

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.

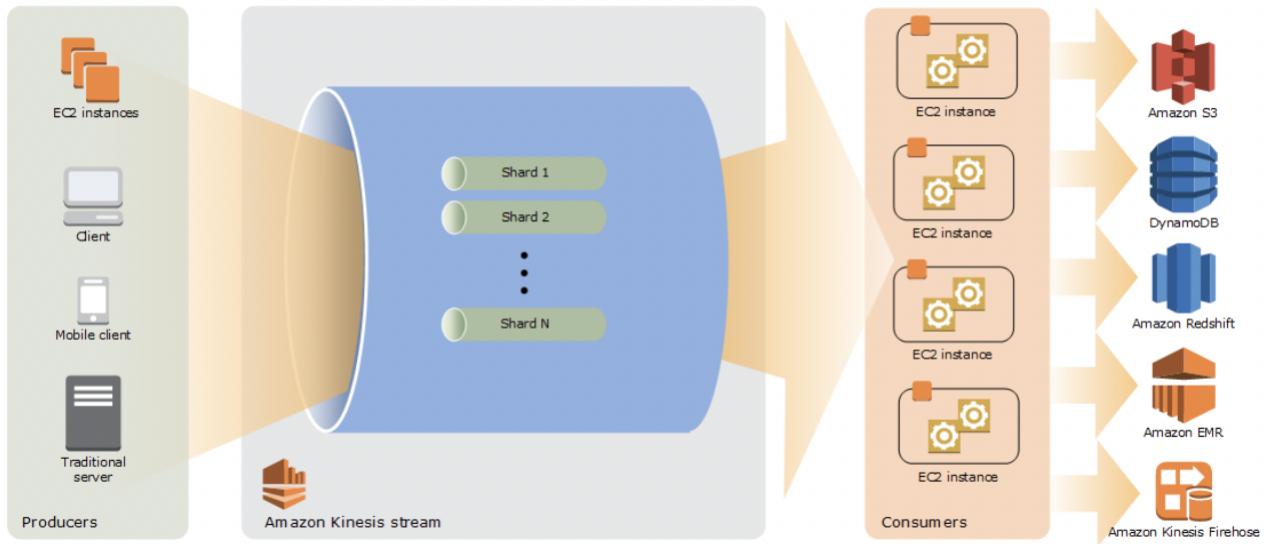


Fig. 1: Kinesis Data-Streams High Level Architecture

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

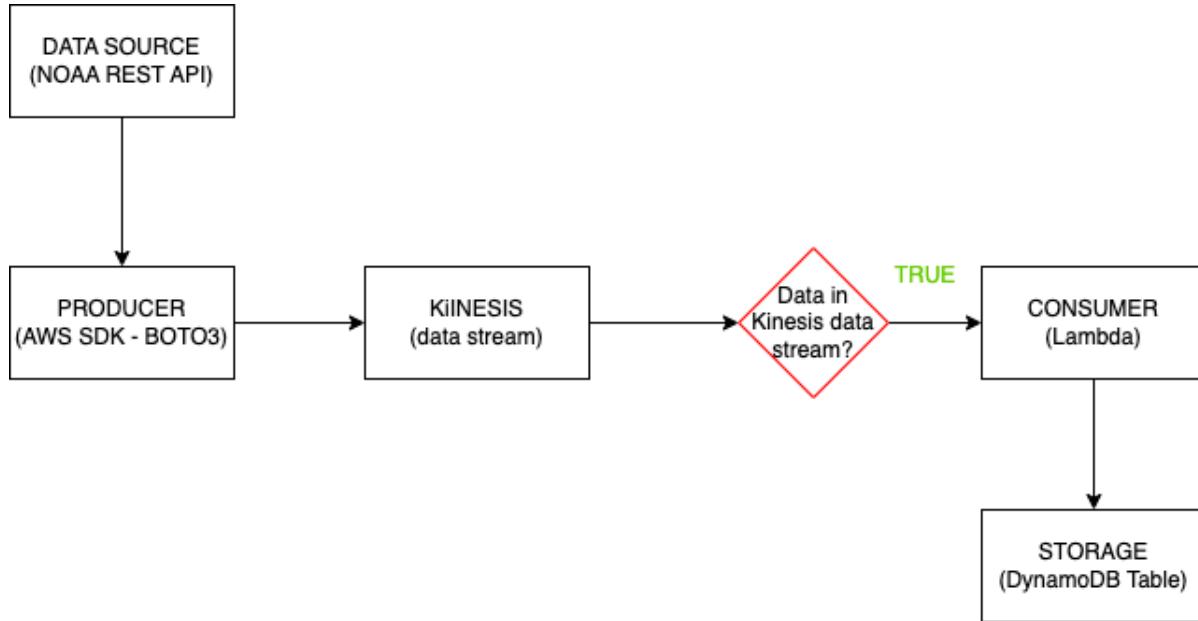


Fig. 3: Project Workflow

- 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.

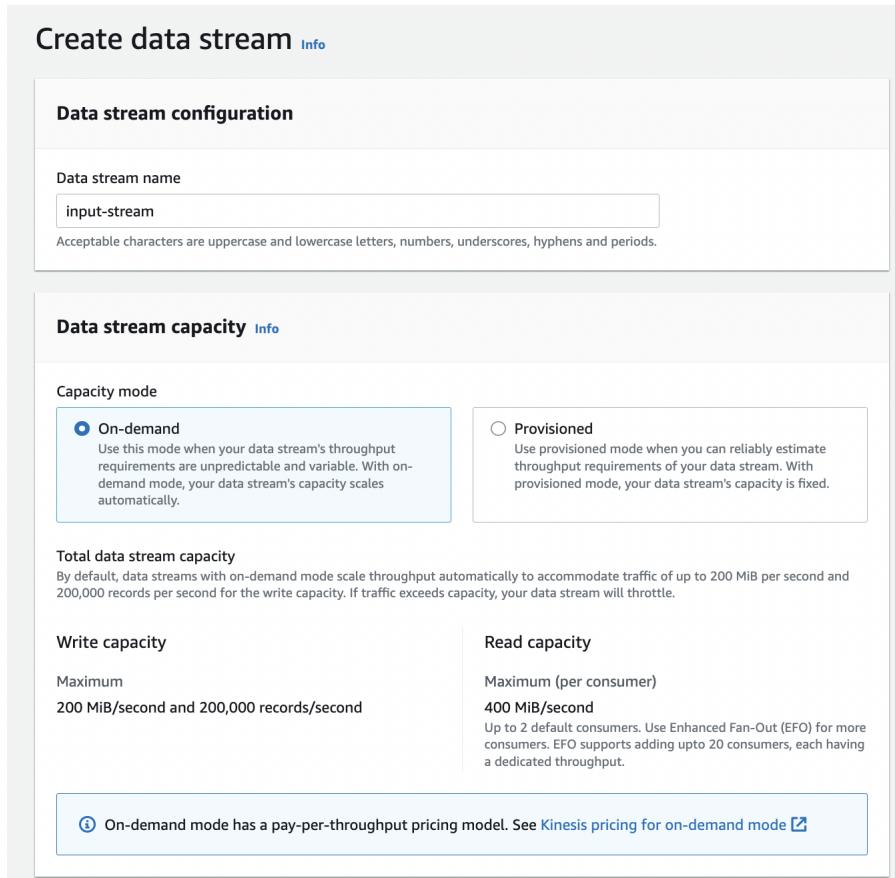


Fig. 4: Step 2: Create AWS Kinesis data stream

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,

Create environment Info

Details

Name Limit of 60 characters, alphanumeric, and unique per user.

Description - *optional* Limit 200 characters.

Environment type Info
Determines what the Cloud9 IDE will run on.

New EC2 instance
Cloud9 creates an EC2 instance in your account. The configuration of your EC2 instance cannot be changed by Cloud9 after creation.

Existing compute
You have an existing instance or server that you'd like to use.

New EC2 instance

Instance type Info
The memory and CPU of the EC2 instance that will be created for Cloud9 to run on.

t2.micro (1 GiB GiB RAM + 1 vCPU)
Free-tier eligible. Ideal for educational users and exploration.

t3.small (2 GiB GiB RAM + 2 vCPU)
Recommended for small web projects.

m5.large (8 GiB GiB RAM + 2 vCPU)
Recommended for production and most general-purpose development.

Additional instance types
Explore additional instances to fit your need.

Platform Info
This will be installed on your EC2 instance. We recommend Amazon Linux 2.

Amazon Linux 2

Timeout
How long Cloud9 can be inactive (no user input) before auto-hibernating. This helps prevent unnecessary charges.

30 minutes

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:

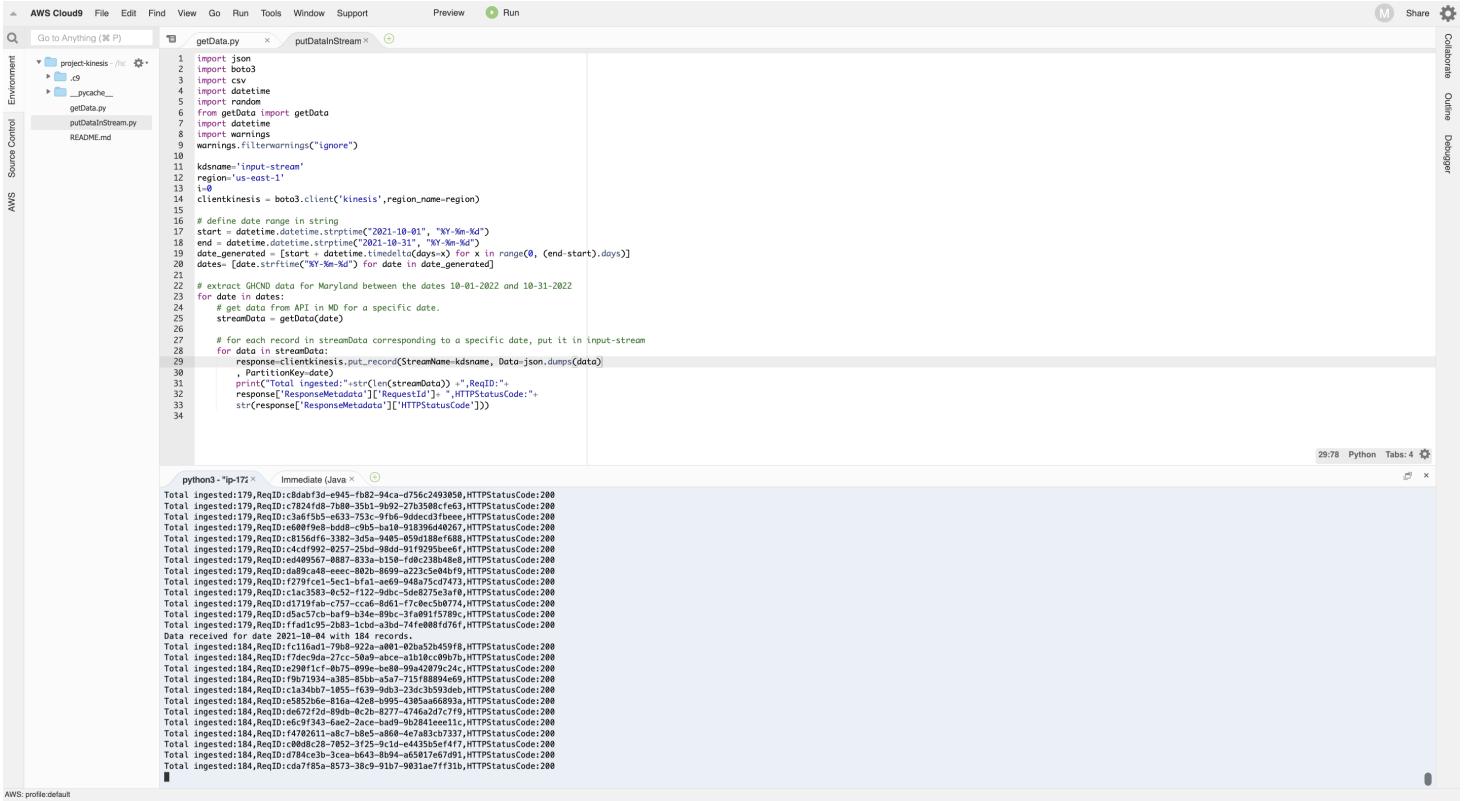
- CPU utilization and network traffic
- Memory and disk usage
- Request count and latency
- Errors and status codes

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.



The screenshot shows the AWS Cloud9 IDE interface. On the left, the file structure includes `project-kinesis`, `getdata.py`, `putdataInStream.py`, and `README.md`. The main editor window contains Python code for interacting with AWS Kinesis. The code defines a `putdataInStream` function that reads data from a file, processes it, and puts it into a Kinesis stream named "kinnessis". It uses the `boto3` library to interact with the service. The code also includes a section for extracting GHNO data for Maryland between specific dates. The bottom right corner shows the Cloud9 status bar with "29:78 Python Tabs: 4".

```

import json
import boto3
import time
import datetime
import random
from getdata import getData
import logging
import warnings
warnings.filterwarnings("ignore")

# Set the stream name and region
ksname="input-stream"
region="us-east-1"

# Define date range in string
start = datetime.datetime.strptime("2021-10-01", "%Y-%m-%d")
end = datetime.datetime.strptime("2022-10-31", "%Y-%m-%d")
date_generated = [start + datetime.timedelta(days=x) for x in range(0, (end-start).days)]
dates= [date.strftime("%Y-%m-%d") for date in date_generated]

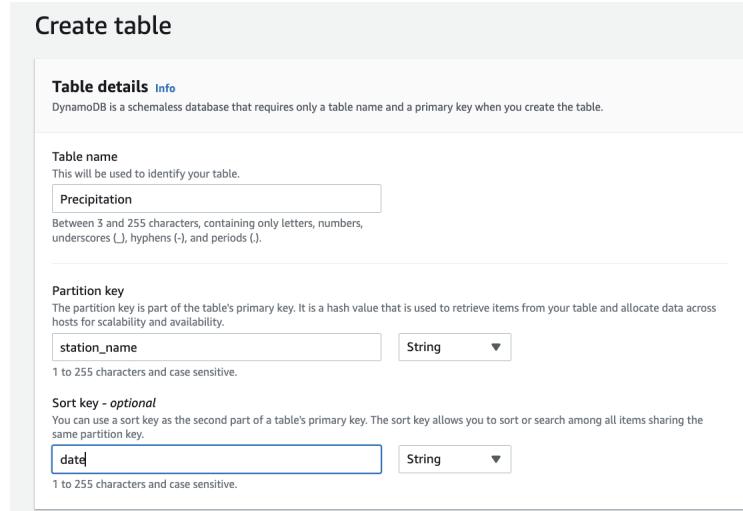
# Extract GHNO data for Maryland between the dates 10-01-2022 and 10-31-2022
for date in dates:
    # Get data from API in MD for a specific date.
    streamData = getData(date)

    # For each record in streamData corresponding to a specific date, put it in input-stream
    for data in streamData:
        clientkinesis.put_record(ksname, Data=json.dumps(data))

print("Total ingested:" + str(len(streamData)) + ",ReqID:" + response['ResponseMetadata']['RequestId'] + ",HTTPStatusCode:" + str(response['ResponseMetadata']['HTTPStatusCode']))

```

Fig. 6: Step 4: SDK producer on Cloud9



The screenshot shows the "Create table" wizard for DynamoDB. The first step, "Table details", is shown. The table name is set to "Precipitation". The primary key is defined with a "Partition key" named "station_name" (String type) and a "Sort key - optional" named "date" (String type). The wizard also notes that DynamoDB is a schemaless database.

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

The screenshot shows the AWS DynamoDB console interface. On the left, there's a sidebar with navigation links like Dashboard, Tables, Update settings, Explore items, PartiQL editor, Backups, Exports to S3, Imports from S3, Reserved capacity, and Settings. The main area is titled 'Tables (2) Info' and shows two tables: 'Precipitation' and 'Temperature'. Both tables are active and have 'date (S)' as the partition key and 'station (S)' as the sort key. They are provisioned with auto scaling for both read and write capacity modes. The table sizes are 0 bytes each, and they belong to the 'DynamoDB Standard' table class.

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.

KinesisLambdaConsumerRole

Allows Lambda functions to call AWS services on your behalf.

Creation date	ARN
December 11, 2022, 19:23 (UTC-05:00)	arn:aws:iam:354077325797:role/KinesisLambdaConsumerRole

Last activity

None

Maximum session duration

1 hour

Permissions Trust relationships Tags Access Advisor Revoke sessions

Permissions policies (2) Info

You can attach up to 10 managed policies.

Filter policies by property or policy name and press enter.

Policy name	Type	Description
LambdaFunctionPolicy	Customer managed	
AWSLambdaKinesisExecutionRole	AWS managed	Provides list and read access to Kinesis streams and write permissions to CloudWatch logs.

Actions Simulate Remove Add permissions

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

Create function Info

AWS Serverless Application Repository applications have moved to [Create application](#).

Author from scratch Start with a simple Hello World example.

Use a blueprint Build a Lambda application from sample code and configuration presets for common use cases.

Container image Select a container image to deploy for your function.

Basic information

Function name
Enter a name that describes the purpose of your function.
KinesisLambdaConsumer

Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime Info
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.
Python 3.9

Architecture Info
Choose the instruction set architecture you want for your function code.
 x86_64
 arm64

Permissions Info
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

Change default execution role

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions
 Use an existing role
 Create a new role from AWS policy templates

Existing role
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.
KinesisLambdaConsumerRole

[View the KinesisLambdaConsumerRole role on the IAM console](#).

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.

The screenshot shows the AWS Cloud9 IDE interface. At the top, there are tabs for Code, Test, Monitor, Configuration, Aliases, and Versions. Below that is a navigation bar with File, Edit, Find, View, Go, Tools, Window, Test, Deploy, and a search bar labeled 'Go to Anything (⌘ P)'. On the left, there's a sidebar titled 'Environment' with a dropdown menu showing 'KinesisLambdaCons' and 'lambda_function.py'. The main area displays the code for 'lambda_function.py'.

```

1 import base64
2 import json
3 import os
4 #python lib for aws
5 import boto3
6 import datetime
7 from decimal import Decimal
8
9
10 def lambda_handler(event, context):
11     """
12     Receive a batch of events from Kinesis and insert into DynamoDB table
13
14     """
15
16     try:
17         my_region = os.environ['AWS_REGION']
18         ##resource assigned from boto library
19         dynamo_db = boto3.resource('dynamodb', region_name=my_region)
20         ##dynamodb table name
21         table_precipitation = dynamo_db.Table('Precipitation')
22         table_temperature = dynamo_db.Table('Temperature')
23
24         for record in event["Records"]:
25             decoded_data = base64.b64decode(record["kinesis"]["data"]).decode("utf-8") # this returns a string
26
27             # replace NaN in the string with "NaN"
28             decoded_data_handle_NaN = decoded_data.replace("NaN", "\u00b7NaN\u00b7")
29
30             # convert json string into python object
31             decoded_data_dic = json.loads(decoded_data_handle_NaN, parse_float=Decimal)
32
33             # generate data for precipitation and temperature and store them in respective tables

```

At the bottom right, there are status indicators: 1:1 Python Spaces: 4.

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.micro 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:

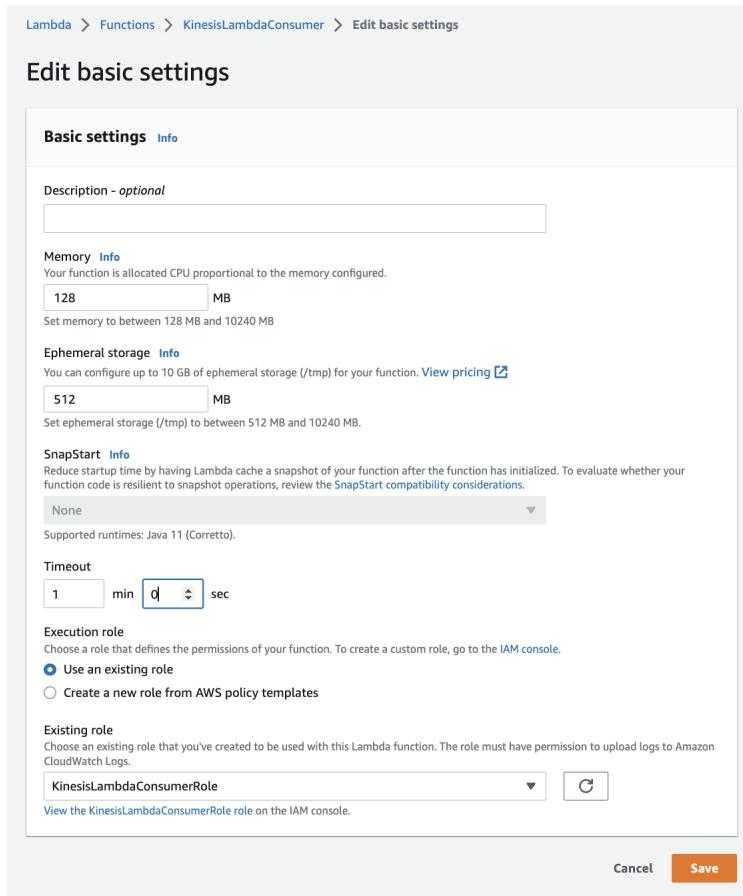


Fig. 13: Step 7: Changing timeout setting

- 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.

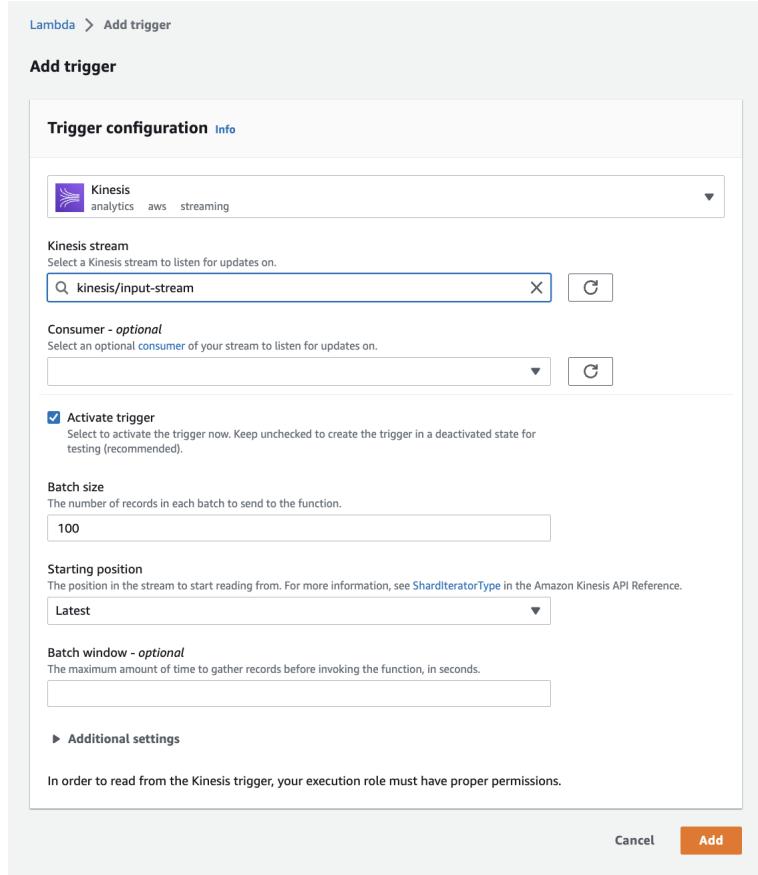


Fig. 14: Step 7: Adding trigger to the lambda consumer

```
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:ea9fd2d7-e10b-47d1-b6e8-bae005b5b03f,HTTPStatusCode:200
Total ingested:179,ReqID:fb0b11ff-b864-fcb6-a7a1-79c85cd0b58,HTTPStatusCode:200
Total ingested:179,ReqID:ce82bfdc-d625c-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.

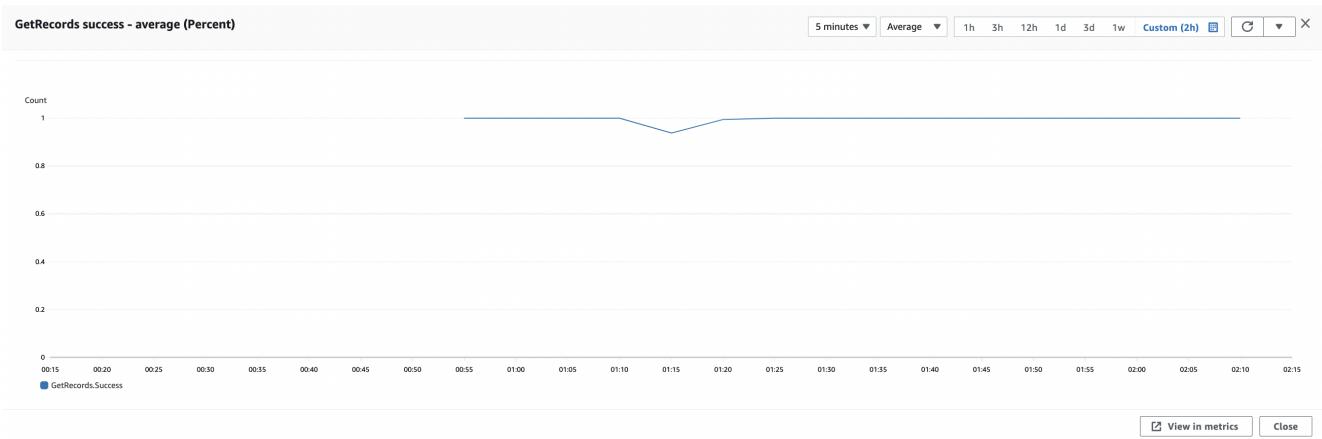


Fig. 16: Testing the presence of data in the kinesis data stream

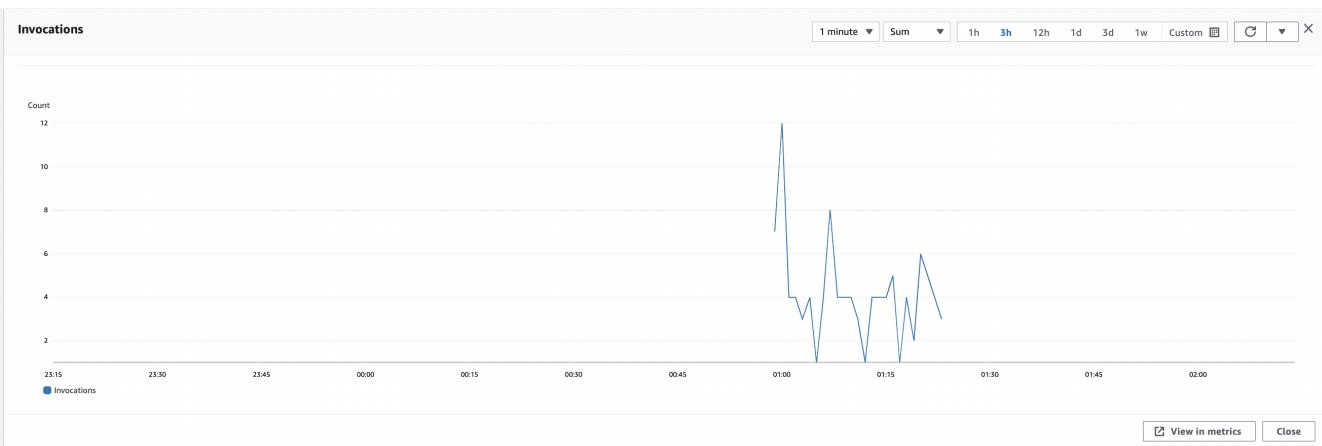


Fig. 17: Testing the invocations for the lambda function

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.

The screenshot shows the AWS DynamoDB console interface. On the left, there's a sidebar with navigation links like Dashboard, Tables, Update settings, Explore items (which is highlighted in orange), PartiQL editor, Backups, Exports to S3, Imports from S3, Reserved capacity, and Settings. Below that is a section for DAX with Clusters, Subnet groups, Parameter groups, and Events.

The main area shows the 'Precipitation' table details. It has two items: 'Precipitation' (selected) and 'Temperature'. The 'Scan/Query items' section is active, showing a query for 'station_name (Partition key)' set to 'BELTSVILLE' and 'date (Sort key)' set to 'Equal to'. There are 'Run' and 'Reset' buttons. A message at the bottom says 'Completed' with a note about consumed read capacity units.

The 'Items returned (4)' section displays the following data:

	station_name	date	PRCP	SNOW	station
<input type="checkbox"/>	BELTSVILLE	2021-10-01	0	0	GHCND:USC00180700
<input type="checkbox"/>	BELTSVILLE	2021-10-02	0	0	GHCND:USC00180700
<input type="checkbox"/>	BELTSVILLE	2021-10-03	0	0	GHCND:USC00180700
<input type="checkbox"/>	BELTSVILLE	2021-10-04	0	0	GHCND:USC00180700

Fig. 18: Running query on the Precipitation table

This screenshot is similar to Fig. 18 but shows the 'Temperature' table instead. The sidebar and table selection logic are identical.

The 'Scan/Query items' section shows a query for 'station_name (Partition key)' set to 'BELTSVILLE' and 'date (Sort key)' set to 'Equal to'. The 'Run' button is visible.

The 'Items returned (4)' section displays the following data:

	station_name	date	station	TMAX	TMIN
<input type="checkbox"/>	BELTSVILLE	2021-10-01	GHCND:US...	233	89
<input type="checkbox"/>	BELTSVILLE	2021-10-02	GHCND:US...	222	89
<input type="checkbox"/>	BELTSVILLE	2021-10-03	GHCND:US...	267	128
<input type="checkbox"/>	BELTSVILLE	2021-10-04	GHCND:US...	300	172

Fig. 19: Running query on the Temperature table