SFL Techincal Assessment

By Keith L Baskerville Jr.

GitHub Repo

Section 1: Consulting Soft Skills

A key function of this position is producing or contributing to functional specifications and other written or presentation quality technical documentation. Please provide responses that address the following hypothetical "client concerns". These responses should illustrate your personal writing style and your ability to provide clients with cogent and technically relevant information.

- ***What is a Data Lake? Explain its benefits, how it differs from a data warehouse, and how it might benefit a client.***

A Data Lake is **a special type of database** that enables users **to store all data types** including structured and unstructured data. Such as videos, images, documents, and other source file types. For example, a Data Lake is used as a **central storage location to hold data in its raw format** and is designed to allow users to hold a large amount of data without restriction. This allows users to quickly transform raw data to **improve the performance of analytics, data science, and machine learning**. Data Lakes are a popular choice for enterprises today because of their ability to store a variety of data **at a low cost.** Depending on your organization's approach, Data Lakes are most beneficial for MLOps practices **to collect and model** a diverse set of data sources; that users are doing to inform organizations **on how to develop and improve intelligent applications.**

In comparison to a Data Warehouse, Data Lakes are more affordable and flexible due to the ability to scale all data types. However, without the proper tools in place, relying on a Data Lake alone could potentially lead to issues including **reliability, slow performance, and the lack of security.** As the size of the Data Lake increases, it becomes more difficult for users to organize and catalog the data. Enterprises benefit the most when Data Lakes are **leveraged alongside the organization's structured database i.e. a database table to ensure high-quality and reliable data**. This means the data saved can be stored and organized in Databases or Data Warehouses. The key difference between a Data Lake and Data Warehouse **is the user's ability to access data for reporting and visualizing data queries**. Data Warehouses are used to improve the performance of accessing large amounts of data. A common use case for most MLOps teams would be to leverage Data Lakes to store any amount of Data at a low cost and a Data Warehouse to quickly structure data for easy access for its users.

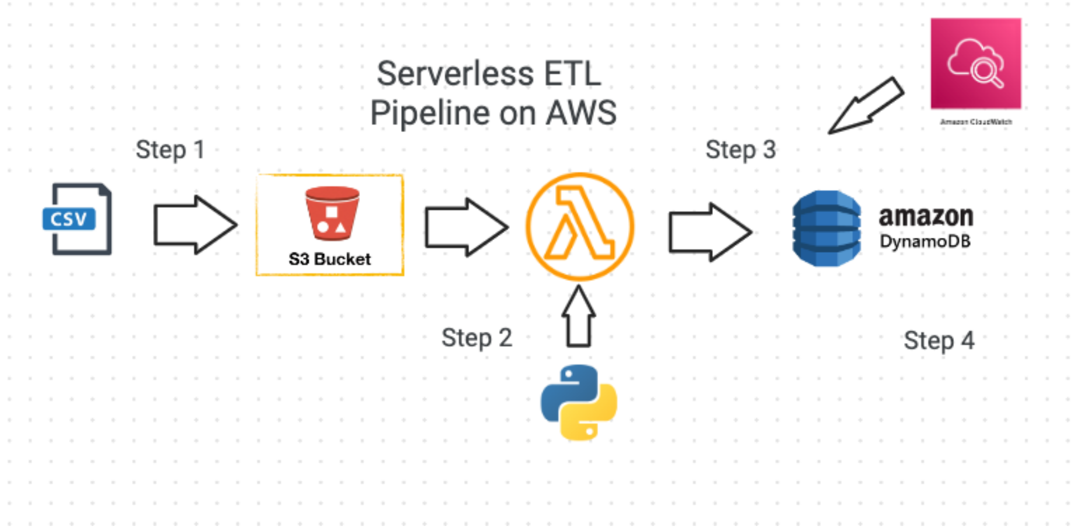
- ***Explain serverless architecture. What are its pros and cons?***

As Cloud computing gains more and more popularity, Enterprises have been transitioning from on-premise to Cloud-based infrastructure **to save costs. Serverless Architecture** is a cloud computing service that enables organizations to adopt a **"pay what you use"** model. Serverless Architecture helps developers design an approach to build and run software **fast and efficiently**. This means developers can focus on code and deployment while leaving the maintenance of the traditional hardware infrastructure to the cloud-based providers. For example, Serverless Architecture is leveraged **to run and maintain servers and applications, store data, and scale on-demand**. Some significant benefits for Enterprises that move towards Serverless computing include: **lowering the cost of unused servers, scalability to downsize or increase in response to an application's traffic, and the improvement of productivity.**

However, as developers spend less time managing infrastructure on-prem, there are a few drawbacks associated with Serverless Architecture. Such as a loss of control, security, performance, and testing. For example**, issues pertaining to hardware, data center outage,** or other problems associated with the server, fall on the hands of the cloud provider to solve. This is why most Serverless use cases are provisioned to complete **small repeatable tasks and to offload workloads.** Most developers migrate to Serverless computing by moving parts of an application **one at a time.** Serverless

Architecture is best used t**o perform triggered-based tasks, build RestFul APIs, and implement automation in stages of a CI/CD Pipeline.** It is also important to keep in mind each cloud provider may set certain limits and capabilities of its serverless services and would be best to consult with a Solution Architect to find the best solution to meet your Enterprises needs.

- **Please provide a diagram for an ETL pipeline (ex: Section 2) using serverless AWS services. Describe each component and its function within the pipeline.**

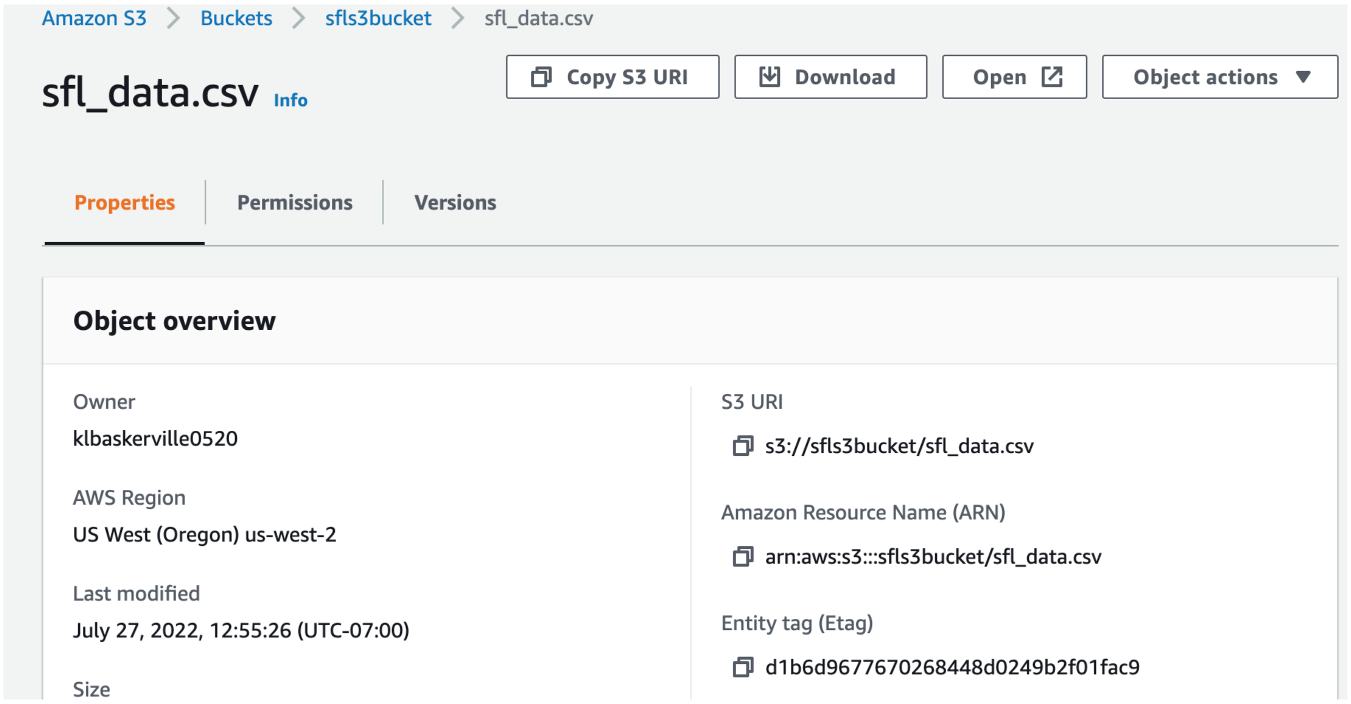


The diagram above illustrates how we can use **AWS Serverless Architecture to read data from a CSV file and store its contents in a database every time a file is uploaded.** This solution includes the following AWS Services:

- **Amazon Free Tier Account**
- **Amazon S3 Bucket -** Simple Storage for us to load CSV file(Cloud Object storage)
- **Amazon Lambda Function -** import python SDK to use boto3 and CSV packages, creates triggers for S3 bucket to read data from CSV file and store into our DynamoDB with Python Scripts
- **Amazon DynamoDB -** Stores are transformed data into a database.
- **Amazon CloudWatch- allows you to monitor your data across AWS resources. CloudWatch collects, reads, and accesses data**

Step 1 Create and Name S3 Bucket(Add and Attach Policies)

Create Policy for Lambda Function and create S3 Bucket to load the provided CSV file. Once we create the policies we can attach them to a role. The image below highlights the created S3 bucket and uploads the CSV file.



Amazon S3 > Buckets > sfls3bucket > sfl_data.csv

sfl_data.csv Info

Object overview

Owner
klbaskerville0520

AWS Region
US West (Oregon) us-west-2

Last modified
July 27, 2022, 12:55:26 (UTC-07:00)

Size

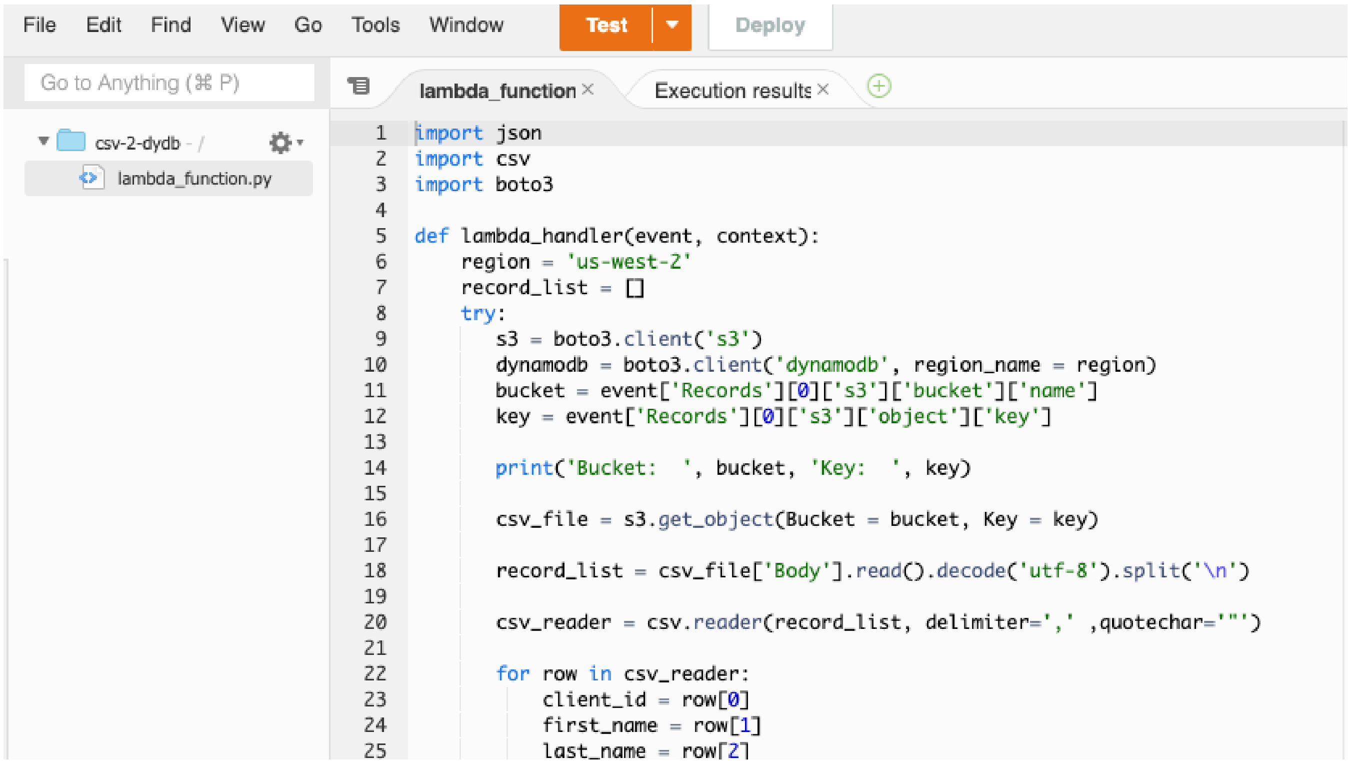
S3 URI
s3://sfls3bucket/sfl_data.csv

Amazon Resource Name (ARN)
arn:aws:s3:::sfls3bucket/sfl_data.csv

Entity tag (Etag)
d1b6d9677670268448d0249b2f01fac9

Step 2 Create Python Lamdba Function

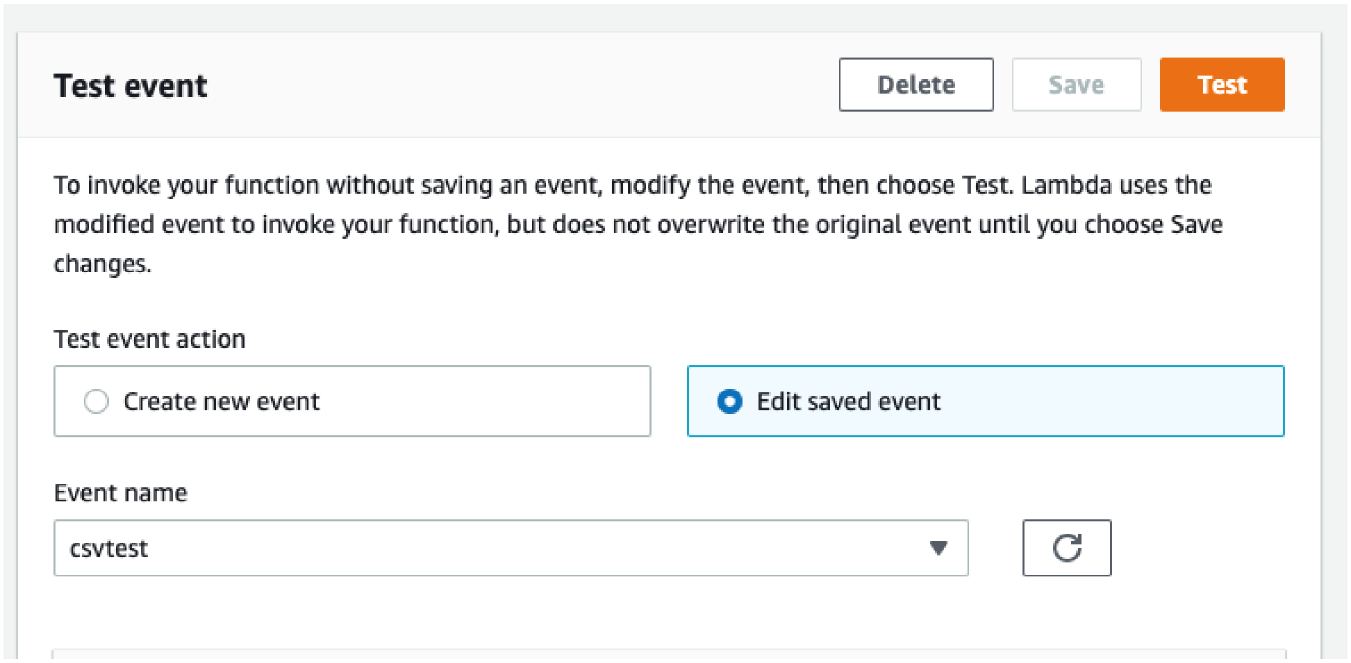
Once we configure our Lamda Function we can import boto3 to invoke the client (‘s3). We will also import CSV, a python package that allows us to read CSV files when we trigger our function.



```python
import json
import csv
import boto3

def lambda_handler(event, context):
    region = 'us-west-2'
    record_list = []
    try:
        s3 = boto3.client('s3')
        dynamodb = boto3.client('dynamodb', region_name = region)
        bucket = event['Records'][0]['s3']['bucket']['name']
        key = event['Records'][0]['s3']['object']['key']

        print('Bucket:  ', bucket, 'Key:  ', key)

        csv_file = s3.get_object(Bucket = bucket, Key = key)

        record_list = csv_file['Body'].read().decode('utf-8').split('\n')

        csv_reader = csv.reader(record_list, delimiter=',' ,quotechar='"')

        for row in csv_reader:
            client_id = row[0]
            first_name = row[1]
            last_name = row[2]
```

Step 3 Upload CSV File to test First Trigger

The first Lambda Trigger we configured reads the content from the CSV upload to our S3 bucket. If successful, we will receive a message in our Execution log. Inside the Lambda dashboard, we can create test events using JSON format. For the first test, we change the bucket name and the key name in the object field to the name of the CSV file.


Test event

Delete | Save | Test

To invoke your function without saving an event, modify the event, then choose Test. Lambda uses the modified event to invoke your function, but does not overwrite the original event until you choose Save changes.

Test event action

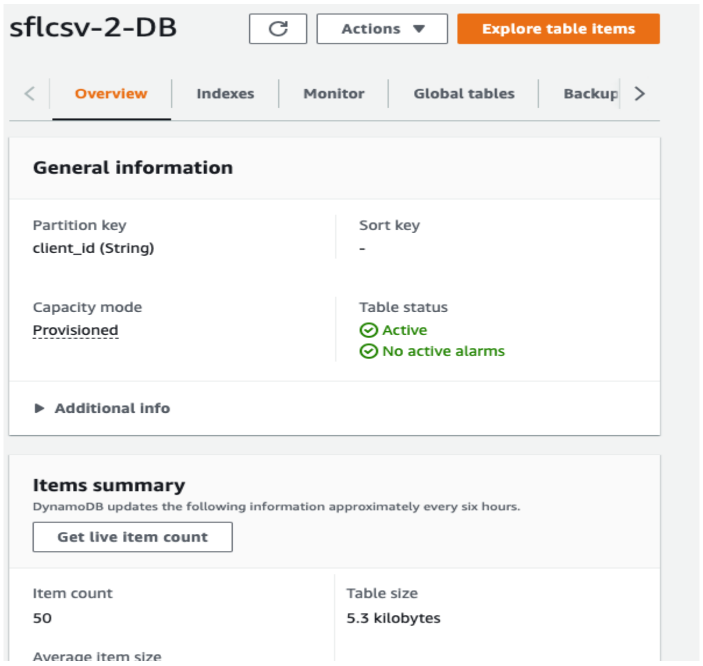
Create new event [✓ Edit saved event]

Event name

csvtest

Step 4 Create DynamoDB Table and Create Trigger to store Data

Once we see our content in the log, we can move on to create our database using DynamoDB Table. Once we create the table and give it a name, we can write Python code in our Lambda function to store the data in the table once triggered. Just like we created a client for an S3 we also want to create one for our database. Inside our DynamoDB dashboard, we can view insights with CloudMetrics to visualize metrics and view logs. Additionally, we can see if our table is active and can scan the database to see if there are any items stored. Below is a screenshot of the DynamoDB Table showing the status of Active and a live item count of 50 items.



sflcsv-2-DB

General information

Partition key: client_id (String)

Sort key: -

Capacity mode: Provisioned

Table status: Active, No active alarms

Items summary

DynamoDB updates the following information approximately every six hours.

Item count: 50

Table size: 5.3 kilobytes

Average item size

To review, the diagram and steps above walkthrough of how to use **AWS Lambda & S3 Bucket to Automate CSV File Processing. The S3 Bucket then pushes In DynamoDB Using Python Lambda Function.** In Section 2, I will walk step by step on how to configure the Serverless ETL Pipeline solution above. I will also provide a reproducible code for python lambda triggers(S3 Bucket and DynamDB).

- **Describe modern MLOps and how organizations should be approaching management from a tool and system perspective.**

Machine Learning Operations or **MLOps** is best described as the application of DevOp tools and practices to the Machine Learning workflow. The DevOps lifecycle includes the combination of software development and Operations: plan, build, code, test, release, deploy, operate, and monitor. However, the key difference between DevOps and MLOps is their focus **on monitoring and the deployment of ML models**. As DevOps builds the bridge for Software Developers and Operations teams, MLOps **improves communication between Data Scientists and Operations**. The main goals of MLOps are to implement **faster experimentation and model development, faster deployment when we update models, and quality assurance.** Focusing on model performance, ML engineers automate processes to quickly run datasets through models to gain insights for Enterprises.

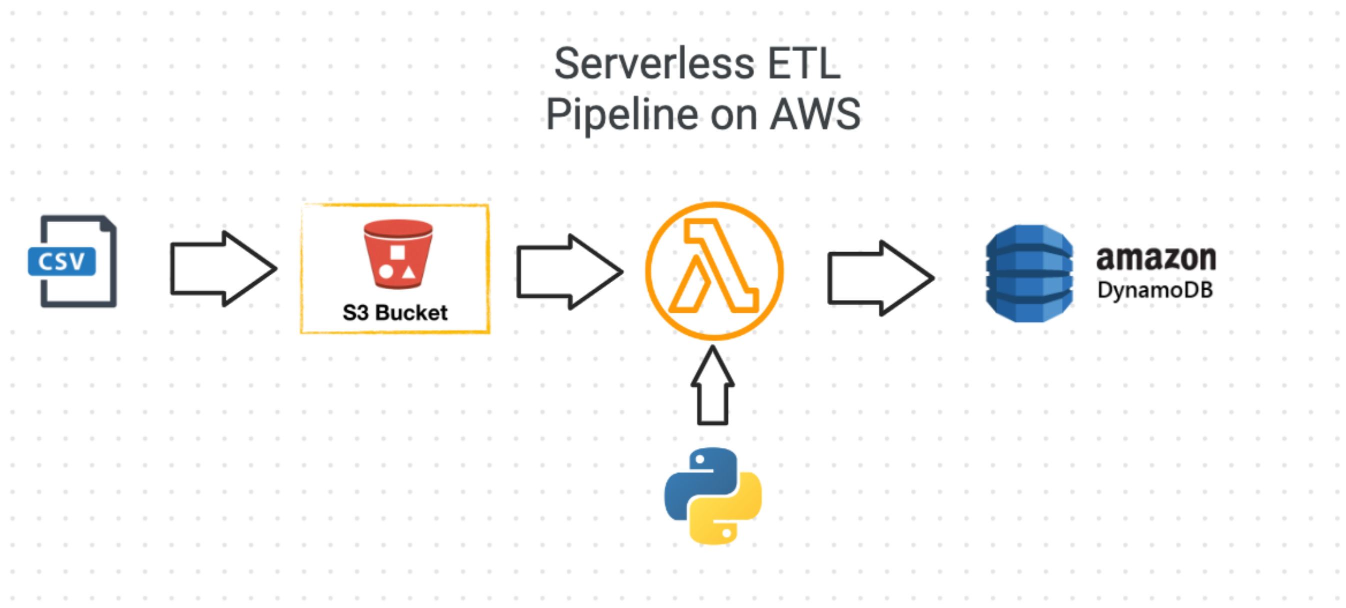
MLOps help businesses improve system management tools and practices in more than one way. The Benefits of implementing MLOps include **scalability and testing**, which enables organizations to create small repeatable tasks through automation, testing, and validation in high production environments. MLOps also make **better use of data and can enhance productivity in the work environment.** MLOps systems also **increase communication between teams and allow collaboration to combine team skill sets.** Enterprises implementing MLOps **produce efficient and dynamic production pipelines** as the process is integrated seamlessly. As responsibilities are shared, the Data Scientist and Operations teams work together **to reduce the risk and bias of unreliable or poorly designed models.** In addition to effectively meeting compliance and regulations, ML systems perspectives should be **collaborative, continuous, reproducible, tested, and monitored.**

Section 2: Database & Python ETL

Provision one database of your choosing (SQL, NoSQL, Graph). Write a python ETL that ingests the provided data, transforms it in some way, and loads it into the database. This should be reproducible code with documentation. (Terraform / Cloudformation / Ansible, docker-compose etc).

In this section, I will demonstrate how to load CSV data into AWS DynamoDB (NoSQL) using the Lambda Python function on AWS. To accomplish I will complete the following steps**:**

1. Create **an s3 bucket** and upload the CSV file to s3
2. Create **Lamda Function** Trigger using Python scripts
3. Provision NoSQL Database(**Amazon DynamoDB Table**)
4. **Test** the imported Data with **Python Lambda Function**
5. Test S3 event trigger **to push data into DyanmoDB**

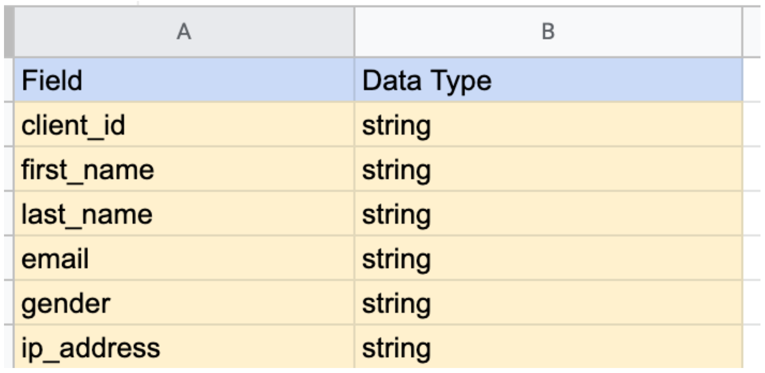


Once we will load the provided CSV file into an **Amazon simple storage service, S3**, We will create and invoke a **Python Lambda function.** Our Lambda function will read the CSV file and pass the information into a **NoSQL database, DynamoDB**. Our goal is **to trigger an event** every time a file is loaded into our S3 bucket and transform the data and store it in our database.

Step 1: Create Amazon S3 Bucket

After logging into my AWS Account, I created an **s3 bucket** with a unique name `sfls3bucket` and selected my region to be `us-west-2`. Next, I made a few adjustments to the provided CSV file. I renamed the file to `sfl_data.csv`, and changed the primary key from `id` to `client_id`.

CSV DATA



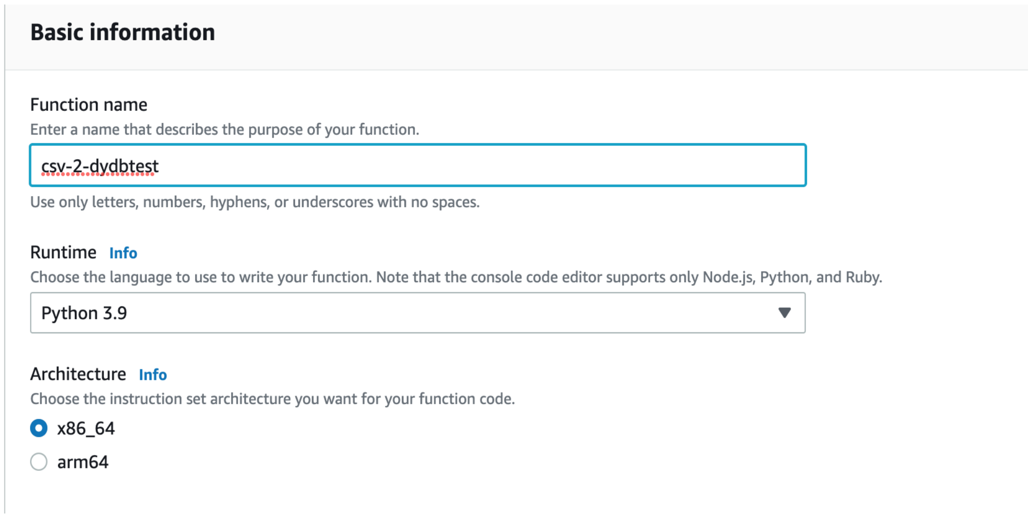
A	B
Field	Data Type
client_id	string
first_name	string
last_name	string
email	string
gender	string
ip_address	string

Next, I set up permissions for my Lambda function(which we will create in the next step). Navigate to the **IAM dashboard to create a new role(`csv-2-dydb`).** I added the following permissions to complete Section Two.

- **AmazonS3FullAccess**
- **CloudWatchFullAccess**
- **AmazonDynamoDBFullAccess**
- **AWSOpsWorksCloudWatchLogs**

Step 2: Create Python Lambda Function, Attach roles, and create S3 Trigger

In this step, I created a Python Lambda function to extract and transform our data. Before coding, I configured the Lambda function's runtime to Python 3.9 and attached the roles we created in Step One.


Basic information

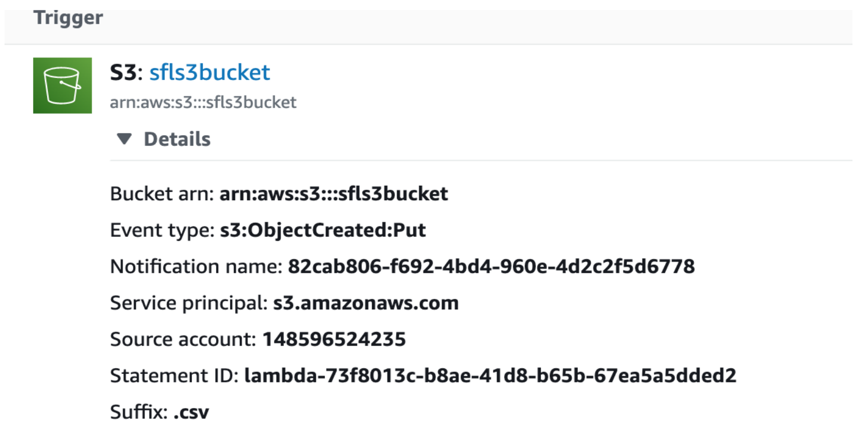
Function name
Enter a name that describes the purpose of your function.
csv-2-dydbtest
Use only letters, numbers, hyphens, or underscores with no spaces.

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

Let's break down our Lambda function into two main parts. The first part is reading the CSV file that is uploaded into the S3 Bucket. The second part is taking the uploaded data and writing it into DynamoDB.

Step 3: Test the imported Data with Lambda Function
Trigger Configuration gives us the ability **to add a Put Method**, which is an Event that triggers the lambda function when a .csv file(add in the suffix) is loaded into the S3 bucket.


Trigger

S3: sfls3bucket
arn:aws:s3:::sfls3bucket

▼ Details

Bucket arn: **arn:aws:s3:::sfls3bucket**
Event type: **s3:ObjectCreated:Put**
Notification name: **82cab806-f692-4bd4-960e-4d2c2f5d6778**
Service principal: **s3.amazonaws.com**
Source account: **148596524235**
Statement ID: **lambda-73f8013c-b8ae-41d8-b65b-67ea5a5dded2**
Suffix: **.csv**

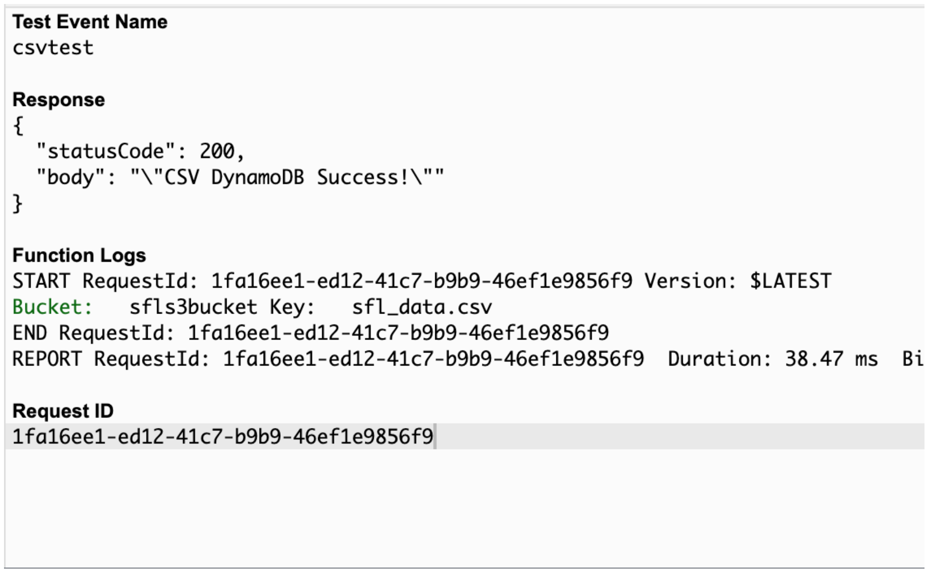
Next, I'm going to navigate to the Code tab in our Lambda function and write Python code. First I used the `csv` and `boto3` imports. The `csv` imports the libraries for python to read our csv files. Boto3 allows us to use Amazon SDK for python to create, configure, and manage AWS services.

```
import json
import csv
import boto3
```

```
def lambda_handler(event, context):
    region = 'us-west-2'
    record_list = []
    try:
       s3 = boto3.client('s3')
       bucket = event['Records'][0]['s3']['bucket']['name']
       key = event['Records'][0]['s3']['object']['key']

       print('Bucket:  ', bucket, 'Key:  ', key)

      Except Exception as e:
            print(str(e))

    return {
        'statusCode': 200,
        'body': json.dumps('CSV DynamoDB Success!')
    }
```

Now we can Test our trigger to see what happens when we upload a CSV file. We can create a test event in the Lambda function. Here I reformatted the JSON format to match the S3 bucket **"name"** and the name of the CSV file uploaded in **"key"**.

S3 Trigger Test 1



Test Event Name
csvtest

Response
```
{
  "statusCode": 200,
  "body": "\"CSV DynamoDB Success!\""
}
```

Function Logs
```
START RequestId: 1fa16ee1-ed12-41c7-b9b9-46ef1e9856f9 Version: $LATEST
Bucket:   sfls3bucket Key:   sfl_data.csv
END RequestId: 1fa16ee1-ed12-41c7-b9b9-46ef1e9856f9
REPORT RequestId: 1fa16ee1-ed12-41c7-b9b9-46ef1e9856f9  Duration: 38.47 ms  Bi
```

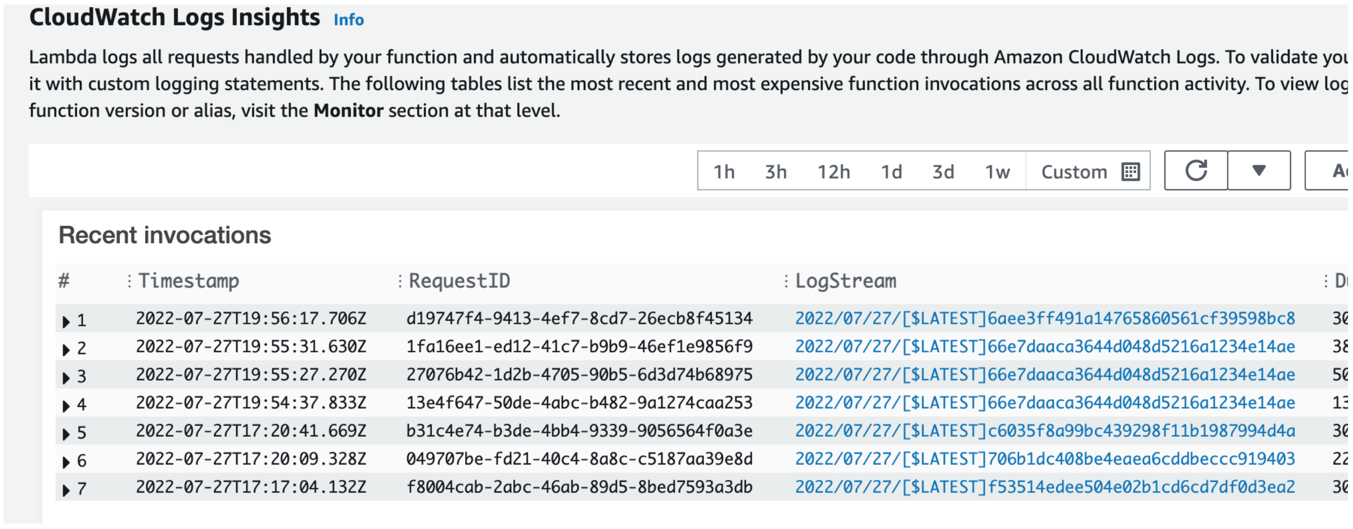
Request ID
1fa16ee1-ed12-41c7-b9b9-46ef1e9856f9

Json Format

```
"s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "sfls3bucket", // change bucket name here
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::example-bucket"
        },
        "object": {
          "key": "sfl_data.csv", // add uploaded csv file here
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
}
```

Our first test also allows us to view **Cloudwatch Log** Insights to see all requests by our functions

CloudWatch Logs



CloudWatch Logs Insights Info

Lambda logs all requests handled by your function and automatically stores logs generated by your code through Amazon CloudWatch Logs. To validate you
it with custom logging statements. The following tables list the most recent and most expensive function invocations across all function activity. To view log
function version or alias, visit the **Monitor** section at that level.

Recent invocations

#	Timestamp	RequestID	LogStream	D
1	2022-07-27T19:56:17.706Z	d19747f4-9413-4ef7-8cd7-26ecb8f45134	2022/07/27/[$LATEST]6aee3ff491a14765860561cf39598bc8	30
2	2022-07-27T19:55:31.630Z	1fa16ee1-ed12-41c7-b9b9-46ef1e9856f9	2022/07/27/[$LATEST]66e7daaca3644d048d5216a1234e14ae	38
3	2022-07-27T19:55:27.270Z	27076b42-1d2b-4705-90b5-6d3d74b68975	2022/07/27/[$LATEST]66e7daaca3644d048d5216a1234e14ae	50
4	2022-07-27T19:54:37.833Z	13e4f647-50de-4abc-b482-9a1274caa253	2022/07/27/[$LATEST]66e7daaca3644d048d5216a1234e14ae	13
5	2022-07-27T17:20:41.669Z	b31c4e74-b3de-4bb4-9339-9056564f0a3e	2022/07/27/[$LATEST]c6035f8a99bc439298f11b1987994d4a	30
6	2022-07-27T17:20:09.328Z	049707be-fd21-40c4-8a8c-c5187aa39e8d	2022/07/27/[$LATEST]706b1dc408be4eaea6cddbeccc919403	22
7	2022-07-27T17:17:04.132Z	f8004cab-2abc-46ab-89d5-8bed7593a3db	2022/07/27/[$LATEST]f53514edee504e02b1cd6cd7df0d3ea2	30

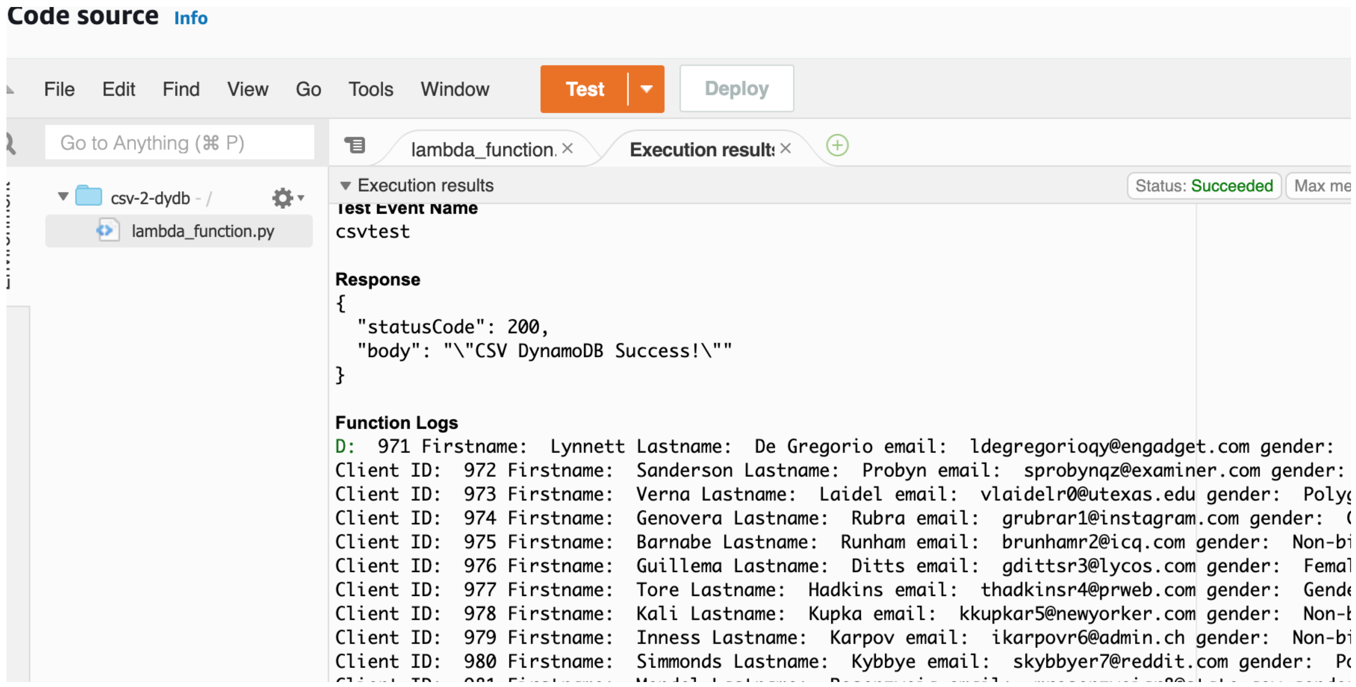
Trigger Test 2

Our first test checked to see if our bucket name and key name triggered when the CSV file was uploaded. The second test will be to read the **`boto.client('s3')`**, then it grabs the data in our CSV file. The code below gets the loaded object in the S3 bucket(`csv_file`)and puts the data from the CSV files in rows(` record_list`). Next, we use `csv_reader` to read each of the rows. And the last step for this step is to look for each of the rows using a for loop `for row in csv_reader:`

```
import json
import csv
import boto3

def lambda_handler(event, context):
    region = 'us-west-2'
    record_list = []
    try:
        s3 = boto3.client('s3')
        bucket = event['Records'][0]['s3']['bucket']['name']
        key = event['Records'][0]['s3']['object']['key']

        print('Bucket:  ', bucket, 'Key:  ', key)
        // grabs s3 bucket and file name
        csv_file = s3.get_object(Bucket = bucket, Key = key)
        // creates rows for list
        record_list = csv_file['Body'].read().decode('utf-8').split('\n')
        // read csv file
        csv_reader = csv.reader(record_list, delimiter=',' ,quotechar='"')
        //Looks at each row
        for row in csv_reader:
            client_id = row[0]
            first_name = row[1]
            last_name = row[2]
            email = row[3]
            gender = row[4]
            ip_address = row[5]

            print('Client ID: ', client_id, 'Firstname: ', first_name, 'Lastname:
', last_name, 'email: ', email, 'gender: ', gender, 'IP Address: ', ip_address)


            print('Successfully added the records to the DynamoDB Table')

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


    return {
        'statusCode': 200,
        'body': json.dumps('CSV DynamoDB Success!')
    }
```

Function Logs

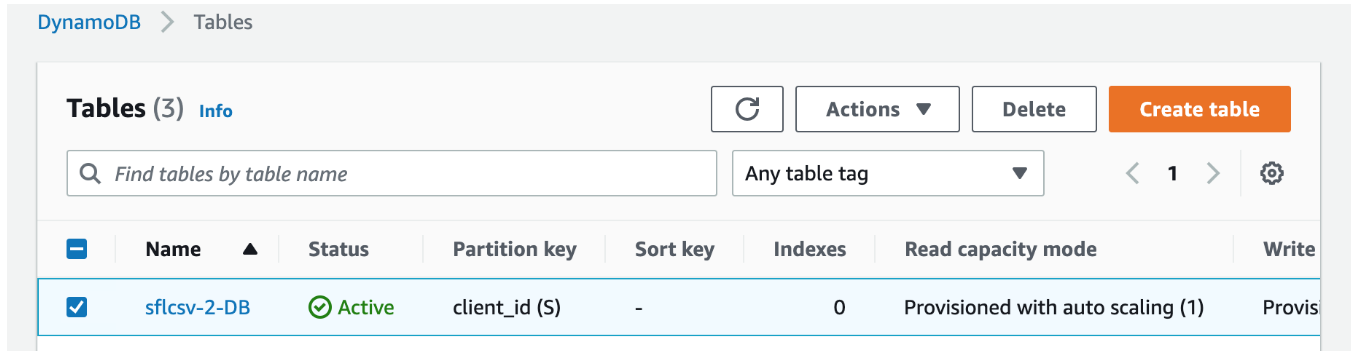
In the picture below, we can see our code printed the CSV file data! We have completed the first part of the Lambda function. We can now successfully upload CSV files in an S3 Bucket and read the contents in a Python Lambda Function. Before we complete the second part of Lambda Function lets first **provision our NoSQL database(DynamoDB)**.



```
Test Event Name
csvtest

Response
{
  "statusCode": 200,
  "body": "\"CSV DynamoDB Success!\""
}

Function Logs
D:  971 Firstname:  Lynnett Lastname:  De Gregorio email:  ldegregorioqy@engadget.com gender:
Client ID:  972 Firstname:  Sanderson Lastname:  Probyn email:  sprobynqz@examiner.com gender:
Client ID:  973 Firstname:  Verna Lastname:  Laidel email:  vlaidelr0@utexas.edu gender:  Poly
Client ID:  974 Firstname:  Genovera Lastname:  Rubra email:  grubrar1@instagram.com gender:  (
Client ID:  975 Firstname:  Barnabe Lastname:  Runham email:  brunhamr2@icq.com gender:  Non-bi
Client ID:  976 Firstname:  Guillema Lastname:  Ditts email:  gdittsr3@lycos.com gender:  Femal
Client ID:  977 Firstname:  Tore Lastname:  Hadkins email:  thadkinsr4@prweb.com gender:  Gende
Client ID:  978 Firstname:  Kali Lastname:  Kupka email:  kkupkar5@newyorker.com gender:  Non-b
Client ID:  979 Firstname:  Inness Lastname:  Karpov email:  ikarpovr6@admin.ch gender:  Non-bi
Client ID:  980 Firstname:  Simmonds Lastname:  Kybbye email:  skybbyer7@reddit.com gender:  Po
```

Step 4: Provision NoSQL Database(**Amazon DynamoDB Table**)

In AWS console search DynamoBD, and create a new table. When we create a table it is important to remember the name of the table name `sflcsv-2-DB` and primary key `client_id` (S) when we head back to our Lambda Function.



Name	Status	Partition key	Sort key	Indexes	Read capacity mode	Write
sflcsv-2-DB	Active	client_id (S)	-	0	Provisioned with auto scaling (1)	Provis

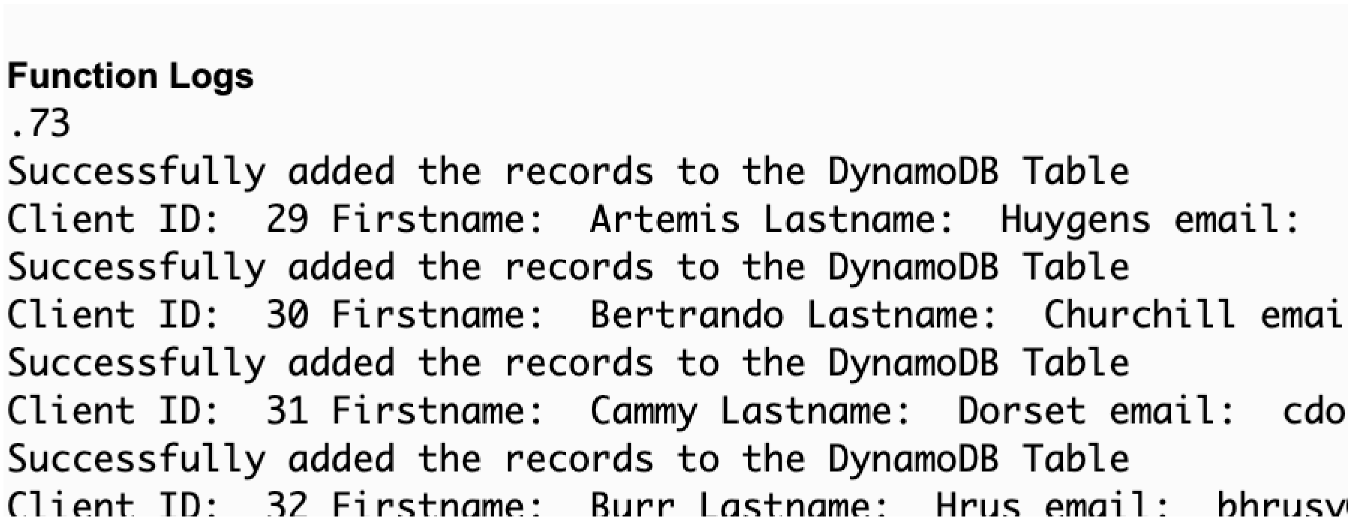
Let's head back to the Lambda Function to create a `dynamodb.client` similar to the `boto3.client` we created. Now that we have our table created we can write the Python script to take the object we load in the S3 bucket and put it in our newly created table(database). Copy and paste the snippet below for the pipeline to run from start to finish.

```
import json
import csv
import boto3

def lambda_handler(event, context):
    region = 'us-west-2'
    record_list = []
    try:
        s3 = boto3.client('s3')
// adds dynamo db table we created
```

```
        dynamodb = boto3.client('dynamodb', region_name = region)
        bucket = event['Records'][0]['s3']['bucket']['name']
        key = event['Records'][0]['s3']['object']['key']

        print('Bucket:  ', bucket, 'Key:  ', key)

        csv_file = s3.get_object(Bucket = bucket, Key = key)

        record_list = csv_file['Body'].read().decode('utf-8').split('\n')

        csv_reader = csv.reader(record_list, delimiter=',' ,quotechar='"')

        for row in csv_reader:
            client_id = row[0]
            first_name = row[1]
            last_name = row[2]
            email = row[3]
            gender = row[4]
            ip_address = row[5]

            print('Client ID: ', client_id, 'Firstname: ', first_name, 'Lastname:
', last_name, 'email: ', email, 'gender: ', gender, 'IP Address: ', ip_address)

            add_to_db = dynamodb.put_item( // adds cvs file to table
                TableName= 'sflcsv-2-DB',
                Item = {
                   'client_id' : {'S': str(client_id)}, // each row in csv table
                   'first_name' : {'S': str(first_name)},
                   'last_name' : {'S': str(last_name)},
                   'email' : {'S': str(email)},
                   'gender' : {'S': str(gender)},
                   'ip_address' : {'S': str(ip_address)},

                 })

            print('Successfully added the records to the DynamoDB Table')

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


    return {
        'statusCode': 200,
        'body': json.dumps('CSV DynamoDB Success!')
    }
```

Now that we have our Lambda function ready we can test our trigger again.

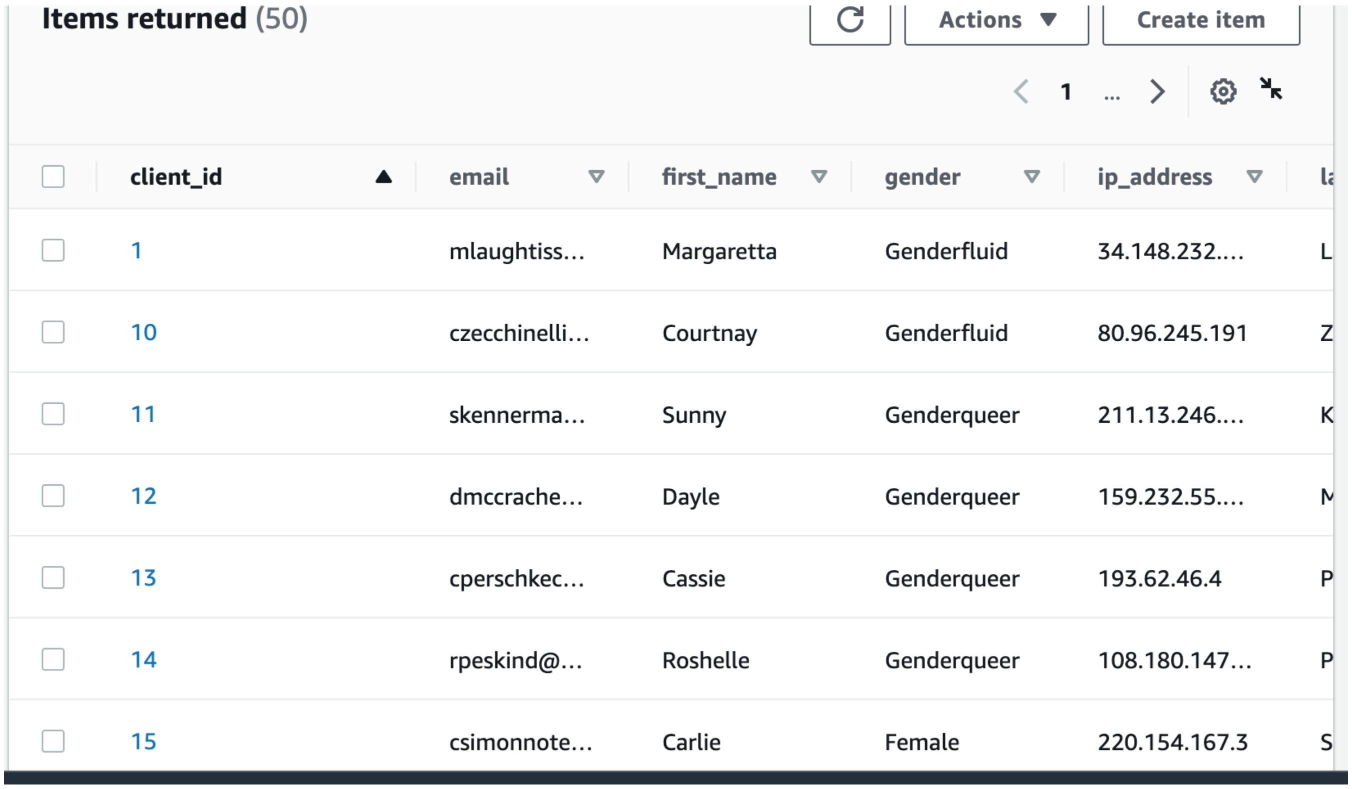
Step 5: Test S3 event trigger to import Data into DyanmoDB



```
Function Logs
.73
Successfully added the records to the DynamoDB Table
Client ID:  29 Firstname:  Artemis Lastname:  Huygens email:
Successfully added the records to the DynamoDB Table
Client ID:  30 Firstname:  Bertrando Lastname:  Churchill emai
Successfully added the records to the DynamoDB Table
Client ID:  31 Firstname:  Cammy Lastname:  Dorset email:  cdo
Successfully added the records to the DynamoDB Table
Client ID:  32 Firstname:  Burr Lastname:  Hrus email:  bhrusv
```

It looks like our script successfully passed. If we go back to our DynamoDB Table, we can see the CSV file was successfully stored in the database. We did it!

DynamoBD(NoSQL)



client_id	email	first_name	gender	ip_address	la
1	mlaughtiss...	Margaretta	Genderfluid	34.148.232....	L
10	czecchinelli...	Courtnay	Genderfluid	80.96.245.191	Z
11	skennerma...	Sunny	Genderqueer	211.13.246....	K
12	dmccrache...	Dayle	Genderqueer	159.232.55....	M
13	cperschkec...	Cassie	Genderqueer	193.62.46.4	P
14	rpeskind@...	Roshelle	Genderqueer	108.180.147...	P
15	csimonnote...	Carlie	Female	220.154.167.3	S

To overview what we accomplished, w**e used AWS's Serverless Cloud computing to Extract data from a CSV file and transform it and store it in a database**. We leveraged the **Amazon S3 bucket, Python Lambda function(Boto3 SDK, CSV package), and DynamoDB(NoSQL database).** For further explanation of each component and its functionality please refer to the diagram in Section One.

Section 3: ML API

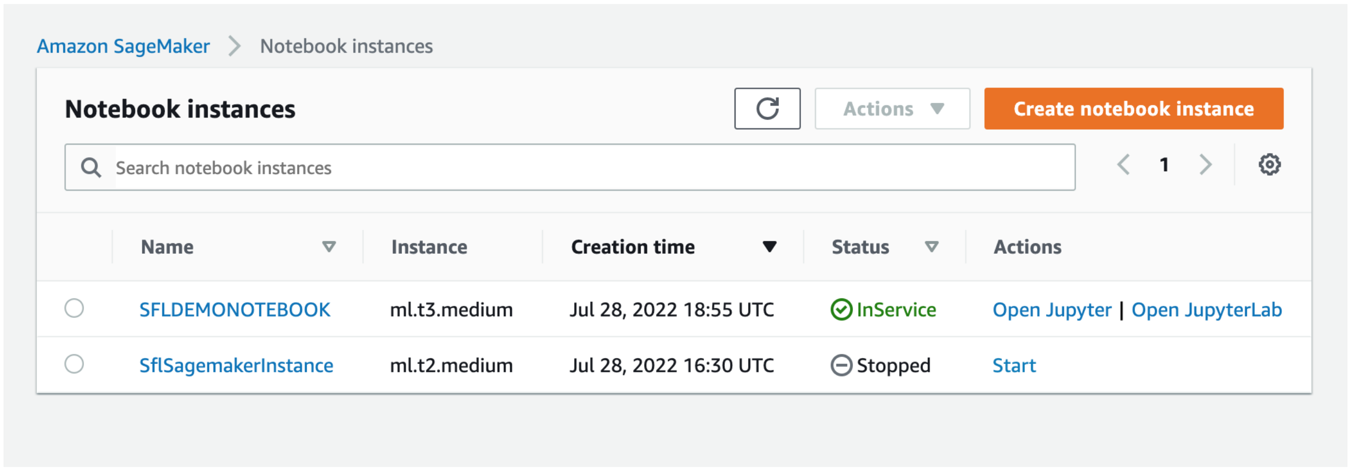
Deploy (on a cloud provider [AWS, Azure, etc] or locally [docker, kubernetes, etc]) an ML model (MNIST: https://paperswithcode.com/dataset/mnist, Fashion MNIST: https://github.com/zalandoresearch/fashion-mnist), as an API endpoint. Provide reproducible code with documentation for both deployment and usage.

In this section, I will use **AWS SageMaker and PyTorch to Experiment with MNIST Handwritten Digits Classification.** Check out my Github repo to see the created Notebook and reproducible I used to complete this task. Amazon Sagemaker and sagemaker-experiments are the resources I used to help me along the way. In this demonstration I will accomplish the following:

1. **Create a SageMaker notebook instance**
2. **Add GitHub Repo**
3. **Prepare the data**
4. **Set up Experiment and Train the model to learn from the data**
5. **Deploy the model**
6. **Evaluate your ML model's performance**

Steps 1 and 2: Create a SageMaker notebook instance and GitHub Repo

Amazon Sagemaker enables us to create notebooks to download and process data. In the process of creating our notebook, we can assign it a name, choose an Ec2 instance type, set permission, and select the S3 bucket. For the purposes of this demonstration, we will create in the S3 with our Python script inside the Notebook Kernal. We can also add our git hub repo(I cloned a repo I created for question 2). Once we created our notebook we can start to prepare the data.


Amazon SageMaker > Notebook instances
Notebook instances
Actions
Create notebook instance
Search notebook instances
1

Name	Instance	Creation time	Status	Actions
SFLDEMONOTEBOOK	ml.t3.medium	Jul 28, 2022 18:55 UTC	InService	Open Jupyter | Open JupyterLab
SflSagemakerInstance	ml.t2.medium	Jul 28, 2022 16:30 UTC	Stopped	Start

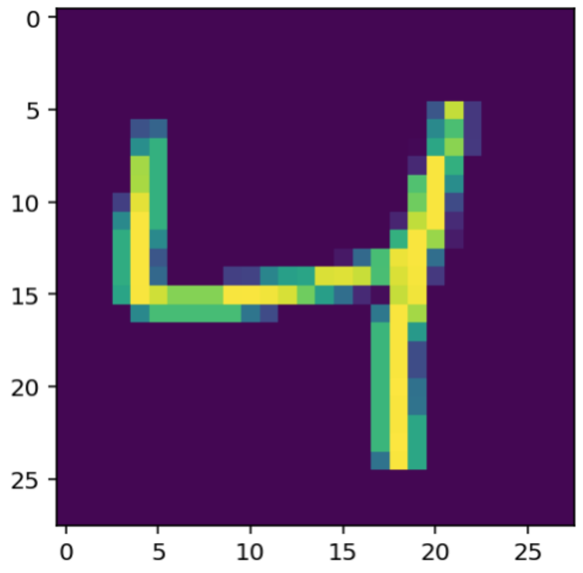
Step 3: Prepare the data

Now that we have our notebook status "inService" we can open Jupyter, a web-based computing platform that allows us to write code. Choose New, and `conda_amazonei_latest_p3` as the kernel. Next, we want to import our system modules and install the Sagemaker Experiment SDK. After we set up our environment we want to download the data. The code snippet below downloads the MNIST handwritten digits dataset, and then apply to transform each image.

```
bucket = sm_sess.default_bucket() // creates default bucket
prefix = "DEMO-mnist"
print("Using S3 location: s3://" + bucket + "/" + prefix + "/")

datasets.MNIST.urls = [ // loads Data

"https://sagemaker-sample-files.s3.amazonaws.com/datasets/image/MNIST/train-image
s-idx3-ubyte.gz",

"https://sagemaker-sample-files.s3.amazonaws.com/datasets/image/MNIST/train-label
s-idx1-ubyte.gz",

"https://sagemaker-sample-files.s3.amazonaws.com/datasets/image/MNIST/t10k-images
-idx3-ubyte.gz",

"https://sagemaker-sample-files.s3.amazonaws.com/datasets/image/MNIST/t10k-labels
-idx1-ubyte.gz",
]

# Download the dataset to the ./mnist folder, and load and transform (normalize)
them
train_set = datasets.MNIST(
    "mnist",
    train=True,
    transform=transforms.Compose(
        [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
    ),
    download=True,
)

test_set = datasets.MNIST(
    "mnist",
    train=False,
    transform=transforms.Compose(
        [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
    ),
    download=False,
)
```

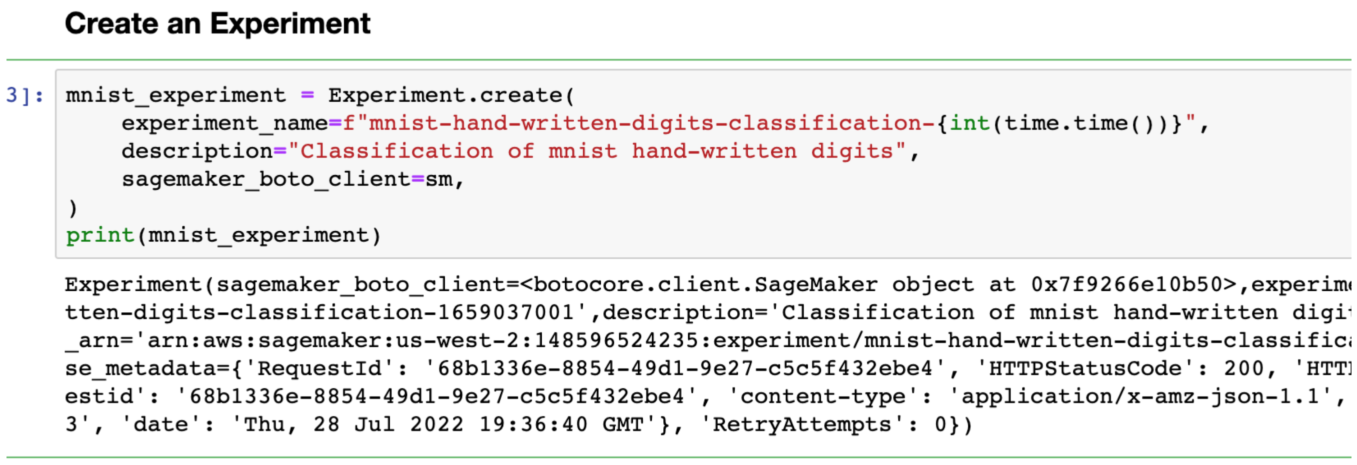
View an example image from the dataset.

```
In [10]: plt.imshow(train_set.data[2].numpy())

Out[10]: <matplotlib.image.AxesImage at 0x7f9267b290d0>
```



After transforming the images in the dataset, we upload it to S3.

Step 4: Set up Track Experiment and Train the model to learn from the data
Now that we have transformed the images in the dataset we can create an experiment to track all the model training iterations. Experiments are a great way for us to organize the data. The screenshot below highlights the experiment's output.



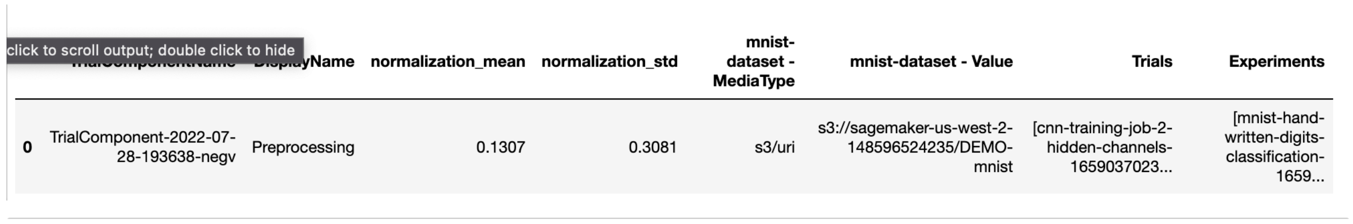
Create an Experiment

```
3]: mnist_experiment = Experiment.create(
        experiment_name=f"mnist-hand-written-digits-classification-{int(time.time())}",
        description="Classification of mnist hand-written digits",
        sagemaker_boto_client=sm,
    )
    print(mnist_experiment)

    Experiment(sagemaker_boto_client=<botocore.client.SageMaker object at 0x7f9266e10b50>,experim
    tten-digits-classification-1659037001',description='Classification of mnist hand-written digi
    _arn='arn:aws:sagemaker:us-west-2:148596524235:experiment/mnist-hand-written-digits-classific
    se_metadata={'RequestId': '68b1336e-8854-49d1-9e27-c5c5f432ebe4', 'HTTPStatusCode': 200, 'HTT
    estid': '68b1336e-8854-49d1-9e27-c5c5f432ebe4', 'content-type': 'application/x-amz-json-1.1',
    3', 'date': 'Thu, 28 Jul 2022 19:36:40 GMT'}, 'RetryAttempts': 0})
```

Step 4 Continued

Compare the model training runs for an experiment

Now we use the analytics capabilities of the Experiments SDK to query and compare the training runs for identifying the best model produced by our experiment. You can retrieve trial components by using a search expression. Below is a screenshot of our query.

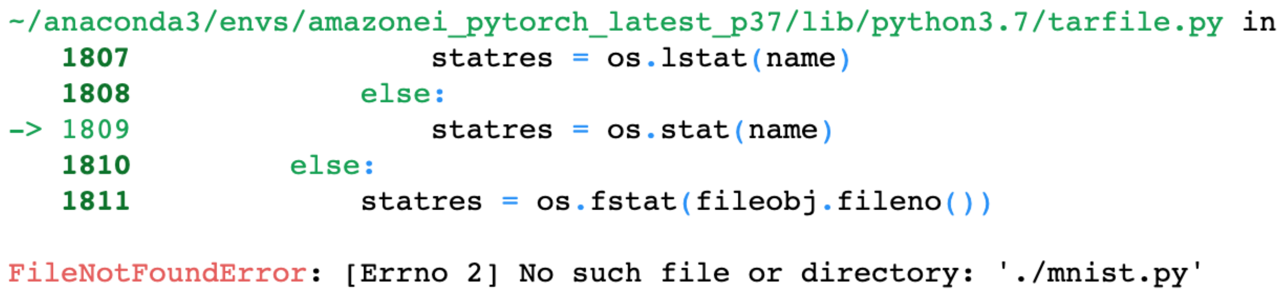


click to scroll output; double click to hide

	TrialComponentName	DisplayName	normalization_mean	normalization_std	mnist-dataset - MediaType	mnist-dataset - Value	Trials	Experiments
0	TrialComponent-2022-07-28-193638-negv	Preprocessing	0.1307	0.3081	s3/uri	s3://sagemaker-us-west-2-148596524235/DEMO-mnist	[cnn-training-job-2-hidden-channels-1659037023...	[mnist-hand-written-digits-classification-1659...

Step 4 and 5 Deployment Issues

I received the FileNotFoundError when I attempted to run the Track Experiment. I believe it is giving me an error because of the `estimator()`. Within the `estimator()` the key-value pair `"TrialComponentDisplayName": "Training"`, is not found. This is also throwing out the Deployment of the Model.



```
~/anaconda3/envs/amazonei_pytorch_latest_p37/lib/python3.7/tarfile.py in
   1807                     statres = os.lstat(name)
   1808                 else:
-> 1809                     statres = os.stat(name)
   1810             else:
   1811                 statres = os.fstat(fileobj.fileno())

FileNotFoundError: [Errno 2] No such file or directory: './mnist.py'
```

```
best_trial_component_name =
trial_component_analytics.dataframe().iloc[0]["TrialComponentName"]
```

The “TrialComponantName” is giving us an error. After multiple attempts, I decided to move on.

Conclusion

To overview, in this section we used AWS Sagemaker and Pytorch to load and store data using a Notebook Instance. In this tutorial, we were able to get the data from **MNIST Handwritten Digits Classification** to return as an image in our notebook. We were also able to configure an experiment test and organize and query our table with a Trail Component Table. However, due to the unresolved bug, I was unable to deploy the ML model successfully. Again for full step by step on How I completed this Section please visit here