CSCI 5409: Assignment 3: Part B:

Waleed R. Alhindi (B00919848)

GitLab A3 Part B Repository:

The A3-Part B code has been pushed to the GitLab repository's "A3" branch under the "Part_B" folder, which can be found at the following URL:
https://git.cs.dal.ca/alhindi/csci5410-summer-23-b00919848/-/tree/A3/Part_B

OR

Under the "A3" folder's "Part_B" sub-folder in the "main" branch, which can be found here:
https://git.cs.dal.ca/alhindi/csci5410-summer-23-b00919848/-/tree/main/A3/Part_B

Additionally, the professor and all TAs have been granted "Maintainer" access to this GitLab repository. This includes assigning "Maintainer" roles to the following GitLab accounts:

- *@saurabh (Dr. Saurabh Dey)*
- *@mudgal (Ankush Mudgal)*
- *@bharatwaaj (Bharatwaaj Shankanarayanan)*
- *@rmacwan (Rahul Ashokkumar Macwan)*

Operations Performed:

1. Creating S3 Buckets:

Our first step is to create the two AWS S3 [1] buckets: a "sampledata-b00919848" bucket, where the files from the "Tech" folder will be stored, and "tags-b00919848", where the extracted feature JSON of each file will be stored.

Note that although the S3 Buckets will be created manually, the files from the Tech folder will be uploaded to the "sampledata-b00919848" bucket using a script.

Thus, we can go ahead and create the "sampledata-b00919848" bucket as seen in Figure 1:

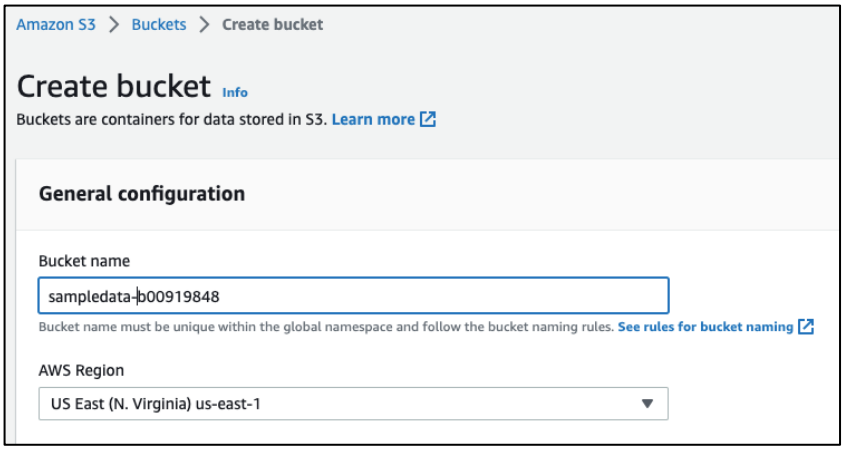

Fig 1. Creating "sampledata-b00919848" S3 Bucket [1]

Similarly, we create the "tags-b00919848" Bucket as seen in Figure 2:

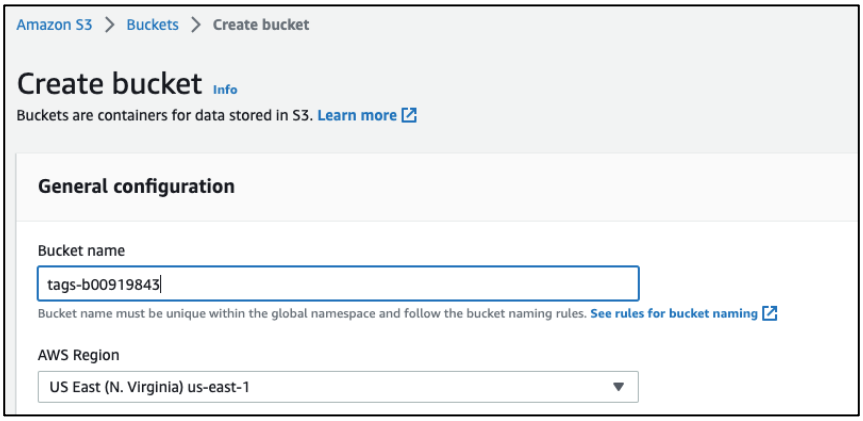

Fig 2. Creating "tags-b00919848" S3 Bucket [1]

As seen in Figure 3, the two S3 Buckets have been successfully created:

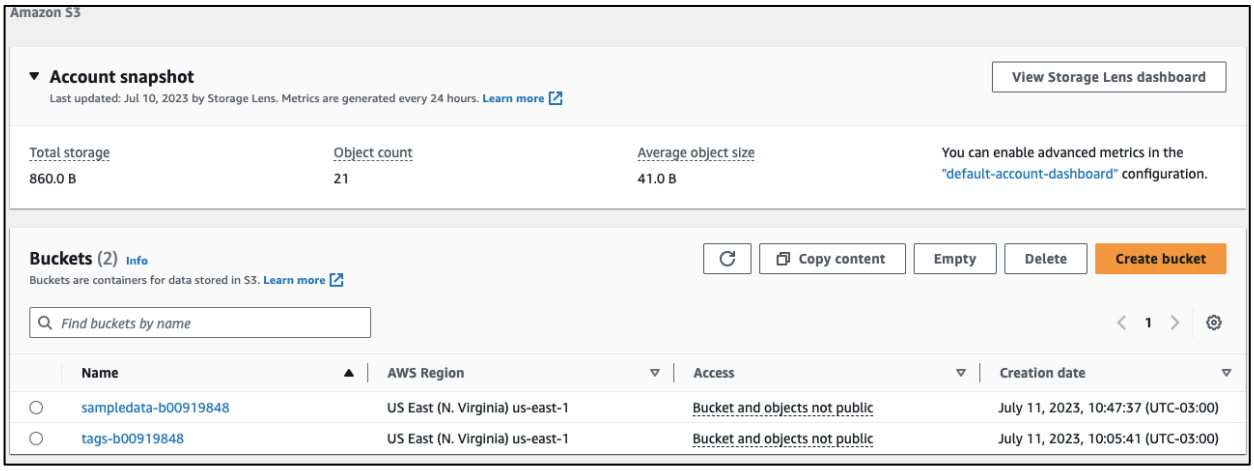

Fig 3. S3 Buckets Successfully Created [1]

2. Creating DynamoDB Table:

In addition to the two S3 Buckets [1] created, we need to create a DynamoDB [2] table to store each extracted feature (named entity) and its number of occurrences. Thus, we can go ahead and create the "namedEntities" DyanmoDB [2] table as seen in Figure 4:

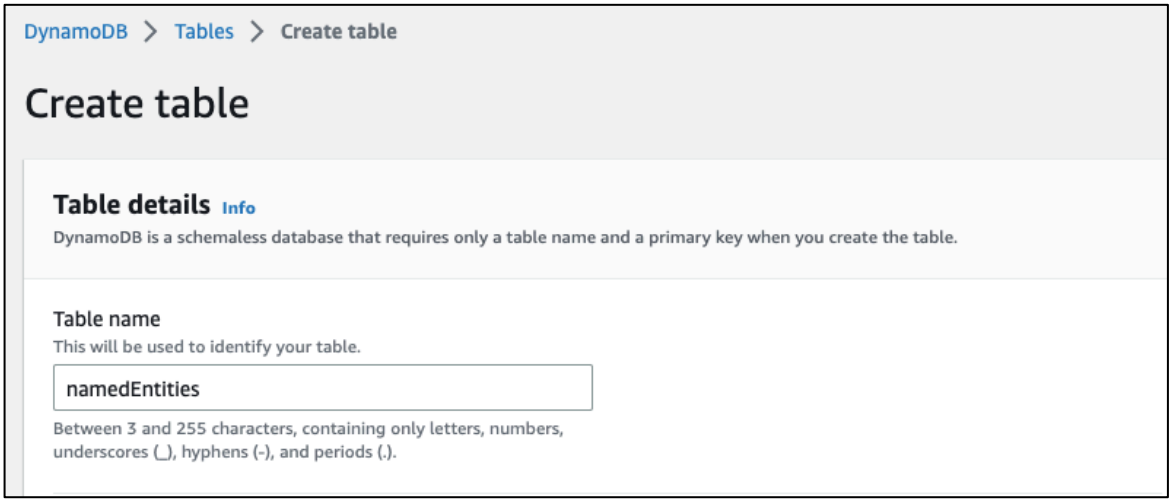

Fig 4. Creating "namedEntities" DynamoDB Table [2]

However, as seen in Figure 5, the "namedEntities" table is created with a partition key called "entity", which refers to the named entity being stored. Therefore, each document will contain two fields: an "entity" partition key field that denotes the actual word being stored and a "tally" value that denotes how many times it has occurred. For example, the JSON {"USA": 2, "Canada": 1} will be stored as two documents: {"entity": "USA", "tally": 2} and {"entity": "Canada", "tally": 1}. Additionally, these documents will be updated as newer (applicable) JSONs are stored in the "tags-b00919848" S3 Bucket [1].

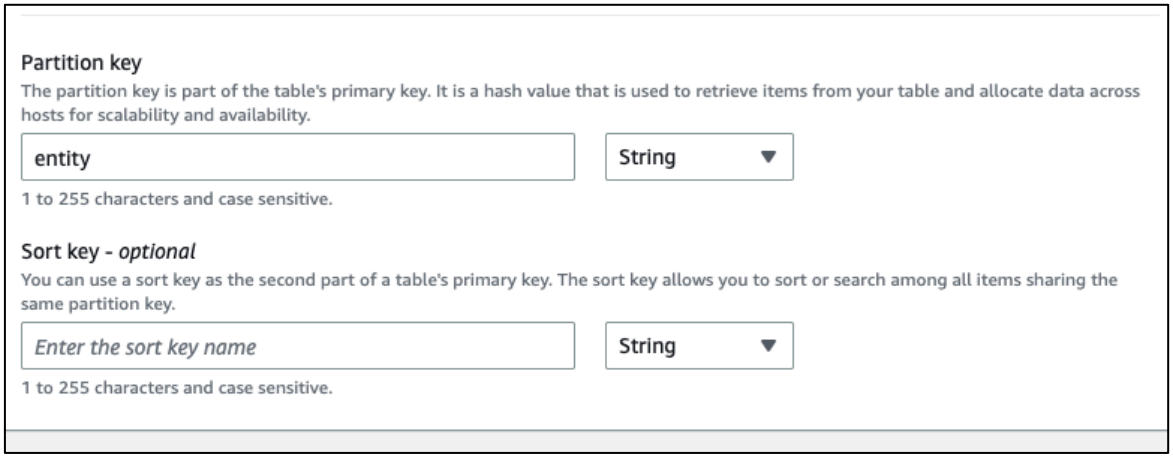

Fig 5. Defining "entity" Parititon Key in "namedEntities" Table [2]

Finally, as seen in Figure 6, the "namedEntities" DyanmoDB table [2] has been created successfully:

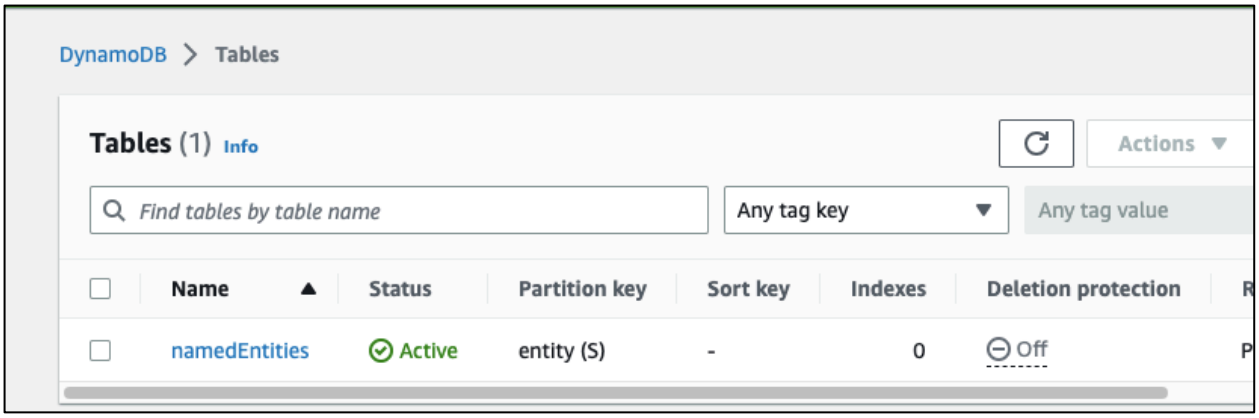

Fig 6. "namedEntities" DyanmoDB Table Successfully Created [2]

3. Composing extractFeatures Lambda:

Now that our S3 Buckets [1] and DynamoDB table [2] are set up, we can begin implementing the "extractFeatures" AWS Lambda [3] function. This function is meant to be triggered any time an object is added to our "sampledata-b00919848" S3 Bucket [1], and will extract all named entities in the newly added object (file from Tech folder) into a JSON string that it will then store in our "tags-b00919848" S3 Bucket [1].

Let us first create the "extractFeatures" function as seen below:

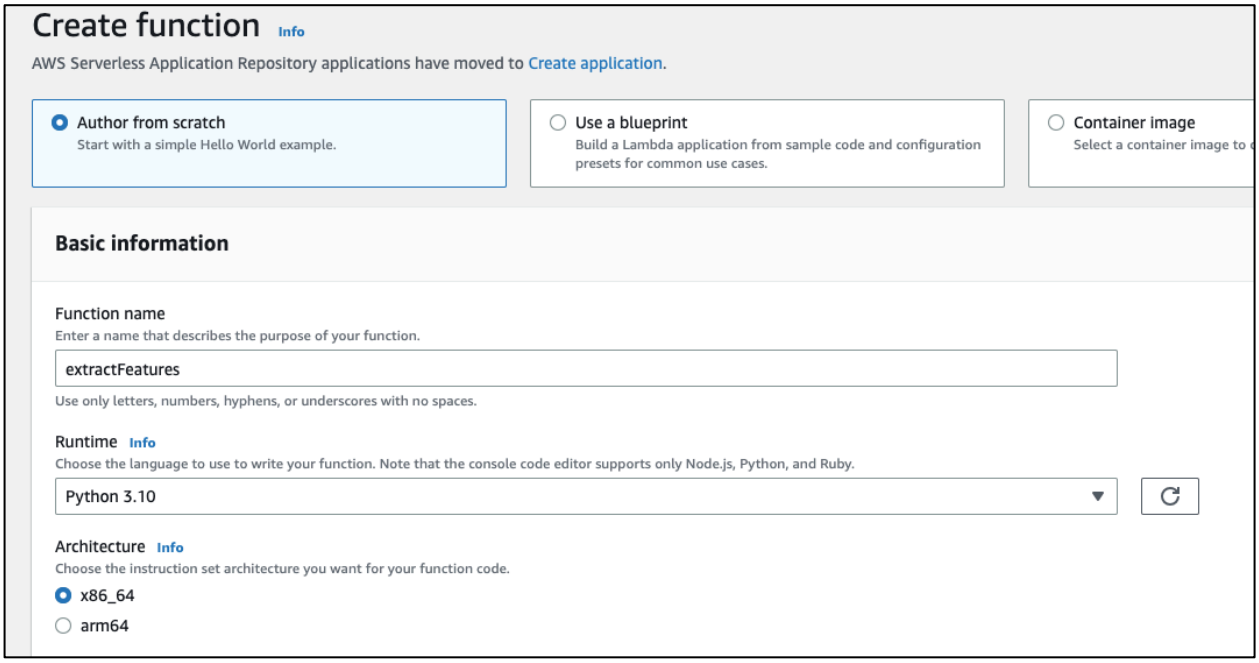

Fig 7. Creating "extractFeatures" Lambda Function [3]

After creating the function, we need to configure a trigger on to the "extractFeatures" that will be invoked whenever an object is pushed to our "sampledata-b00919848" S3 Bucket [1]:

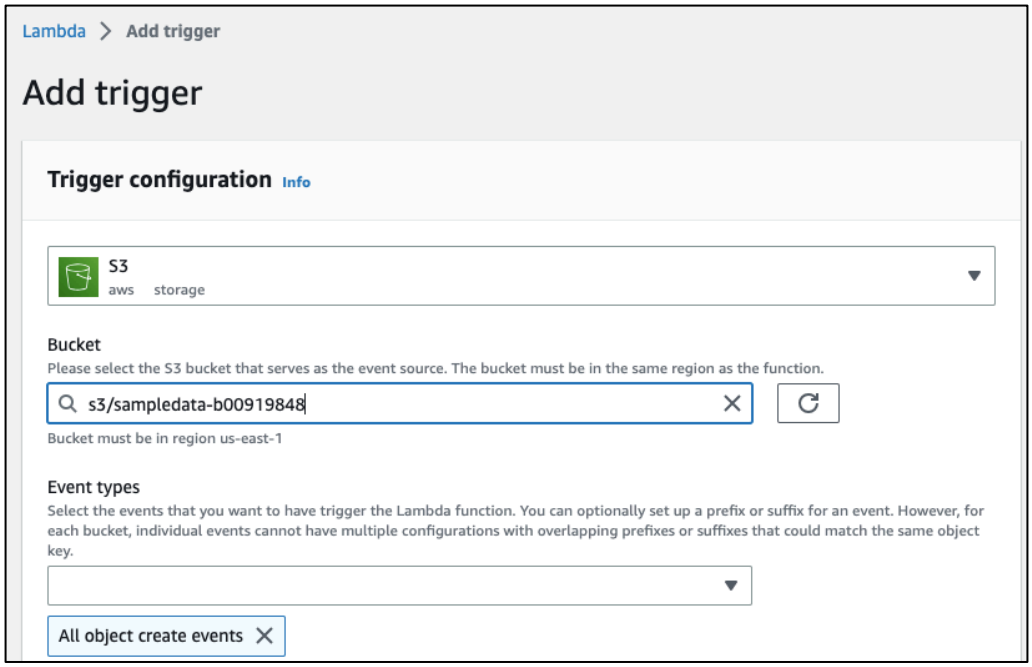

Fig 8. Adding Trigger to "extractFeatures" Lambda [3]

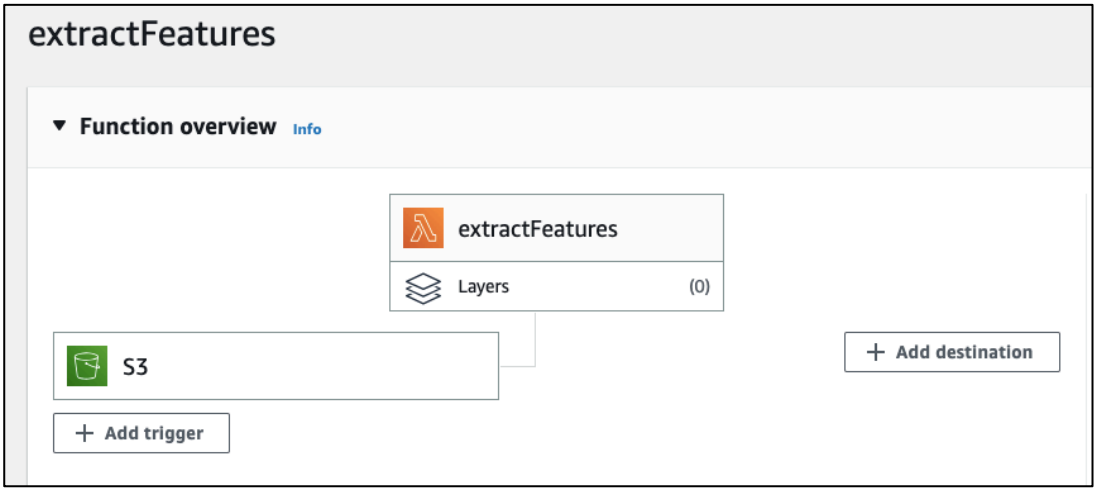

Fig 9. Trigger Successfully Added to "extractFeatures" Function [3]

Next, the function's source code was implemented as seen in Figure 10:

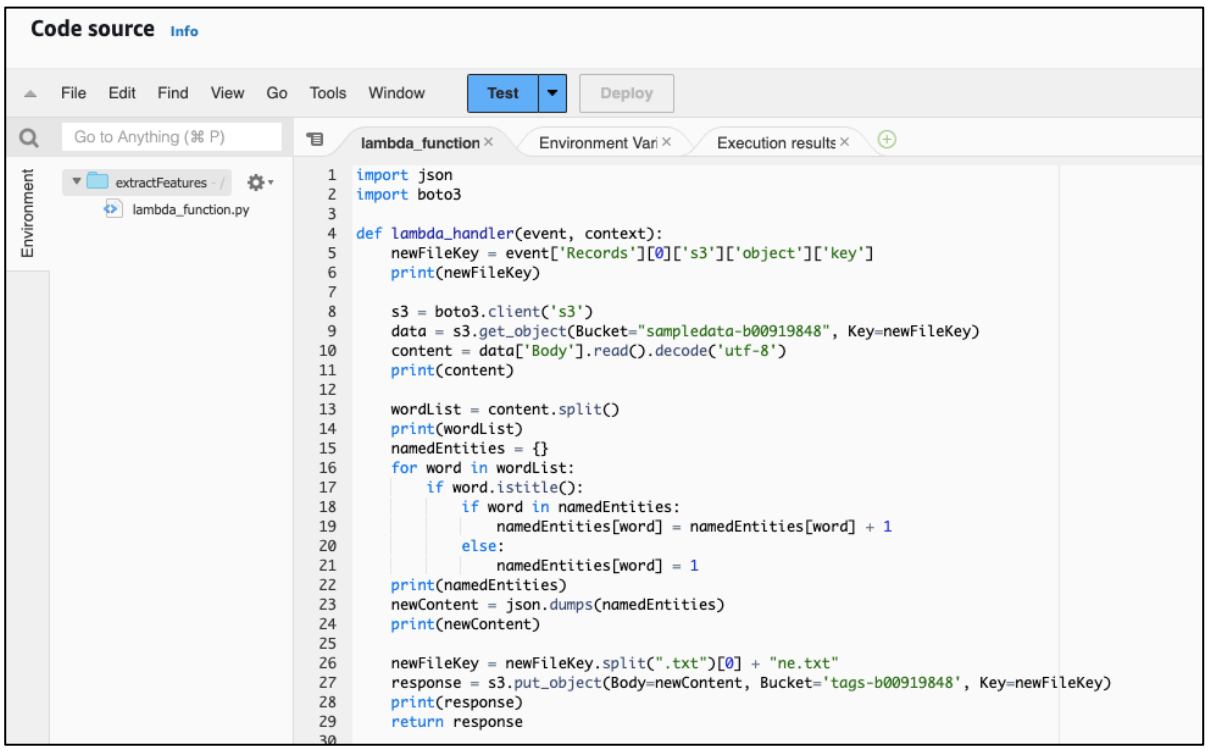

Fig 10. "extractFeature" Function's Source Code [3]

The code depicted in Figure 10 is triggered whenever an object is pushed to the "sampledata-b00919848" bucket; supplying the function with an "event" that contains information about the object that was just pushed. It then retrieves the newly pushed object, scans through its data, and added each named entity (i.e., each word that begins with a capital letter) into a JSON. This JSON is then converted into an encodable string and stored in our "tags-b00919848" bucket.

Additionally, the pseudocode of the code depicted in Figure 10 is outlined below:

Get new object's key from "event"

Create empty JSON

Retrieve object using key

Split object's string data into an array of words

Iterate through each word in that array

    If that word's first letter is capital

        If that word exists in our JSON

            Increment its value by 1

        Else

            Add {word: 1} to the JSON

Convert JSON into string (dumps)

Upload JSON string to "tags-b00919848" bucket

4. Composing accessDB Lambda:

The “accessDB” Lambda [3] function is meant to be triggered once the “extractFeatures” Lambda [3] has uploaded an object to the “tags-b00919848” bucket. Once triggered, it will retrieve the named entities from the newly uploaded file and update the “namedEntities” DynamoDB [2] table accordingly. In other words, when a JSON string file is uploaded to the “tags-b00919848” bucket, this function will retrieve that file and use the named entities stored in it to update the “namedEntities” table accordingly.

Implementing the “accessDB” Lambda [3] function is very similar to our “extractFeatures” function above. We first create the “accessDB” function, then create a trigger for it. However, as seen in Figure 11, this time our trigger will be for the “tags-b00919848” bucket:

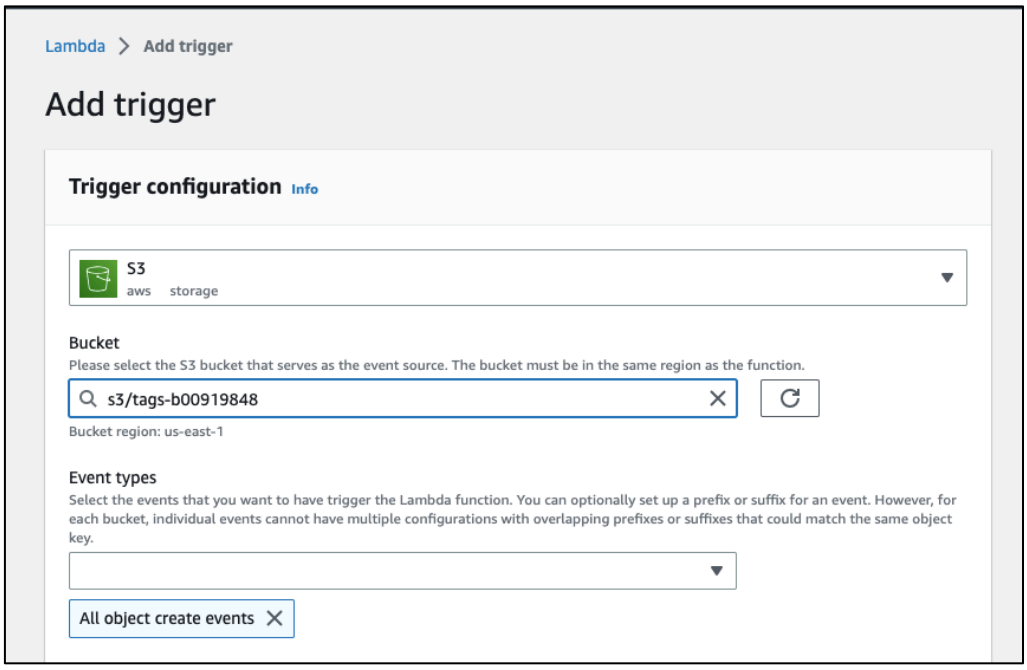

Fig 11. Adding Trigger to “accessDB” Function [3]

Fig 12. Trigger Successfully Added to “accessDB” Function [3]

After which, we can implement the function’s source code, which can be seen in Figure 13 below:

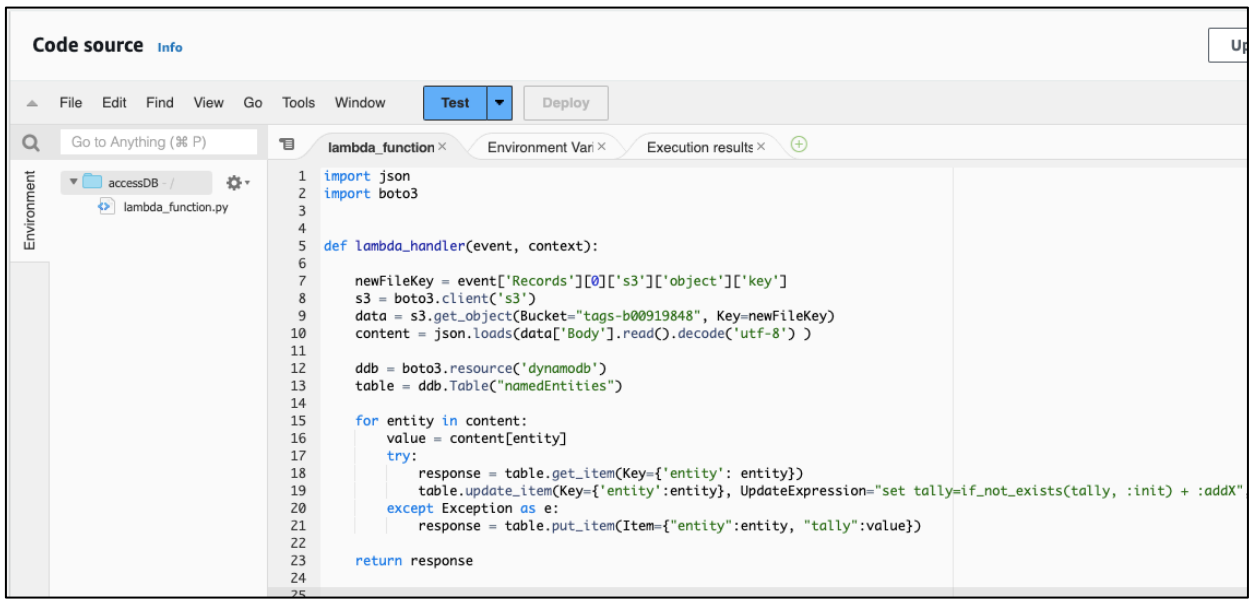

Fig 13. "accessDB" Function's Source Code [3]

The code depicted above is invoked whenever a new object is added to the target S3 bucket and is passed information about the newly added object. Using that information, the object is retrieved. Then, the JSON string is converted back into a JSON object (loads). Then, for each key in the JSON, it either adds that named entity to our DynamoDB [3] table if it does not already exist or increments that entity's tally in the table if it does exist.

The pseudocode of the code depict in Figure 13 is outlined below:

Get new object's key from "event"

Retrieve that object using that key

Load object's data into a JSON

Iterate through each key (named entity) in the JSON

If the key already exists in the DyanmoDB table

Increment its corresponding tally based on the key's value

Else

Create new item in the table to represent this key and set its tally to the key's value

5. Implementing Script to Upload Files from Tech Folder to S3:

The last piece of our event-driven puzzle is the script that will upload each of the 401 files in the provided Tech folder to the "sampledata-b00919848" bucket with a 100 millisecond delay between each upload.

Figure 14 showcases the simple Python script used to upload each file in the Tech folder:

Fig 14. Python Script to Upload Tech Folder's Files to "sampledata-b00919848" Bucket

The code depicted above is a simple Python script that uses my locally stored AWS credentials (i.e., the ones stored in ~/.aws/credentials) and the boto3 library to upload each file in the provided Tech folder to our "sampledata-b00919848" bucket. Note, however, that after each upload, a 100 millisecond delay is awaited.

6. Testing and Verification:

Now that all we have configured our S3 Buckets, DynamoDB table, Lambda functions, and file upload script, all that is left to do is to verify and test its behavior. Note that running the file upload script should initiate the entire trigger pipeline. In other words, we shall execute the file upload script, then verify whether files are being correctly extracted and stored into the S3 buckets and DynamoDB table.

As such, we go ahead and *locally* execute the script as seen in Figure 15:

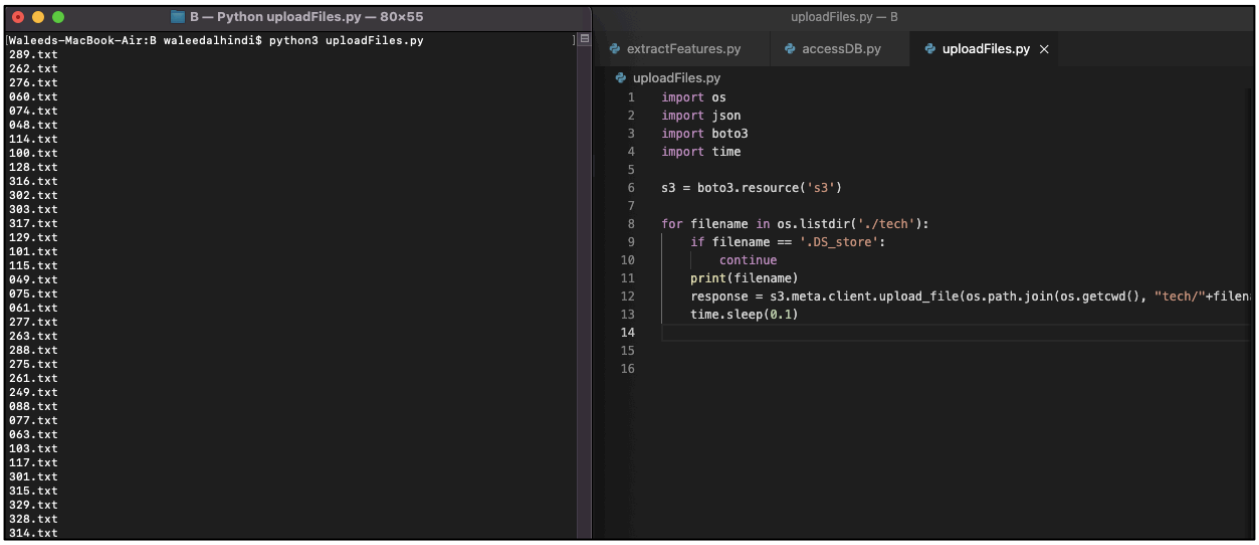

Fig 15. Begin Executing UploadFiles.py Script

The script takes some time to execute due to the number of files, but once complete we can see that it has finished executing without any exceptions being raised; meaning the files should have been successfully uploaded.

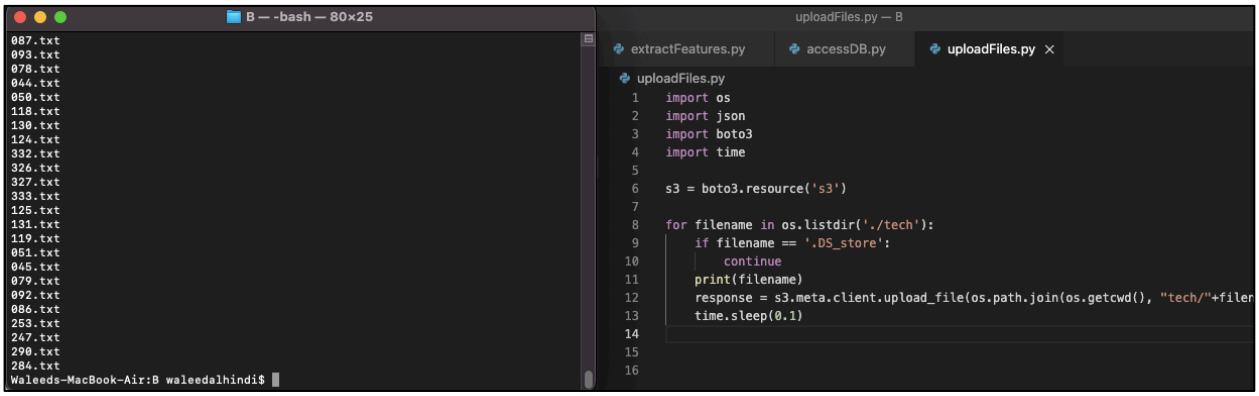

Fig 16. UploadFiles.py Script's Execution Completed

While Figures 15 and 16 suggest that all files have been uploaded to the bucket, we still need to verify that the files were indeed uploaded by navigating to our "sampledata-b00919848" bucket. As expected, each of the 401 files were uploaded to the bucket as seen in Figure 17:

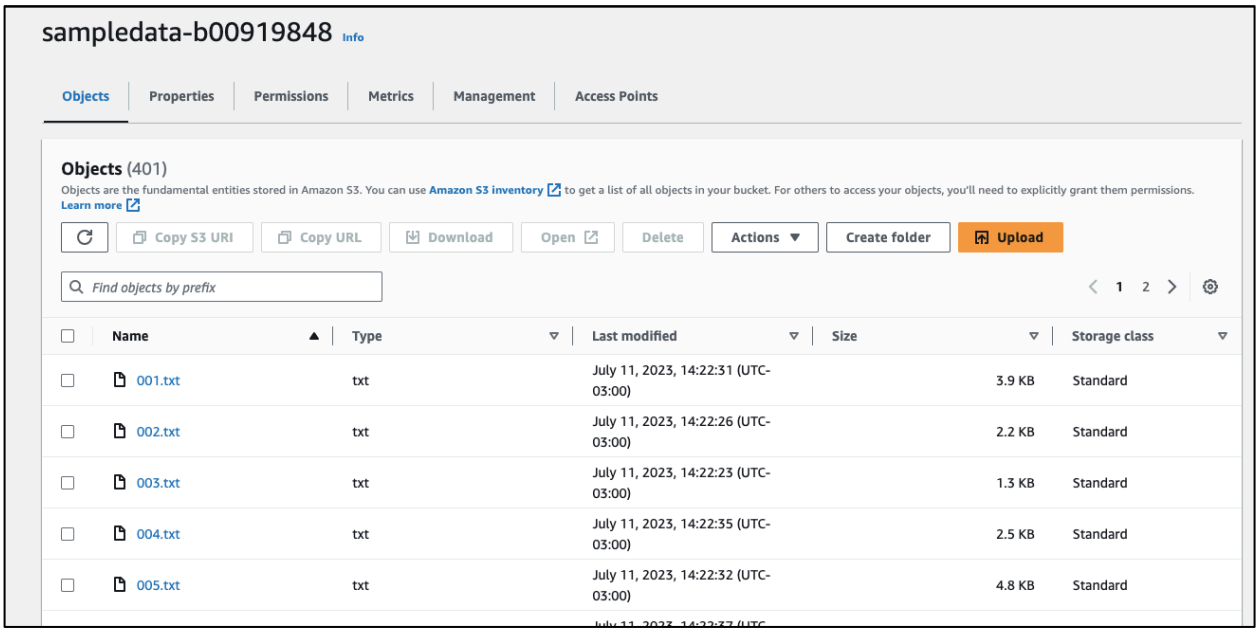

Fig 17. Tech Files Successfully Uploaded to "sampledata-b00919848" Bucket [1]

Furthermore, let us verify that the contents of the files are correct as well by accessing one of the files then downloading and verifying its contents as seen in Figures 18 and 19:

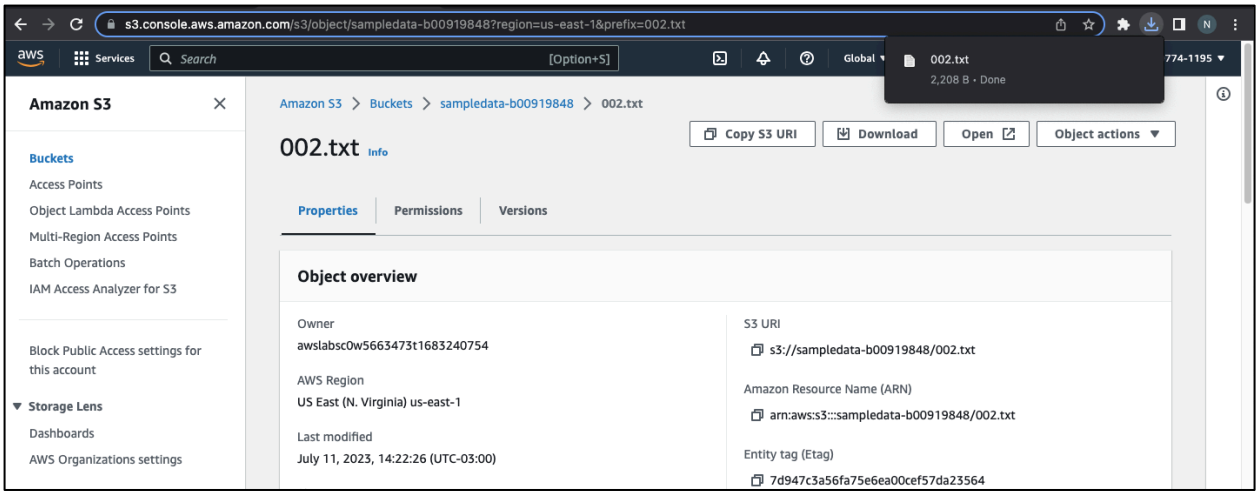

Fig 18. Downloading a File to Verify its Contents [1]

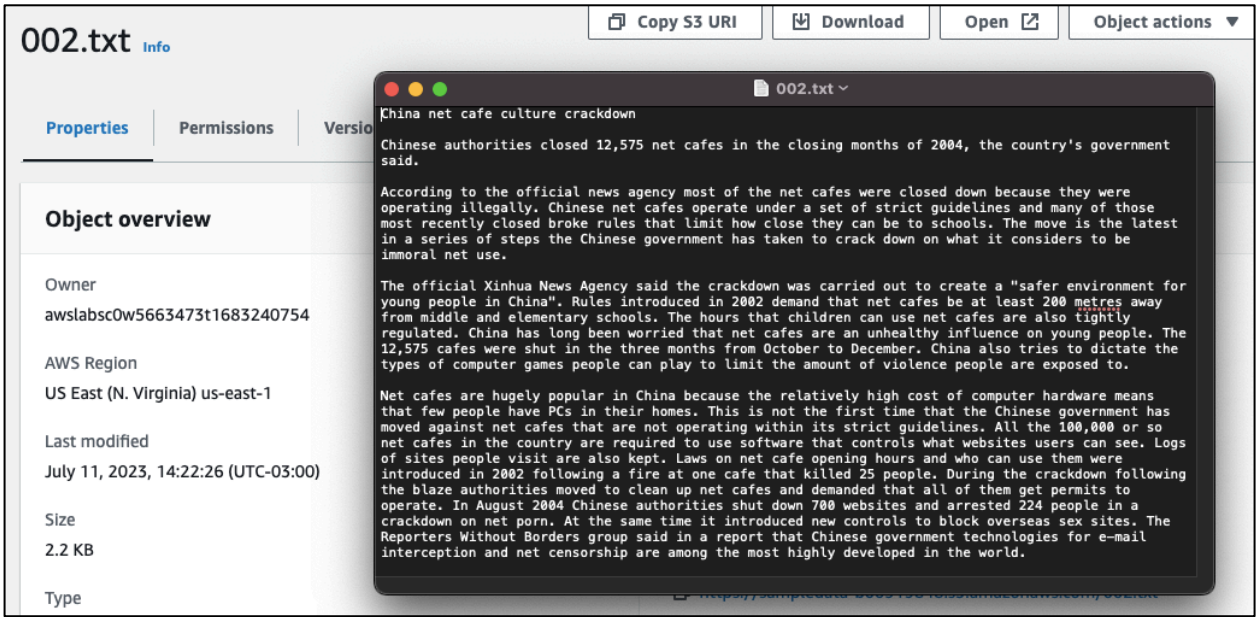

Fig 19. Verifying Downloaded File's Contents

Now that we have verified that the UploadFiles.py script has indeed uploaded all the Tech folder's files to the correct S3 bucket, we need to confirm that "extractFeature" Lambda was triggered on each upload to the "sampledata-b00919848" bucket. To verify that it has been triggered successfully, we navigate to the function's logs in Cloud Watch [4], as seen in Figure 20, where we can see a bunch of logs have been created:

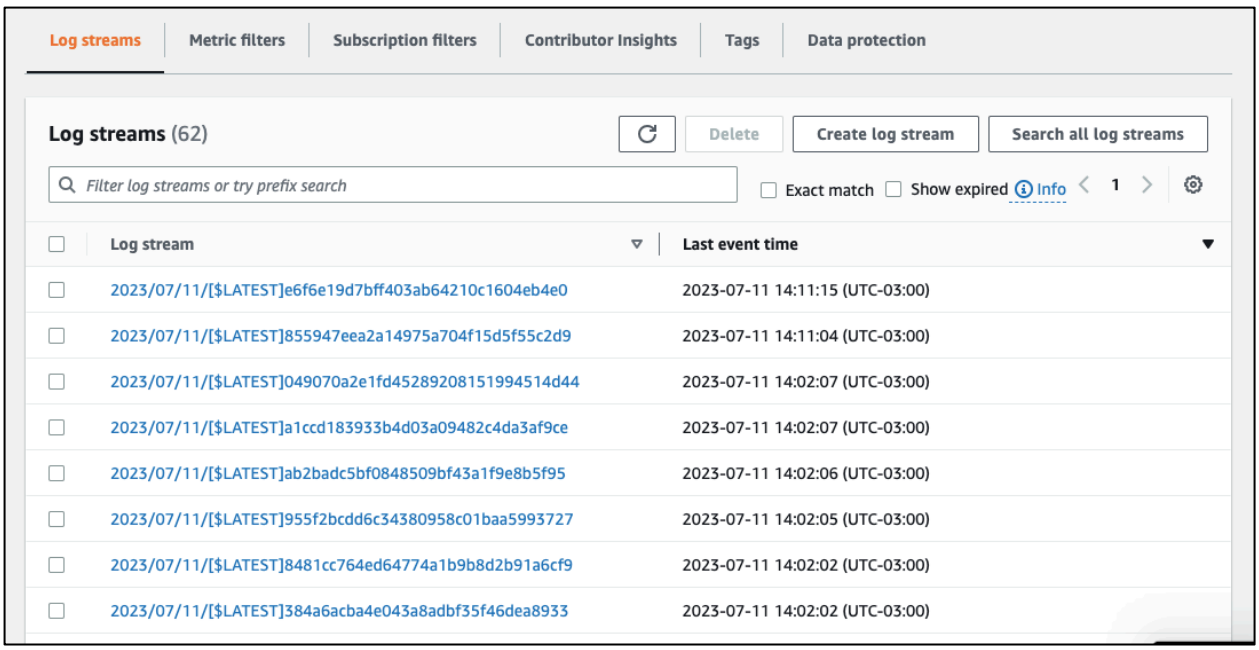

Fig 20. "extractFeatures" Lambda's Execution Logs in Cloud Watch [4]

Now, let us navigate inside one of these log streams to verify that "extractFeatures" is executing properly:

Fig 21. Sample "extractFeatures" Log Stream [4]

Figure 21 depicts a snippet of one of the log streams created by the "extractFeatures" invocation. The portion highlighted in the red box corresponds to a single Lambda invocation triggered by an upload to the "sampledata-b00919848" bucket. However, the most important part of the

highlighted section is the portion depicted in Figure 22 which logs the response the Lambda function got from uploading a file to the “tags-b00919848” bucket:

Fig 22. Expanded Response Log Line from Figure 21 [4]

Additionally, we must verify the contents of the “tags-b00919848” bucket to ensure that features were correctly extracted into a JSON string. So, we navigate to the “tags-b00919848” bucket as seen in Figure 23:

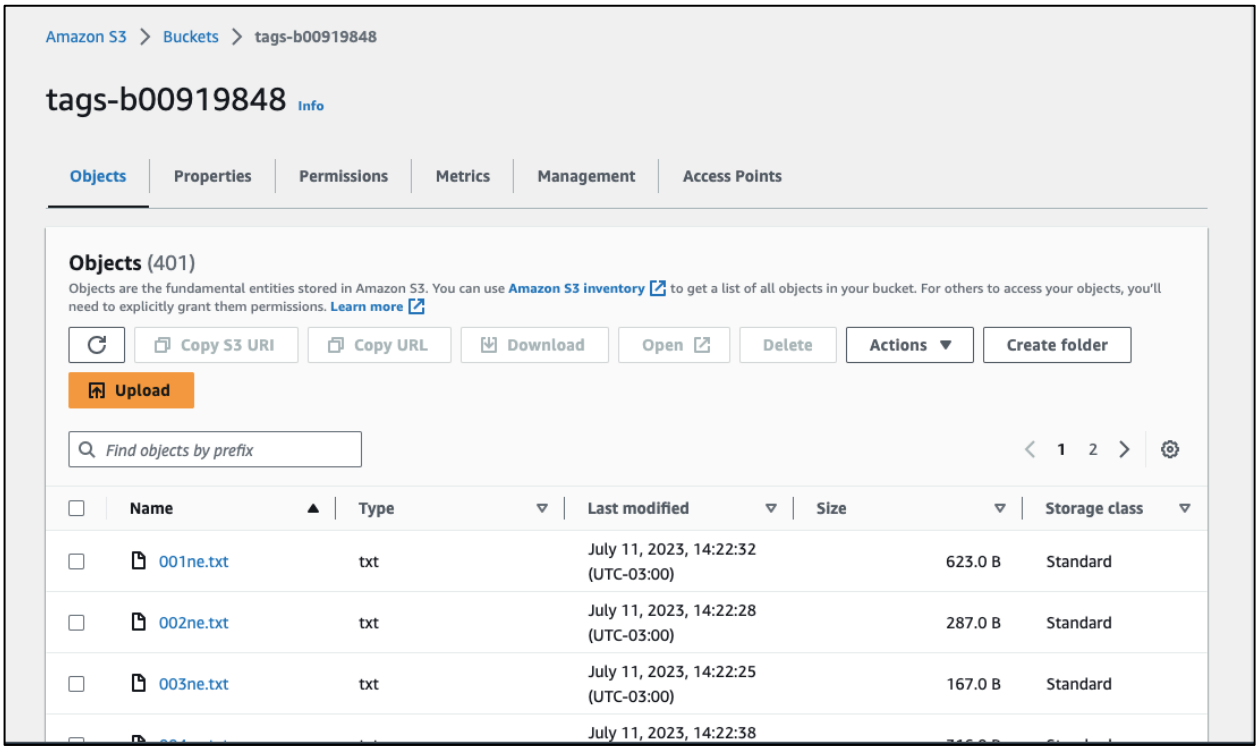

Fig 23. Extracted Named Entities Files Successfully Uploaded to “tags-b00919848” [1]

To verify the contents of the uploaded files, a sample file was downloaded and its contents verified as seen in Figure 24:

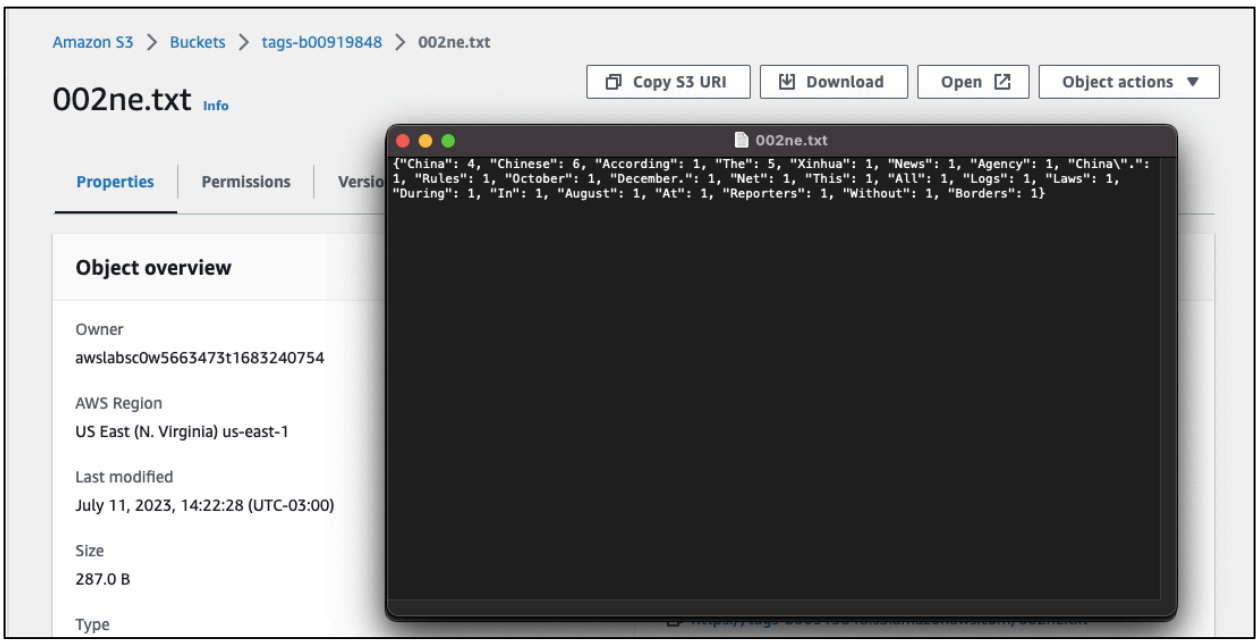

Fig 24. Verifying Sample Named Entity File in "tags-b00919848" [1]

Indeed, the named entity files have been successfully uploaded to the target "tags-b00919848" bucket via triggering the "extractFeatures" function by uploading objects to the "sampledata-b00919848" bucket.

Next, we must verify that our second Lambda function, "accessDB", has been triggered correctly as "extractFeatures" uploads each file to the "tags-b00919848" bucket; wherein "accessDB" then updates the "namedEntities" DynamoDB table accordingly. So, we once again navigate to Cloud Watch [4]. However, this time we check the "accessDB" logs, where we can see many logs have been created:

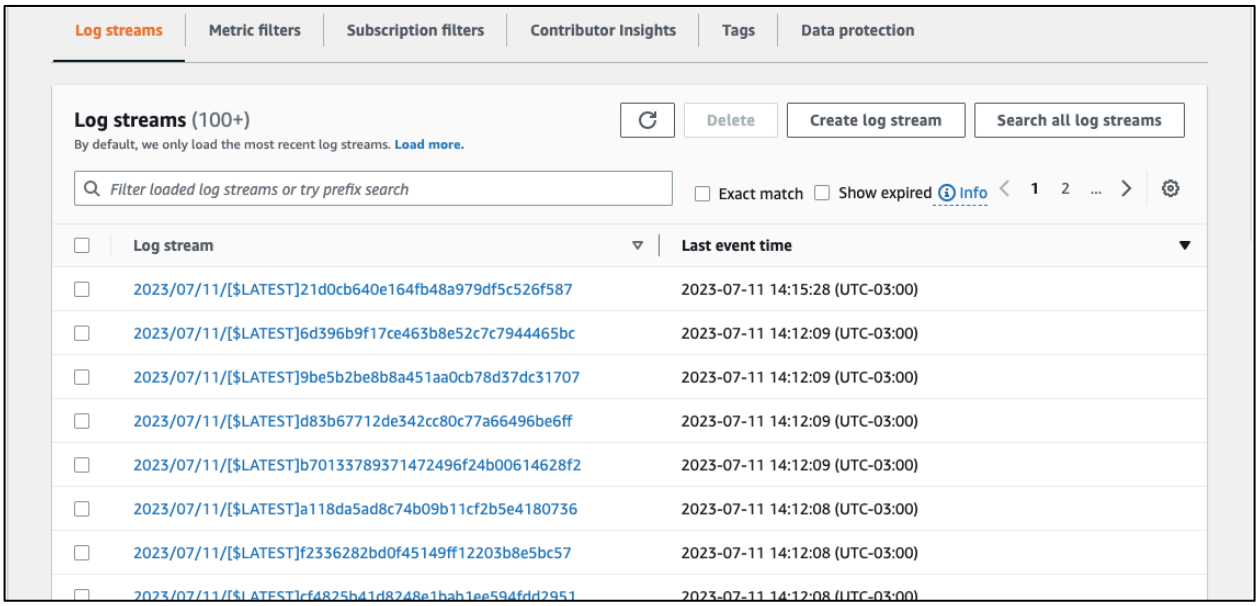

Fig 25. "accessDB" Cloud Watch Logs [4]

We then verify execution by checking a sample log stream as seen in Figure 26, where we can see that the request has been services with no exceptions being raised:

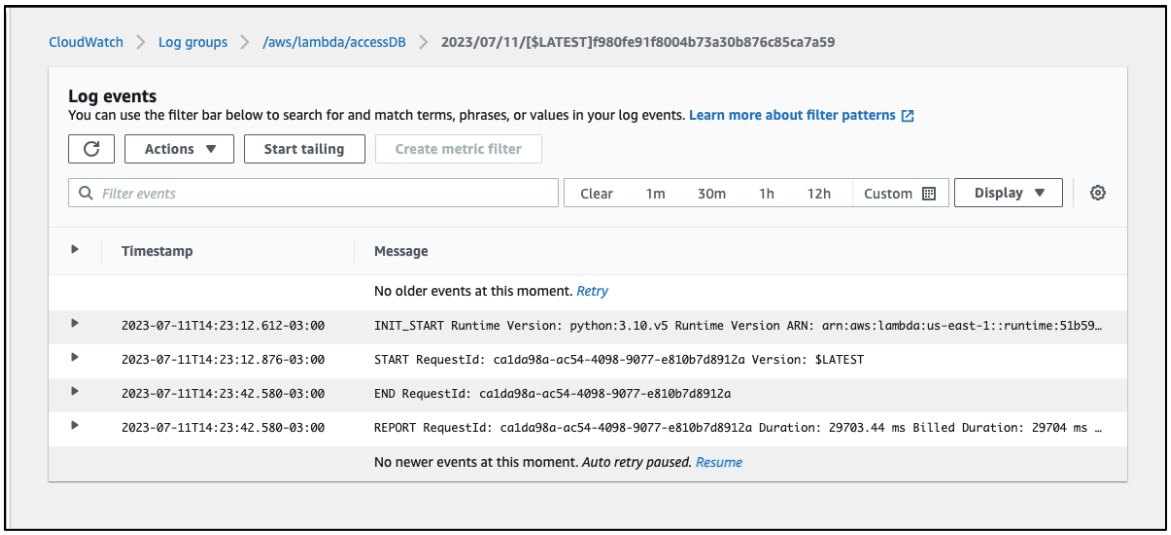

Fig 26. Sample "accessDB" Log Stream [4]

Figures 25 and 26 suggest that our "namedEntities" DynamoDB table should have been populated correctly since the many log streams correspond to each invocation triggered by "extractFeatures" uploading to "tags-b00919848". However, to verify this fact, we navigate to our "namedEntities" DynamoDB table, depicted in Figure 27, where we can see that 4,671 named entities and their tallies have been successfully stored in the table:

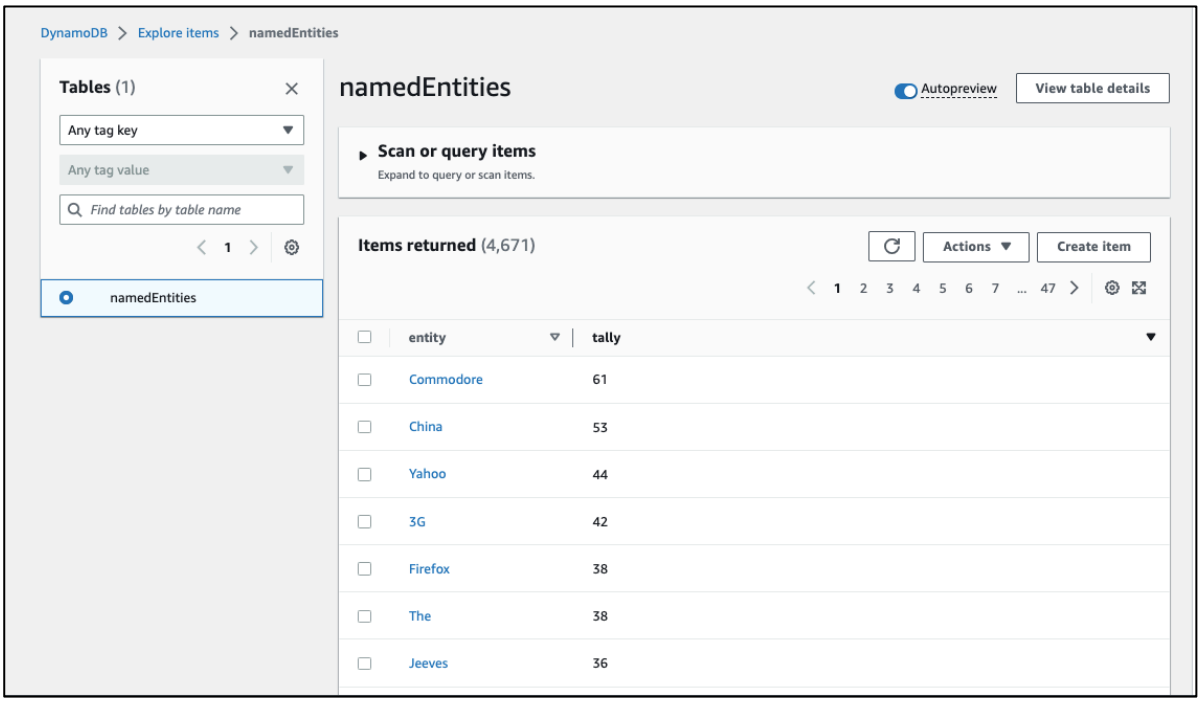

Fig 27. Verifying "accessDB" Correctly Stored Named Entities into "namedEntities" DynamoDB Table (Page 1)

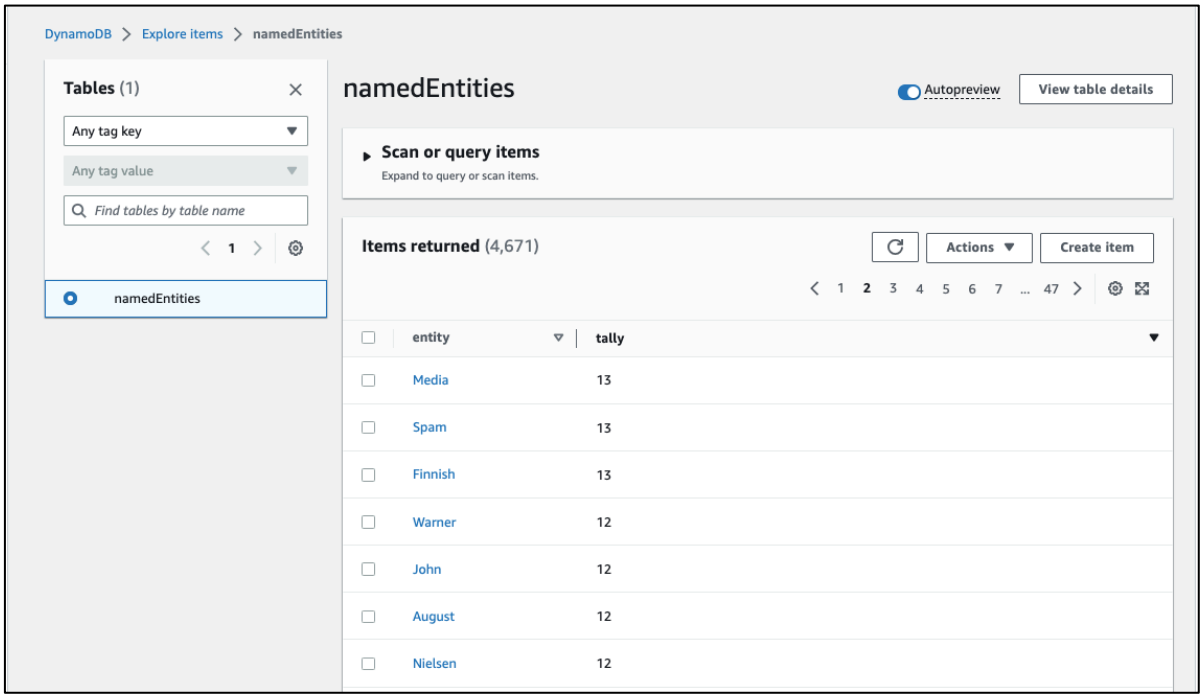

Fig 29. Verifying "accessDB" Correctly Stored Named Entities into "namedEntities" DynamoDB Table (Page 2)

Thus, we have verified that executing *only* the uploadFiles.py script correctly triggers "extractFeatures", which extracts the named entities of newly uploaded files and uploads it to "tags-b00919848"; triggering "accessDB" to extract the named entities from the newly uploaded file and update our "namedEntities" table accordingly.

References:

[1] Amazon Web Services Inc., "Amazon S3," *Amazon Web Services Inc.* [Online], Available: https://aws.amazon.com/s3/ [Accessed: July 11, 2023].

[2] Amazon Web Services Inc., "Amazon DynamoDB," *Amazon Web Services Inc.* [Online], Available: https://aws.amazon.com/pm/dynamodb/ [Accessed: July 11, 2023].

[3] Amazon Web Services Inc., "AWS Lambda," *Amazon Web Services Inc.* [Online], Available: https://aws.amazon.com/lambda/ [Accessed: July 11, 2023].

[4] Amazon Web Services Inc., "Amazon CloudWatch," *Amazon Web Services Inc.* [Online], Available: https://aws.amazon.com/cloudwatch/ [Accessed: July 11, 2023].