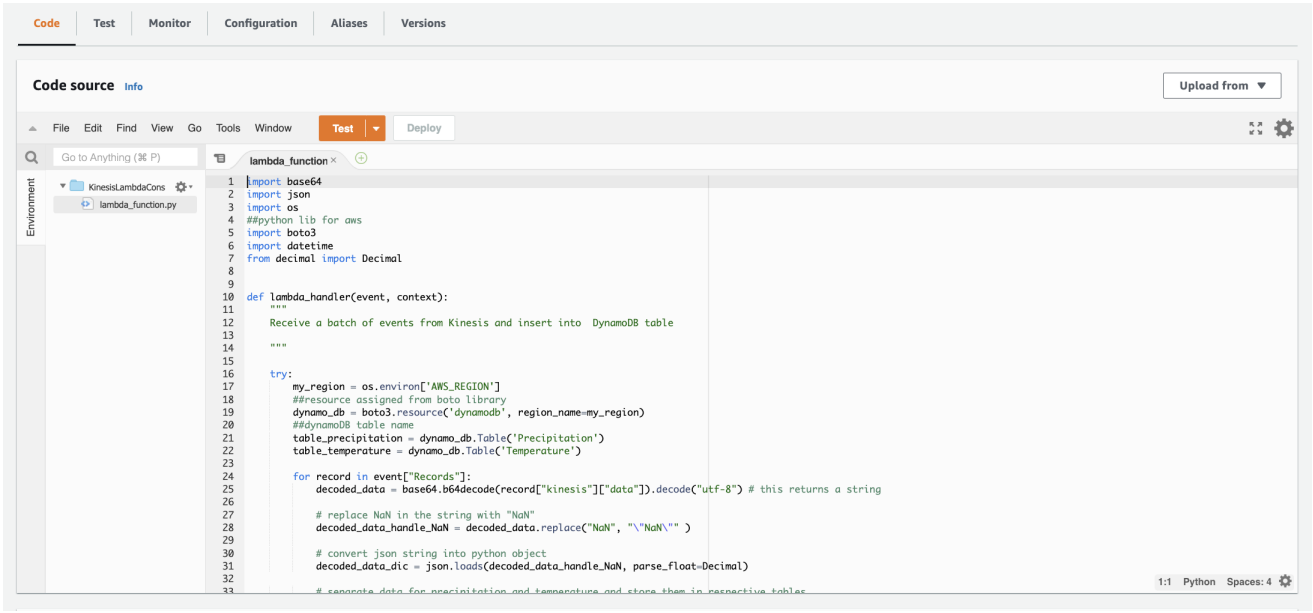

Fig. 12: Step 7: Uploading Python code for the lambda consumer

C. Step 3: Create Producer

Now that we have our data and data stream ready, it's time to put the data into the stream. For this purpose, we will use AWS SDK for Python— *Boto3*[4]. The SDK acts as a producer that creates a low-level service to connect to AWS Kinesis and put data onto it. For the range of dates between October 1, 2021, and October 31, 2021, we extract data using steps Section V-A, serialize each Python object in *streamData* variable, and put it on the stream using SDK. The code is available in Appendix B under the script name *putDataInStream.py*.

D. Step 4: Ingest data into the stream

We now create an Integrated Development Environment (IDE) on AWS using **Cloud9** service to run *putDataInStream.py* script and use the SDK producer to put data from the NOAA's REST API onto the Kinesis data stream called *input-stream* (created in Section V-B). For this purpose, we simply go to **Cloud9** service on AWS and click on 'Create environment'. The environment is named as **project-kinesis** with default settings as t2.micros as the EC2 instance and Amazon Linux 2 as the operating system. There is no change made to other parameters. Refer to Figure 5 for guidance.

Once the AWS Cloud9 environment is created, open it and upload the two scripts— *getData.py* and *putDataInStream.py*. Additionally, we manually install the following packages— boto3, requests, and pandas, using the command *pip3 install package-name*. Finally, we put the data onto the data stream by running the script *putDataInStream.py*, using the command *python3 putDataInStream.py*, as shown in Figure 6.

E. Step 5: Create DynamoDB Tables

Before extracting the data from the data stream, we create two DynamoDB tables— **Precipitation** and **Temperature**. For this purpose, we go to AWS **DynamoDB** service and click on 'Create table'. We then fill in the appropriate table names, and select *station_name* and *date* as the partition and sort keys, respectively, as seen in Figure 7. All other parameters have the default settings.

Once the tables have been created, they can be accessed using the **Tables** section of the DynamoDB service, as shown in Figure 8.

F. Step 6: Create IAM Role and Policy

In this step, we will create an Identity and Access Management (IAM) role for the consumer to perform the following functions:

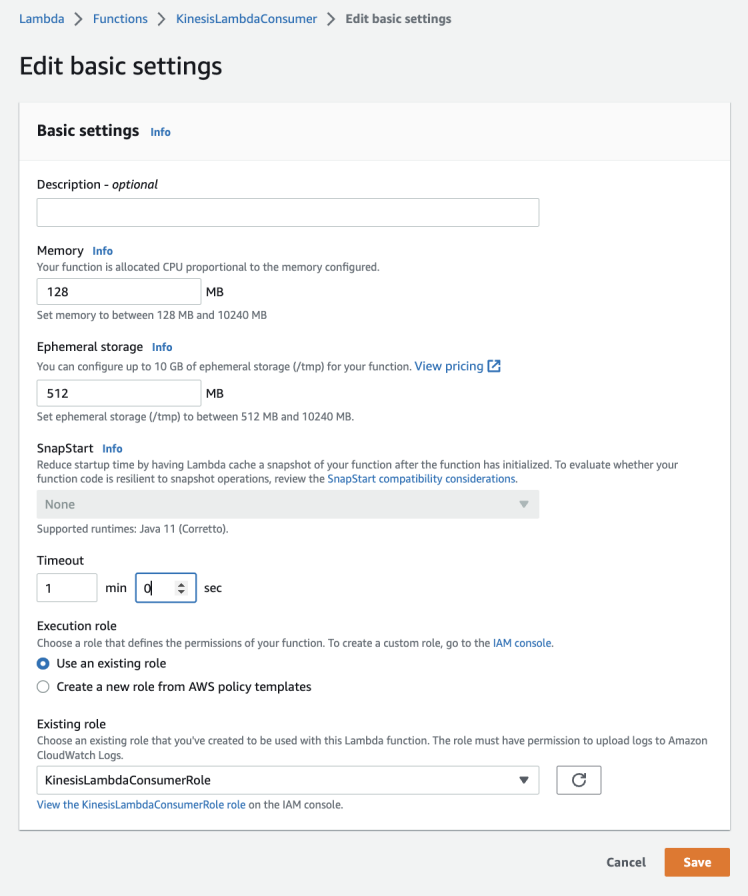

Fig. 13: Step 7: Changing timeout setting

- Provide read access from the Kinesis data stream and write to CloudWatch logs.
- Provide edit, read and write access for the DynamoDB tables.

For this purpose, we associate 1 customer managed IAM policy— *LambdaFunctionPolicy* and 1 AWS managed IAM policies to the role. To create a customer managed policy, go to the 'Policies' tab in the AWS **IAM** service, click on 'Create Policy' and provide the JSON file with name— **LambdaFunctionPolicy**, as shown in Figure 9. All other settings for the policy remains unchanged.

In the policy, we use the Amazon Resource Name (ARN) of the DynamoDB tables for the *Resource* key. Next, we create IAM role— **KinesisLambdaConsumerRole** by going to the 'Role' tabs and clicking on the 'Create role'. We use *AWS Service* and *Lambda* as the trusted entity type and use case, respectively. In addition to the *LambdaFunctionPolicy*, we further add the *AWSLambdaKinesisExecutionRole* that provides list and read access to Kinesis streams and write permissions to CloudWatch logs. The rest of the parameters have the default settings. Figure 10 displays a screenshot of the IAM role.

G. Step 7: Create Consumer

In this step, we use the AWS **Lambda** service to create a consumer called **KinesisLambdaConsumer** that gets triggered when there is data on the *input-stream*, created in Section V-B. For this purpose, we click on the 'Create function' tab provided under the AWS *Lambda* service with the following modifications (refer Figure 11):

- Python 3.9 is chosen as the programming language to write the lambda function.
- *KinesisLambdaConsumerRole* is associated with the function.
- All other parameters remain unchanged.

Next, we write a Python script— **lambda_handler.py** with a function called **lambda_handler** to give instructions to the lambda function on how to put data from the stream onto the DynamoDB tables, as mentioned in Appendix C.

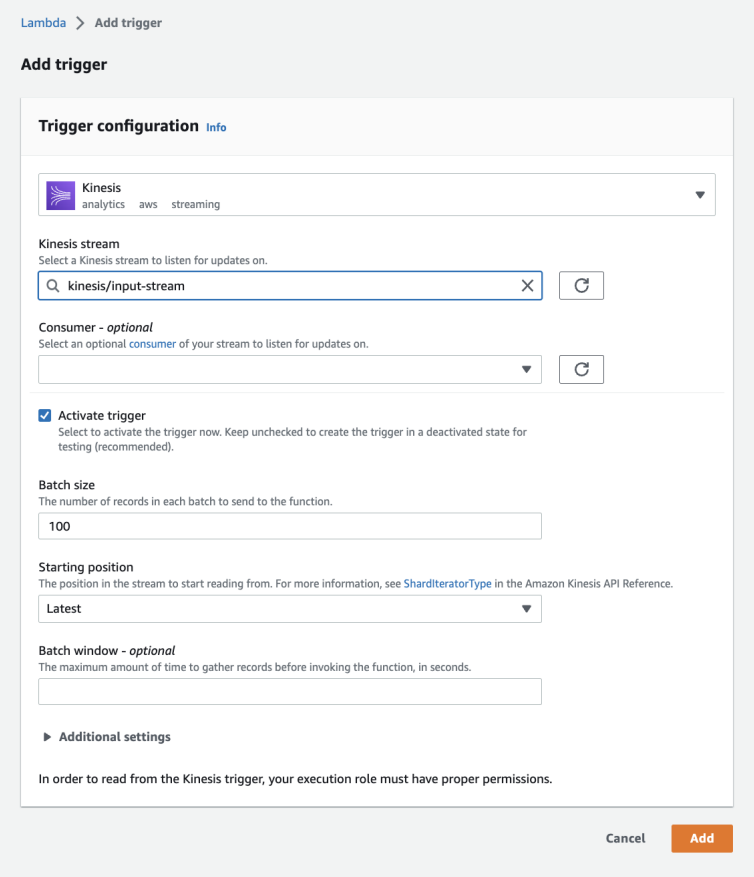

Fig. 14: Step 7: Adding trigger to the lambda consumer

```
Total ingested:179,ReqID:c42a0a4a-858b-c923-983b-627d61353ecd,HTTPStatusCode:200
Total ingested:179,ReqID:fb5ecb7e-370e-e831-a74f-a349d3b01fdf,HTTPStatusCode:200
Total ingested:179,ReqID:c840e18c-de0e-d312-9451-89bb3ab024fc,HTTPStatusCode:200
Total ingested:179,ReqID:e9c94daa-886a-7404-b5d8-259d6cd483ea,HTTPStatusCode:200
Total ingested:179,ReqID:d1e89a3b-af4a-3e27-8df9-f20c4bf4c9c9,HTTPStatusCode:200
Total ingested:179,ReqID:d731045d-33db-c179-8b20-6c6ad7653697,HTTPStatusCode:200
Total ingested:179,ReqID:fd053ba0-2050-1ed6-a114-5397c4eee938,HTTPStatusCode:200
Total ingested:179,ReqID:c19ef8b3-a125-a62a-9d8f-9084459b51c4,HTTPStatusCode:200
Total ingested:179,ReqID:d7f79478-fd3d-f955-8be6-fc4f19830ebb,HTTPStatusCode:200
Total ingested:179,ReqID:fadea614-1482-41c3-a6cf-ce23f03cb62d,HTTPStatusCode:200
Total ingested:179,ReqID:c21bdd8a-4aad-cada-9e0a-b5bdae133d34,HTTPStatusCode:200
Total ingested:179,ReqID:d04f8326-7b0f-67b0-8c5e-eb119fb1905e,HTTPStatusCode:200
Total ingested:179,ReqID:d6790a35-635a-aafc-8a68-620287e45d12,HTTPStatusCode:200
Total ingested:179,ReqID:eaf9d2d7-e10b-47d1-b6e8-bae005b5b03f,HTTPStatusCode:200
Total ingested:179,ReqID:fbb011ff-b864-fcb6-a7a1-79c85cda0b58,HTTPStatusCode:200
Total ingested:179,ReqID:ce82bfdc-625c-8069-9293-d7eb86e27787,HTTPStatusCode:200
Total ingested:179,ReqID:ce8e524e-71ae-9c0b-929f-3a7995106be5,HTTPStatusCode:200
Total ingested:179,ReqID:e06c7d17-f96a-2903-bc7d-15201dd4deed,HTTPStatusCode:200
Total ingested:179,ReqID:dc0e7356-1d09-32f5-801f-1b61f9b7c51b,HTTPStatusCode:200
```

Fig. 15: Testing the retrieval of data from the REST API

The script takes care of extracting and storing information about the precipitation and snowfall in the *Precipitation* table, and stores the minimum and maximum temperature in the *Temperature* table. Since we used default settings while creating the lambda function, we will stick to the script and function name as **lambda_handler**. The code is uploaded in the *Code* tab provided with the lambda service, as shown in Figure 12.

The lambda function runs the code for a set amount of time (default 3 seconds) before timing out. We change this setting to **1 minute** by editing the general configuration under the 'Configuration' tab, as shown in Figure 13.

In the final step, we create a **trigger** that invokes the **lambda_handler** whenever there is data present in the **input-stream**. For this purpose, we click on 'Add trigger' under the 'Function overview' section of the lambda function. We put the trigger as *Kinesis*, as shown in Figure 14.

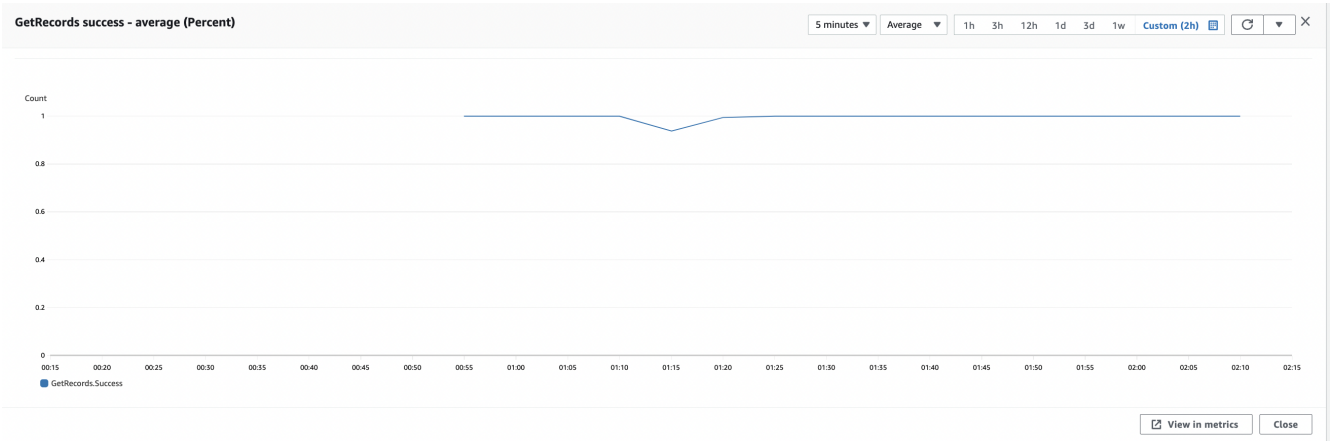

Fig. 16: Testing the presence of data in the kinesis data stream

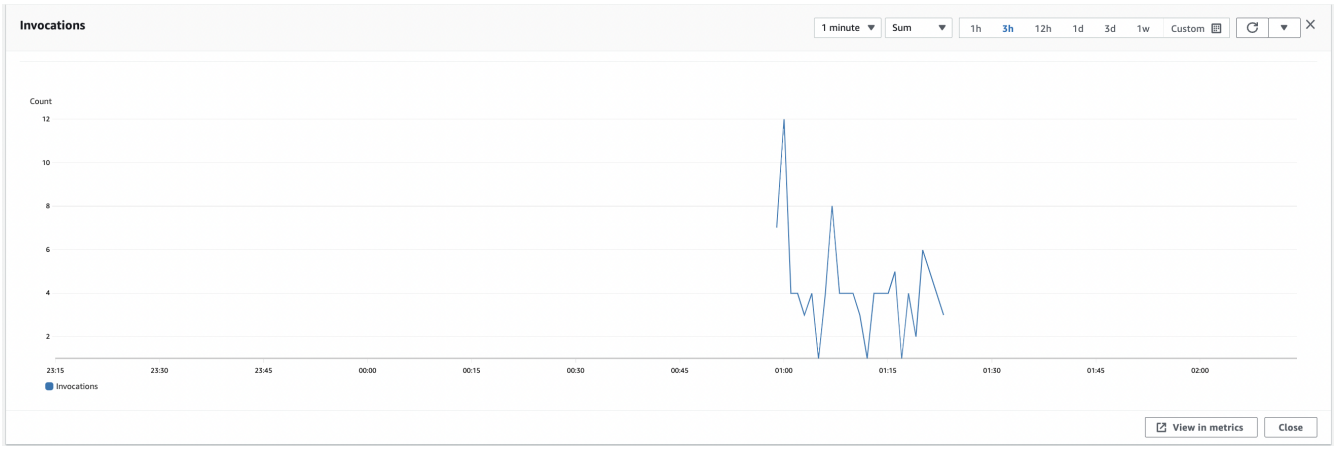

Fig. 17: Testing the invocations for the lambda function

VI. Testing

To test the workflow of the project, we run the script *putDataInStream* in the *project-kinesis* environment created on Cloud9. On running the script, the DynamoDB tables will get updated automatically, since the *KinesisLambdaConsumer* function will get triggered in the presence of data in the *input-stream*. Furthermore, we observe the AWS CloudWatch metrics for different services. The following results showcase the functioning of the project:

- As can be seen in Figure 7 and 15, the python3 terminal prints out the total ingested data from the REST API along with an HTTP status code as 200, verifying a successful data extraction from NOAA's website.
- Figure 16 illustrates a 100% average success rate of *GetRecords* operations, that is, getting data records from kinesis data stream's shard for the *input-stream*. The CloudWatch metric is available in the AWS kinesis service for a particular data stream.
- Figure 17 verifies that the lambda function was triggered over a specified amount of time.The CloudWatch metric is available in the AWS Lambda service for the *KinesisLambdaConsumer* function.
- The output of the project involves filling up the DynamoDB tables— *Precipitation* and *Temperature* and sorting the values using *station_name* and *date* as the partition and sort key, respectively. To illustrate the successful creation of table items, a query is run to obtain the weather parameters for *Beltsville* in chronological order, as shown in Figure 18 and Figure 19. The query is case-sensitive, hence, it is important to provide all locations in capitals.

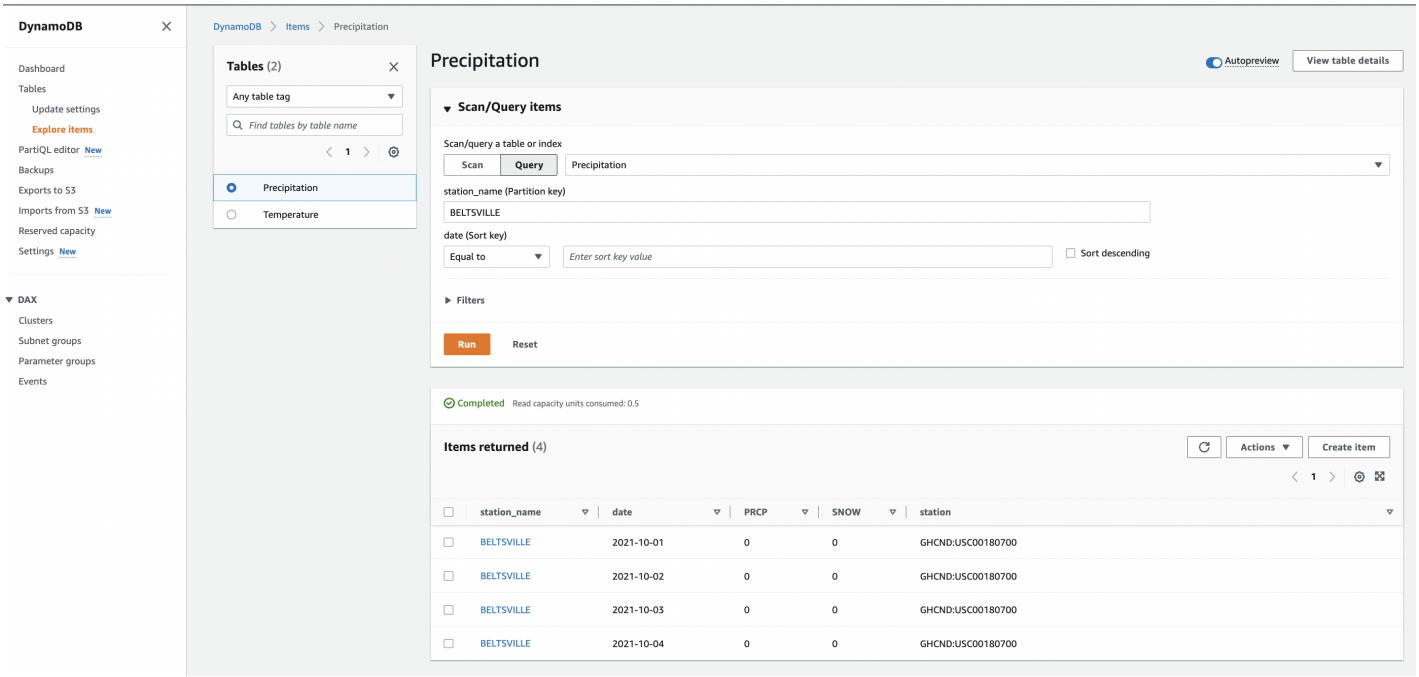

Fig. 18: Running query on the Precipitation table

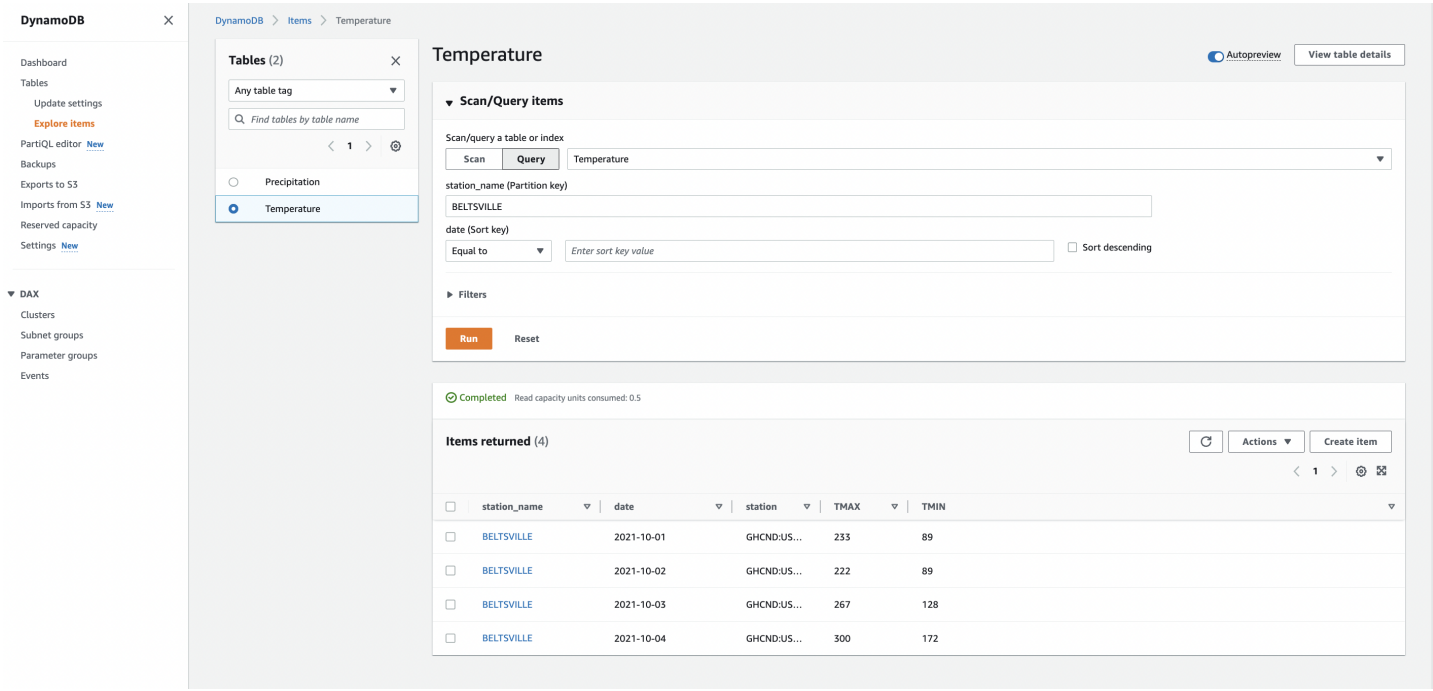

Fig. 19: Running query on the Temperature table