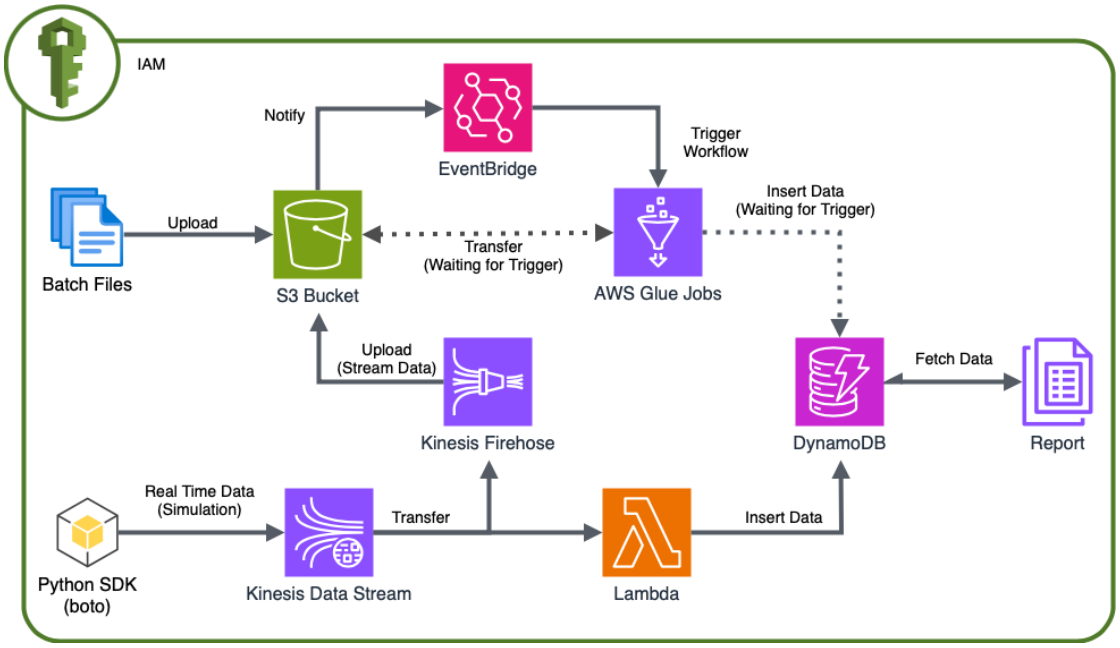
By: Jonathan De La Cruz

Data Practice Documentation

Overview



This project is a system made with AWS that receives 2 types of data, files uploaded representing batch data and real time data, in this case simulated in python. The GitHub repository is available with all files needed.

The batch data gets stored in an S3 bucket and when this gets uploaded it sends a notification to Event Bridge which triggers a workflow in AWS Glue, in here happens multiple operations of transformation, once the data has been processed it is stored in the S3 bucket and is also inserted in the DynamoDB database. In the other hand, the data is simulated with the Python SDK (boto3) to represent streaming data, this is sent to a Kinesis Data Stream, then the data is sent to Kinesis Firehose which allows for the data to be stored in the S3 Bucket, at the same time the data from Kinesis Data Stream is also send to a Lambda function that aggregates the data and sends it to DynamoDB.

These are 2 different types of data that end up in their respective tables in DynamoDB for then to be fetched and displayed on a report using Streamlit and Python.

This document will give instructions on how to recreate this system, as well as explaining some of the process of developing this.

Security

Before everything, we will set up the users and roles we will need for everything to make it easy, we will have just one role with all policies which isn't recommended for security reasons, the better way would be to create specific roles with different policies for each tool, but if this is not a concern we can use a general role for the system, either way you could separate this policies in different roles according to your needs.

First in the AWS Management Console we search for IAM, here we create a policy, at the top of the options we can insert Json code, that is where we insert the code from "main_policy.json", in this code we give permission for the required actions to some services. Something to keep in mind is not to leave the resources as full access, as we create the services needed it is better to pass the arn available for these services, these are different for each user, that is why I did not include it in the resource.

Allow (3 of 440 services)

Service	Access level	Resource
DynamoDB	Limited: Read, Write	All resources
Kinesis	Limited: List, Read, Write	All resources
S3	Limited: Read, Write	All resources

Once we created the policy, we will create a role with a custom trust policy with the next code, found in "trust_policy.json":

This means that the role will be available for our Glue and Lambda services. After that we select the policy created and we also search for and select the CloudWatchFullAccess policy managed by AWS and we include it in the policies for this role, this is for the lambda function, for it to be able to send logs to the CloudWatch system and us to be able to see them if there is any problem.

After the role has been created, we go ahead and create a user, here we attach the custom policy we created earlier, we will get back to it in the streaming data section, but this is what will give us access to our resources from outside using an SDK.

Batch Files

For this project we will use sample data from the Social Media Engagement Report dataset used by Ali Reda in Kaggle. This is a big file with 100,000 rows representing posts, but for this project we will only be using 100 rows maximum. Why? Working with just 100 rows separated in 4 files, means that we will lower the cost as much as possible, this doesn't mean the system will not work with the entire file, since everything in AWS is able to scale up, and keeping a low number of rows allows for the stream data to target the same posts (an interesting scenario) without having to increase the streamed data (also keeping the costs to the lowest).

The columns are the following, but we will keep it simple and remove most of them, but this will be done in AWS Glue:

Column Non-Null Count Dtype
0 Platform 100000 non-null object

1 Post ID	100000 non-null object
2 Post Type	100000 non-null object
3 Post Content	100000 non-null object
4 Post Timestamp	100000 non-null datetime64[ns]
5 Likes	100000 non-null int64
6 Comments	100000 non-null int64
7 Shares	100000 non-null int64
8 Impressions	100000 non-null int64
9 Reach	100000 non-null int64
10 Engagement Rate	100000 non-null float64
11 Audience Age	100000 non-null int64
12 Audience Gender	100000 non-null object
13 Audience Location	100000 non-null object
14 Audience Interests	100000 non-null object
15 Campaign ID	20132 non-null object
16 Sentiment	49900 non-null object
17 Influencer ID	9994 non-null object

You can either grab the sample data from the GitHub directory or use the code from "data_divider.py" to separate it in a notebook (like Colab) or locally by just running the python file.

S3 Bucket (Storage)

We search now for the S3 service, here we create a bucket with the default settings, inside of it we should create the folders 'processedData/posts_processed/' and 'rawData/posts/', 'rawData' and 'processedData' folders being at the same level. 'rawData/posts/' is the folder where we will insert our batch files (it is recommended to upload one file of the batch files for the AWS Glue Crawler we will set up later).

Finally, select the bucket once again and go to its properties, inside properties we will look for Amazon EventBridge, we will enable the "Send notifications to Amazon EventBridge for all events in this bucket", this will allow EventBridge to be notified when we upload a file later on. In the properties tab of the bucket the arn is available, so it will be a good idea to change the policy we created before, remove the "*" from resources and just pass the arn in the statement with the S3 operations.

DynamoDB

We will need 2 tables in DynamoDB, the first table preferably should have the name ‘ProcessedData’ with the Partition Key set to ‘post_id’. The second table should have the name ‘ProcessedStream’ with the Partition Key set to ‘post_id’ and the Sort Key set to ‘window_start’.

Additionally, when creating the tables you can go to ‘Table settings’ and choose ‘Customize settings’ and set the ‘Read/write capacity settings’ to ‘Provisioned’, set off autoscaling and then chose the lowest capacity units, this will guarantee that you have the lowest cost for the table, but will not be able to scale.

We will use these tables later on, but they have to be running beforehand.

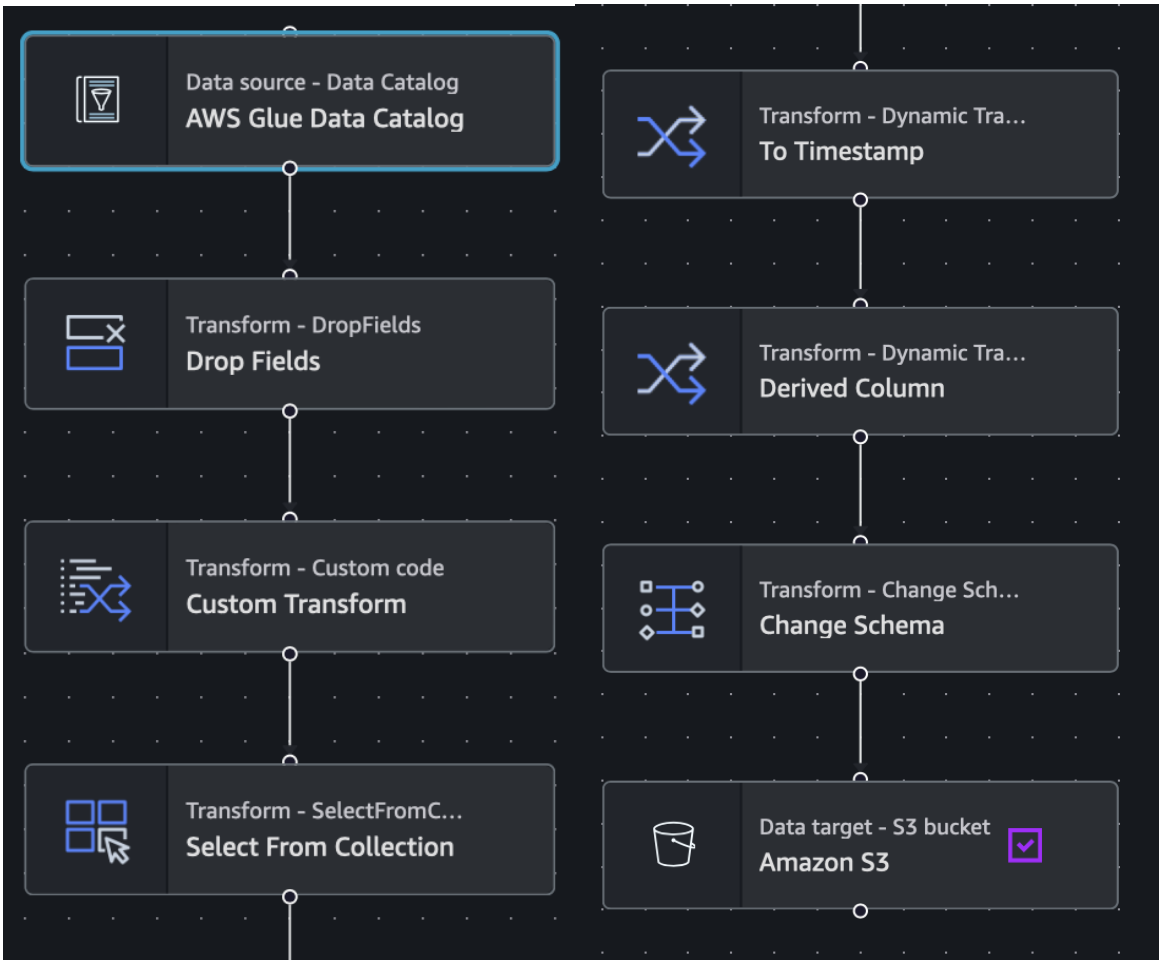
AWS Glue

First, we create 2 databases under the Data Catalog, one called ‘raw_data’ and the other ‘processed_data’. Now we will work in the AWS Glue service, we will select “Crawlers” here we create one with the data source being our folder for raw data in our bucket, for example “s3://your-bucket/rawData/posts/”, then we select our previously created role, we select the ‘raw_data’ database.

So, what did we just do? A crawler is a program that will go through our data trying to infer the schema of our data and then creating a table in the database with this schema. This database is a Data Catalog database, this is for AWS Glue so we can work with the data in our ETL jobs directly from the S3 bucket in a table format with a defined schema thanks to the crawler.

Now we go to the ETL jobs section, select the Visual Editor, now in the Actions dropdown button, we select “Upload” and we upload the file “process_batch_data.json”, now we will have 8 step transformation to the data.

Note: In the “Job details” tab is important to set “Job bookmark” to enabled, this will allow the job to exclude previously processed data. You may as well lower the “requested number of workers” to lower the cost.



1. AWS Glue Data Catalog: It selects the data from our, Data Catalog. Make sure to select the correct location for your database and table.
2. Drop Fields: It drops multiple fields at once from the schema of the table from the Data Catalog, we just keep the essentials: ‘post id’, ‘post type’, ‘post timestamp’, ‘likes’, ‘comments’, ‘shares’, ‘reach’.
3. Custom Transform: This is a custom function made to just rearrange the columns, it returns a DynamicFrameCollection, with just one table inside of it.
4. Select From Collection. Since the past function forcefully returns a collection, now we have to select the data in it.
5. To Timestamp: We convert the ‘post timestamp’ which is currently a string to a timestamp type.

6. Derived Column: Here we can create a new column called ‘engagement rate’, based on the description given in the dataset in Kaggle, this is calculated by (likes+comments+shares) / reach
7. Change Schema: Here we just replace the ‘space’ in all the column names to an underscore ‘_’, so ‘post id’ will be ‘post_id’.
8. Amazon S3: This is where we will send the processed data. Make sure that the location is correctly pointing to the ‘posts_processed’ folder, for example, ‘s3://your-bucket/processedData/posts_processed/’.

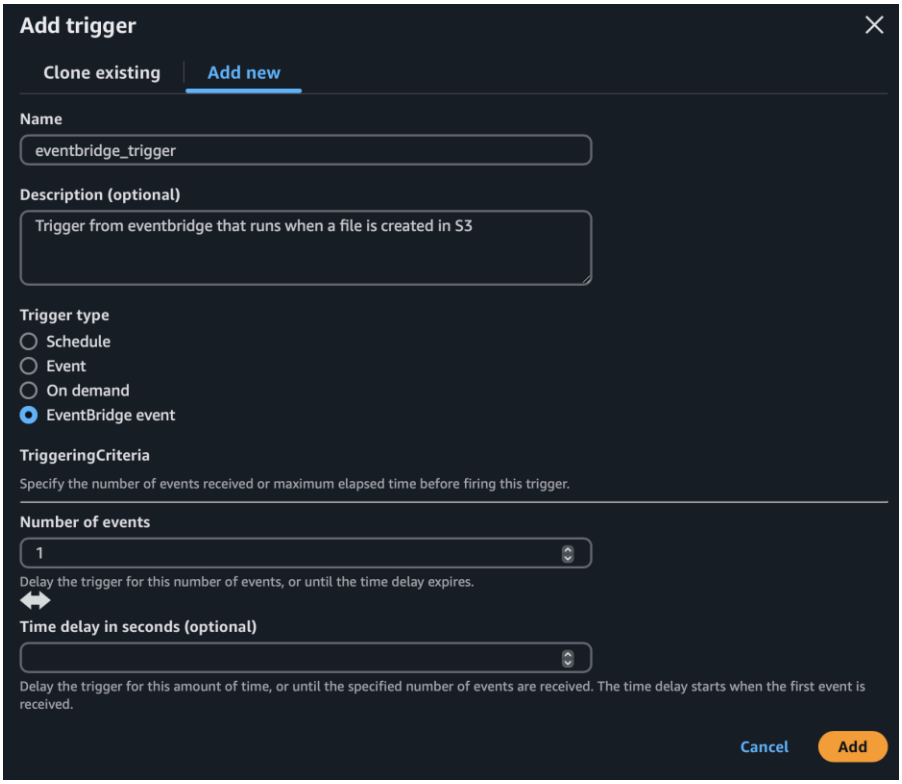
This is the new schema of the data:

Column name	Data type
post_id#0	string
platform	string
post_type#1	string
post_timestamp#2	timestamp
likes	bigint
comments	bigint
shares	bigint
reach	bigint
engagement_rate#3	double

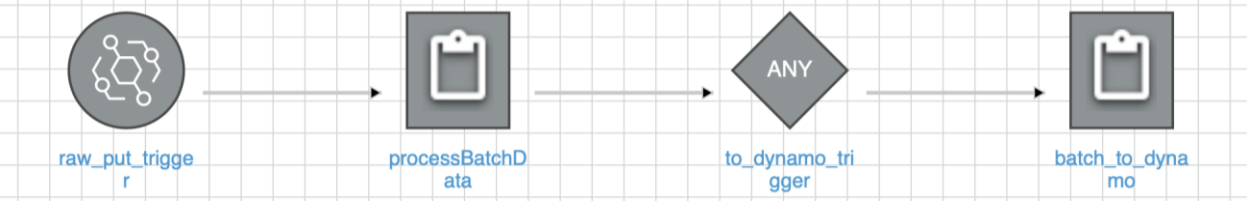
After saving it, we have just created the first ETL Job that gets the raw data inserted, transform it and stores it in another folder in our S3 Bucket. This time we will create an ETL Job that gets the data from the processed folder and inserts it to DynamoDB.

The process is the same, open ETL Jobs, choose Visual ETL or any other option, click on Actions and choose Upload, now the file is ‘batch_to_dynamo.json’. The visual editor will be removed this had to be done this way, because DynamoDB is not available as a destination in the visual editor. The file just read the parquet file in the S3 path provided (make sure to change it to your correct arn) and then uploads the data to the DynamoDB table ‘ProcessedData’ (this can be change). That’s it there are no transformations, make sure to save the job, but also enable job bookmarking as the previous job.

Now that we have our 2 jobs ready, we'll go to 'Workflows' and create a workflow. Here you will need to create a trigger, when adding a trigger make sure is under the 'Add new' trigger tab, write its name and the important part is to select the EventBridge event as the trigger type, like shown below.



After this, add a node, and add the first job created 'process_batch_data', then by clicking on the job there will appear an option to add a trigger, create a trigger this time of trigger type 'Event' and considering we only have one node before the trigger, the trigger logic can be the default value. Then add the other job after this trigger, now the workflow is completed, it can be run manually but is better after setting up EventBridge. The workflow should look something like this:



EventBridge

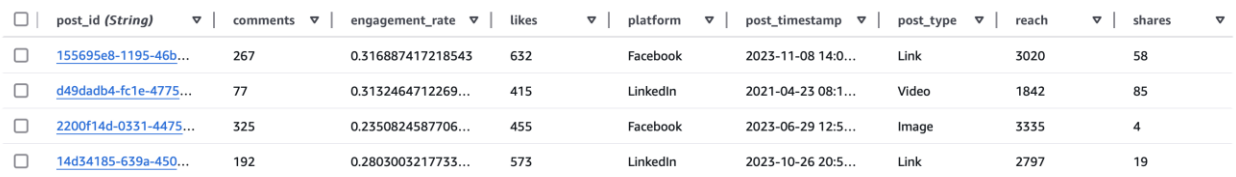
The only thing we are missing for the batch data is the EventBridge, in this service we have to create an EventBridge Rule, here we will create an even rule with the following settings:

- Name: Whatever you would like
- Events:
    - Event source: “Other”.
    - Event pattern – Creation method: “Custom pattern”
        - paste the code from “eventbridge_pattern.json” or you
            - Make sure to change it to the bucket name (not the arn this time), and make sure the prefix is correctly pointing to the rawData’s posts folder.
- Target 1:
    - AWS service: “Glue workflow”
    - Glue workflow name: the workflow we created previously
    - Execution role: “Create a new role for this specific resource”

Finally, we have finished with all the process of the batch data, this EventBridge will be receiving notifications from the S3 Bucket where we previously enabled to send notifications to EventBridge, in the code of “eventbridge_pattern.json” what it does is to only “run” or “trigger” the EventBridge when the specific “Object Created” notification has been given, from the bucket in the specific folder.

Now the workflow works whenever we upload a file to the “rawData/posts/” folder in the S3 bucket. Once the files are uploaded the workflow in AWS Glue will automatically trigger the first job, which is to extract the files to the folder, transform them and load them into the processed folder, when this job is done the next trigger will make the other job run, taking the data from the processed folder into DynamoDB.

The data in the DynamoDB table looks like the image below, the other fields aside from the partition key get placed based on alphabetical order.



post_id (String)	comments	engagement_rate	likes	platform	post_timestamp	post_type	reach	shares
155695e8-1195-46b...	267	0.316887417218543	632	Facebook	2023-11-08 14:0...	Link	3020	58
d49dadb4-fc1e-4775...	77	0.3132464712269...	415	LinkedIn	2021-04-23 08:1...	Video	1842	85
2200f14d-0331-4475...	325	0.2350824587706...	455	Facebook	2023-06-29 12:5...	Image	3335	4
14d34185-639a-450...	192	0.2803003217733...	573	LinkedIn	2023-10-26 20:5...	Link	2797	19

Stream Data

For the stream data we first need to prepare the file that we will use to send data and test our system. The file is called "send_data_to_kinesis.py", there are various ways to run this, see the AWS documentation on Developer Tools, but the project was made using Visual Studio Code, here we must download as an extension "AWS Toolkit". We will need an access key and a secret key.

We go back to the IAM Service, select the user we have already created, then on the summary or under the "Security credentials" tab, you may find where you can create Access keys, create one, and make sure to save both keys.

Now we can sign into Visual Studio Code through the AWS Toolkit, you also need to place those keys in the code of "send_data_to_kinesis.py" when initializing the client of boto3 at the start, if security is a concern try to manage the keys with environment variables. Once that is made, we will need to create a folder called "data" and upload the batch files we have already uploaded to the s3 bucket (this is so the randomized id for the stream data is part of the processed batch files so we can join the data properly), meaning if the files 1 and 2 have already been processed in the batch section, we can place those in the folder 'data'. This folder has to be at the same level of the python file, or the path can be changed. Now we have to run the file in a python environment, the libraries that we have to install before running are boto3 and pandas, after this we are ready to run the file, but we are still missing the kinesis stream is pointing to.

The purpose of this program is to simulate real time data of social media engagements, we are selecting a random id from the data that has already been processed and randomizes the type of engagement. It also randomizes a user id, but since we do not have data for users this is just to make it similar to real data. Then the record is sent to Kinesis, which we will talk about in a moment.

```
while True:
    randomIndex = random.randrange(df.shape[0])
    record = {
        'post_id': df.iloc[randomIndex,1],
        'engagement_type': random.choice(types),
        'user_id': f'{random.randrange(10000):04}',
        "timestamp": datetime.now(timezone.utc).isoformat()
    }

    response = clientkinesis.put_record(
        StreamName=kdsname,
        Data=json.dumps(record),
        PartitionKey=str(record['post_id'])
    )
```

The program has an infinite loop so is important to shut it down in the terminal (Control + C), it also has a waiting time of 1 second, meaning we are sending 1 record per second, this can be removed or modified.

Kinesis Data Stream

Now we go to the Kinesis service, we create a Kinesis Data Stream with the name of “input_stream”, you can also set the capacity to “provisioned” and select the lowest to lower the cost.

Once this is created, we can run the program, and data will start flowing through the Kinesis Data Stream, but we are not doing anything with the data.

Kinesis Firehose Stream

In the Kinesis service we will create a Firehose Stream, this is a perfect tool to send data from our Kinesis Data Stream to our S3 storage. So we will create a Firehose Stream with the following settings:

- Source: “Amazon Kinesis Data Streams”
- Destination: “Amazon S3”
- Source settings: select the Kinesis stream we created previously
- Destination settings: select the S3 bucket used in the project
    - Bucket prefixes: Available in the “s3_prefixes.txt” file as well.
        - Prefix: *rawData/stream/year=!{timestamp:YYYY}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/*
        - Error Prefix: *rawData/stream/!{firehose:error-output-type}/year=!{timestamp:YYYY}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/*

Now the Firehose Stream should be ready, when the “send_data_to_kinesis.py” script is running it will send data to the Kinesis Data Stream, from there Firehose Stream will store the data in the S3 bucket based on the year, month, day and hour records were captured.

Lambda

Now in the Lambda service we will create a function that will aggregate the data based on tumbling windows and send that data to the database.

So, we create a Lambda function from scratch, and we set the runtime to Python 3. Then we will copy the code from the file “lambda_function.py”. Here we gather all the records and place them in a dictionary based on the id and engagement type (meaning the type of engagement the post received on the streamed data record). There we accumulate everything and count all the types of engagements per post.

The most important part of the code is here, for each post and engagement type we will update an item in the database, if the item doesn’t exist, the ‘update_item’ will create an

item for us. First we pass the keys, then we use an expression to update the attribute, if it exists we add 1 to that value and if not, we added to 0, and we also set the window_end attribute of the data. In the 'ExpressionAttributeNames' we add the name of the column we want, in this case is whatever type of engagement we were updating and in 'ExpressionAttributeValues' we set values as if they were variables.

```
# Update DynamoDB
for (post_id, engagement_type), count in engagement_counts.items():
    partition_key = post_id
    sort_key = window_start

    engagement_attr_name = f'engagement_counts.{engagement_type}'

    table.update_item(
        Key={'post_id': partition_key, 'window_start': sort_key},
        UpdateExpression=f"SET #attr = if_not_exists(#attr, :start) + :count, window_end = :window_end",
        ExpressionAttributeNames={'#attr': engagement_attr_name},
        ExpressionAttributeValues={':count': count, ':start': 0, ':window_end': window_end}
    )
```

In the Function overview diagram of the Kinesis Lambda we will add a trigger of Kinesis type, select the Kinesis stream, enable the option to activate trigger, considering we are just sending one record per second we can lower the batch size (or increase the amount of data stream we send) and finally we also set the tumbling window to some value, it can be set to 30, meaning that the window will work with the records of the past 30 seconds.

Finally, we have completed the stream data process, when uploading data to the Kinesis Data Stream, the data will go through the lambda function and insert objects into the 'ProcessedStream' table, the data inserted looks like the image below, where every record is the amount of engagement per type by each window, the empty fields mean 0.

post_id *(String)*	window_start *(String)*	engage...	engage...	engage...	window_end
d49dadb4-fc1e-4775...	2025-03-10T08:00:30Z	1			2025-03-10T08:01:00Z
d49dadb4-fc1e-4775...	2025-03-10T08:01:00Z		1		2025-03-10T08:01:30Z
d49dadb4-fc1e-4775...	2025-03-10T08:01:30Z		1		2025-03-10T08:02:00Z
d49dadb4-fc1e-4775...	2025-03-10T08:03:00Z	1			2025-03-10T08:03:30Z

Report

There are various ways to generate a report, the one chosen was streamlit by fetching the data from the tables in the DynamoDB database. We will not go into details as in what does the code for the report do, because the main part of the project was how the system will work in AWS.

To run the streamlit file "report.py" we must be in a python environment, we must install the streamlit library, we must pass the access key with the secret key to the file and we have to run 'streamlit run report.py'. In this report a lot of operations were done, we fetch the data and place them into pandas' dataframes.

First it displays charts of the batch data using streamlit functions to show some information about the batch data. Then in the stream data we display more information this time not using streamlit functions only (because it was restrictive), but using altair library which is more flexible as well, in these charts we display a lot of information made by multiple aggregation operations and joins between the batch data and the stream data, and it also allows the user to select how some data should be displayed.

If you would like to know how the report looks like, make sure to look at the file 'report.pdf'.