COSC6376 - Final Report

AWS Serverless OCR and Word Count Function

Raymond Fey, Haibo Chu, Jie Xu
rfey@uh.edu, hachu@uh.edu, jxu38@uh.edu

December 16, 2022

Abstract

In this project, we created a serverless Optical Character Recognition with a word count function that accepts text-filled images to be processed. The project runs on Amazon Web Services and provides the word count output as a .JSON file as an attachment to the user's email. Some of the improvements we added after the project proposal has been addressed. The improvements such as determining if a duplicate image was uploaded, deleting the image after processing, and incorporating a logging component were all added.

Contents

1 Introduction **3**

2 Description **3**

3 Design and Implementation **4**
3.1 Cloud System Architectural Design 4
3.1.1 Proposed Design 4
3.1.2 Project Feedback 4
3.1.3 Final Design 5
3.2 Code Implementation 7
3.2.1 Lambda Main Driver Function 7
3.2.2 Lambda Word Count Function 13
3.2.3 AWS Glue PySpark 14
3.2.4 Lambda Update DynamoDB Function 15

4 Testing **16**
4.1 New File 16
4.1.1 Preparing the Image 16
4.1.2 MD5 Hashing the Image 17
4.1.3 Uploading the Image 18
4.1.4 Processing the Image 19
4.1.5 Emailing Results 21
4.2 Duplicate File 22
4.3 Edge Cases 23

5 Milestones and Challenges **24**
5.1 Milestones 24
5.2 Challenges 24
5.2.1 Implementation Challenges 24
5.2.2 Miscellaneous Challenges 27

6 Lessons Learned **28**
6.1 What have we expected to learn? 28
6.2 What we actually learned? 28

7 Conclusion **28**

List of Figures

1 Proposed Top Level AWS System Architecture 4
2 Final Top Level AWS System Architecture 5
3 The Imported Python Libraries 8
4 Initialization of Variables 8
5 Checking DynamoDB 9
6 Calling Textract 10
7 Deleting the Image 10
8 Catching Results 11
9 Creating, attaching, and sending email 12
10 Lambda code to invoke the AWS Glue job 13
11 AWS Glue PySpark word count algorithm 14
12 Lambda code to add an entry to DynamoDB Table 15
13 Sample image used for testing 16
14 MD5 hash value in Terminal 17
15 DynamoDB table prior to file upload 18
16 S3 Bucket prior to file upload 18
17 S3 Bucket front end web-page 19
18 S3 bucket after file upload 20
19 Textract function called in CloudWatch logs 20
20 Creation of .CSV file 21
21 Creation of .JSON folder and file 21
22 JSON file fetched in CloudWatch logs 22
23 Email with JSON attachment 22
24 CloudWatch logs for a duplicate image 22
25 Non-verified email will result in a failed email 23

1 Introduction

With the rise of smartphones and their cameras, the rise of photo-taking has been increasing, as well as their importance. We then use it frequently to capture important information, notes, and memories and share that with others. Many of these photos may be documents that have a lot of text that can be benefited from OCR and word count.

2 Description

As mentioned in the introduction, taking pictures has become very popular as well as taking pictures of documents. This poses a great difficulty if the user wants to grab a `.txt` file or share it as a PDF. Furthermore, being a picture, getting information about word count or usage is much more problematic as it cannot be normally processed through a word editor.

In this project, we created a server-less function to allow users to upload a text-filled image to a storage bucket and the word count will be provided. The picture will be processed using Optical Character Recognition (OCR) and the word count through PySpark.

3 Design and Implementation

This section shows the design and implementation of the services used to achieve our project proposal.

3.1 Cloud System Architectural Design

In this project, we had the proposed cloud system architectural design and the final iteration of it.

3.1.1 Proposed Design

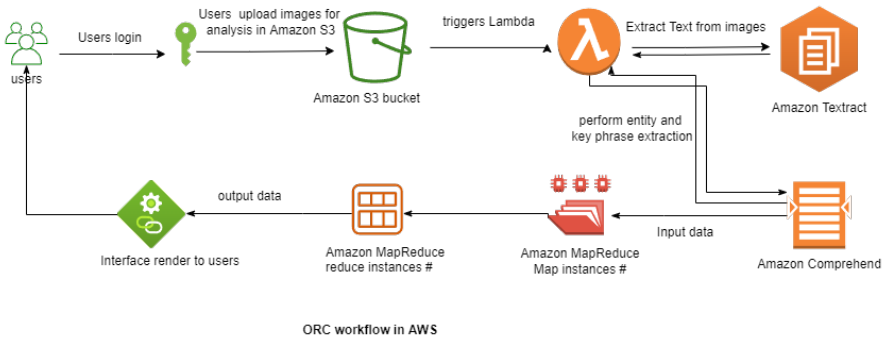


Figure 1: Proposed Top Level AWS System Architecture

In our original proposal, we proposed to use Amazon Web Services (AWS) to complete our project. The core services we expected to use were S3 Buckets for file storage, Textract for Optical Character Recognition, and Elastic MapReduce for Word Count, as seen in **Figure 1**.

We initially had a workflow that required user(s) to log in through a ReactJS web page with their credentials and have the option for them to register as well. Once the user has been successfully authenticated, the user will be able to upload their text-filled images to their own S3 buckets. The Lambda function would be triggered by any new image objects and invoke AWS Textract to transcribe the image into text. The transcribed text would then be inputted into Elastic MapReduce and the results would be displayed on their web page.

3.1.2 Project Feedback

After proposing our initial design, our group was provided feedback to add more tasks to our project. The feedback were as followed:

- If you get an exact image twice (a duplicate), then the system should not make another AWS Textract call and should continue to be handled.

- Images should not be saved in the S3 bucket after getting processed. It should be deleted, but the previous point should still work.
- Have a logging component to monitor your system.

In the next section, we will see that we addressed and implemented all those points.

3.1.3 Final Design

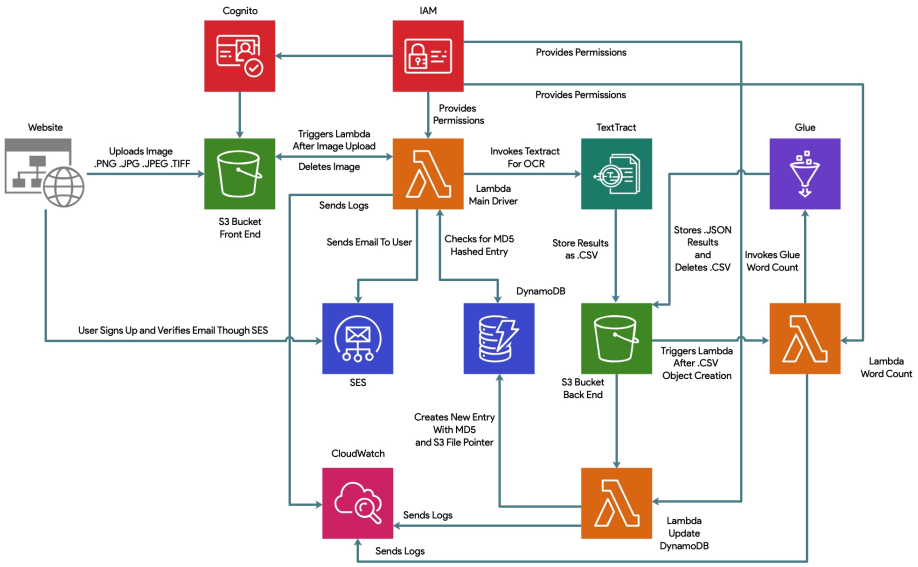


Figure 2: Final Top Level AWS System Architecture

The top-level cloud system architectural design, as seen in **Figure 2** has significantly changed from the proposed one.

In this project, we used Amazon Web Services to complete our project, and the specific services were used as follows:

- AWS S3 Buckets
- AWS Lambda Functions
- AWS SES
- AWS DynamoDB
- AWS Textract
- AWS Glue

- AWS IAM
- AWS Cognito
- AWS CloudWatch

In **Figure 2**, the system architecture can be split into two sections. The first being the front end and the other being the back end.

The front end is what a normal user will interact with, which in this project, will be through a hosted static website. The user will be able to upload pictures directly to the front-end S3 bucket and verify their emails through AWS SES through the website.

In the left-most part of the diagram, we moved away from a restrictive registered user-login-only option to publicly open our service to anyone, as long as the user register and verify their email address with AWS SES. We moved away from the restrictive option, due to not knowing how to effectively pass usernames and passwords securely without putting them in plain-text as parameters in the URL.

As for the front-end web page, we wrote a simple HTML5 code with Javascript scripts embedded and had it hosted directly from the S3 bucket itself. By hosting the site from the S3 bucket, we can publicly allow access instead of requiring some networking setup, and port forwarding, to localhost on our computers. Users who access this web page have the opportunity to provide their email address to verify through AWS SES and also upload images.

Note: Users will never be able to see the contents of the S3 buckets.

The middle part of the diagram, which is the Lambda Function, is the main driver of this project. This Lambda Function will be triggered anytime a supported image type with the suffixes `.PNG`, `.JPG`, `.JPEG`, `.TIFF` is uploaded to the front-end S3 Bucket. What the Lambda Function does first checks the MD5 hash value of the image file, which AWS S3 conveniently places in the `ETag` metadata. The function will compare this MD5 hash value against the DynamoDB table entries. The sole purpose of this action is to address duplicates of an image so that less time and cost are required to produce a result.

If an entry with the said MD5 hash value is already available in the DynamoDB table, we can skip the latter parts of the diagram and directly grab the file results from our backend S3 bucket. We stored the file pointer of the results as attributes in the DynamoDB entry. With that, we can directly create and attach the results to an email, which is shortly sent to the user.

Now, let us say that the file is unique, or has never been uploaded to our service before. This would imply that we do not have an entry with the said MD5

hash value in the DynamoDB table. This cause our main Lambda Function to call Textract. Because Textract and other services will take time to generate the results and put a new entry into the DynamoDB table, we put the main Lambda Function in a waiting function. The waiting function will query the DynamoDB table every 2 seconds until either a result is able to be fetched or it times out. The timeout of 5 minutes is to prevent the Lambda Function from deadlocking. Afterward, an email is sent to the user notifying that the results have been attached, or in the worst case scenario, that the user should try again as our timeout function has been triggered.

Textract is a fairly simple AWS service, and it accepts almost any image file type. It will ingest the text-filled image and return a `.CSV` file of the OCR results to our back-end S3 bucket.

In **Figure 2**, we can move to the right-most Lambda Function, which is soley responsible for checking if any new `.CSV` file has been put into the back-end S3 bucket. If so, this Lambda Function will invoke the AWS Glue job which performs our word count. AWS Glue ingests the `.CSV` file and generates a `.JSON` file of the word count results. Shortly after, the Glue job deletes the `.CSV` file.

On the bottom-most Lambda Function, this function is triggered only if an object ends with a `.JSON` suffix. When a new object with that suffix is put into the back-end S3 bucket, this function will create a new entry into the DynamoDB table with the MD5 hash value as the partition key, and bucket name, and file name as the other two attributes.

3.2 Code Implementation

In this section, we go over the implementation of several different scripts written for Lambda functions and Glue PySpark.

3.2.1 Lambda Main Driver Function

This is the heart of our project, where most of the processing and decision-making are made. The core tasks performed by this Lambda Function are:

- Get the image file from front end S3 bucket
- Determine if the image is new or duplicate using MD5 and querying DynamoDB
- Call TextTract if the image is new
- Delete the image from the front end S3 Bucket
- Get results from the back end S3 Bucket using file pointers in DynamoDB
- Email results as an attachment

```
import os
import time
import email
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.application import MIMEApplication
from botocore.exceptions import ClientError
from datetime import datetime
```

Figure 3: The Imported Python Libraries

In **Figure 3**, we are first importing the necessary packages that are required to run the function. The `os` library is used for performing operating system functions, such as splitting the file path of a file down to the sub-directories and the base file name. The `time` library is used for a timeout, which we will use in a while loop to break out, in order to not infinite deadlock the Lambda Function. This will be explored later in this section. The `email` library is to create and send out formatted emails with an attachment to users of the results.

```
def lambda_handler(event, context):
    read_bucket = event["Records"][0]["s3"]["bucket"]["name"]
    document = event["Records"][0]["s3"]["object"]["key"]
    s3_client = boto3.client("s3")
    s3 = boto3.resource("s3")
    s3_dynamo = boto3.resource('dynamodb')
    client = boto3.client('textract')

    #Get the MD5 hash of the Image Uploaded
    s3_resp = s3_client.head_object(Bucket=read_bucket, Key=document)
    s3obj_etag = s3_resp['ETag'].strip('"')
    print(s3obj_etag)

    s3_email = s3_resp['Metadata']['email']
    print(s3_email)
```

Figure 4: Initialization of Variables

Next, in **Figure 4**, we are interested in initializing variables that will invoke certain AWS services. Namely, `S3`, `DynamoDB`, and `Textract`. Additionally, the file that triggers this lambda function is also stored in variables `read_bucket` and `document`. Using this data, we can get information about the file that was uploaded to the front-end S3 bucket, such as the `MD5` hash value by getting the `ETag` value and our own user-defined metadata `email` to know whom to email the results to.

```
#We will then compare this against our DynamoDB Database
table = s3_dynamo.Table('HardCodersDB')
db_response = table.get_item(Key={ 'MD5':s3obj_etag})

#Flag for calling Textract
flag = 0

if 'Item' in db_response:
    #If the file already exists, return the stored JSON dictionary
    content_object = s3_client.get_object(Bucket=db_response['Item']['bucketpointer'],Key=
        ↪ db_response['Item']['filepointer'])
    #We will dump json_content for the front end
    json_content = content_object['Body'].read().decode()
    print("file exists")
else:
    #Else, the file doesn't exist, set flag to call TextTract
    flag = 1
    print('File does not exist! Calling TextTract!')
```

Figure 5: Checking DynamoDB

In **Figure 5**, the code block is responsible for checking out DynamoDB table to see if an item with the `MD5` partition key exists or not. We first set the `flag` variable to `0` to signify that it does exist. Now once the DynamoDB table response is fetched, we can instantly go to the S3 bucket to fetch our results. If not, we have to set the `flag` variable to `1` and prepare to call Textract.

```
if (flag == 1):
#GET the S3 Object for Textract
response = client.detect_document_text(
    Document={'S3Object':{'Bucket':read_bucket,'Name':document}})

#Get the text blocks
blocks = response['Blocks']

words = []
confidences = []

for block in blocks:
    if block['BlockType'] == "WORD":
        words.append(block['Text'])
        confidences.append(block['Confidence'])

my_data={'Word':words,'Confidence Level':confidences}
df = pd.DataFrame(data=my_data)
#Clean up by removing punctuation from the words
df['Word'] = df['Word'].str.extract('(\w+)', expand = False)

#Write to CSV
write_bucket = "hardcoders-back-end"
csv_buffer = StringIO()
df.to_csv(csv_buffer)

s3_resource = boto3.resource('s3')
#Strip off the file extension
file_name = os.path.splitext(document)[0]
s3_resource.Object(write_bucket, str(s3obj_etag) + '*' + str(datetime.now().strftime("%
    ↪ m_%d_%Y_%H_%M_%S")) + '.csv').put(Body=csv_buffer.getvalue()) #Use * as
    ↪  a separator
```

Figure 6: Calling Textract

In **Figure 6**, we are interested in performing the Textract and word count. We can see that we have an if conditional, which will be true if the file is new. Textract will return `JSON`-like responses of all the words and the remaining metadata on the text-filled image. We stored these results in a variable named `blocks`. Now, we were interested in writing the results into a `.CSV` file for easier processing, so the usage of Pandas and DataFrames was necessary. When writing our `.CSV` file to our S3 bucket, we wanted to append the `MD5` hash value to the file name.

```
#Delete the picture
object_to_be_deleted = s3.Object(read_bucket, document)
object_to_be_deleted.delete()
```

Figure 7: Deleting the Image

To satisfy the requirements of not preserving the image in the front-end S3 bucket, we will call the following commands seen in **Figure 7**. This will cause the image file to be deleted.

```
second_flag = 0
#Deal with waiting for DynamoDB Entry for new images
timeout = time.time() + 60*5 #5 Minute from now
if(flag == 1):

    while True:
        db_response = table.get_item(Key={ 'MD5':s3obj_etag})

        #Check DynamoDB for MD5 Hash value
        if 'Item' in db_response and time.time() < timeout:
            second_flag = 1
            break
        elif 'Item' not in db_response and time.time() > timeout:
            break

        print('Waiting...')
        #Poll every 2 second
        time.sleep(2)

    if(second_flag == 1):
        db_response = table.get_item(Key={ 'MD5':s3obj_etag})
        content_object = s3_client.get_object(Bucket=db_response['Item']['bucketpointer'],Key
            ↪ =db_response['Item']['filepointer'])
        json_content = content_object['Body'].read().decode()
        print('got content!')
    else:
        json_content = {'Error':'Timed Out. Try Again!'}
        print('Timed out')
```

Figure 8: Catching Results

Here, in **Figure 8**, we are concerned with waiting for the result to be put into the DynamoDB table. We can observe in the code block that we have an infinite loop. Within the while loop, we query the DynamoDB table to see the new entry. Then we go through our conditionals. The first conditional checks if the item exists and the timeout period has not passed. If this is satisfied, we set the other variable, `second_flag` to `1`. The second conditional checks to see if the item is not in the DynamoDB table within the set timeout. This causes the while loop to break. This while loop occurs every 2 seconds to lower the resource cost.

Later in the code block, we are determining how to send the email. If the `second_flag` variable is set to `1`, we will get the object from the back-end S3 bucket using the bucket pointer and file pointer provided by the DynamoDB query result. If this is not the case, we will prepare an email stating that the process has timed-out.

```
#Preapre for SES Email of Results

SENDER = "fey432@gmail.com"
RECIPIENT = s3_email
AWS_Region = "us-east-1"
CHARSET = "UTF-8"
mail_client = boto3.client('ses',region_name=AWS_Region)

if(flag == 1 and second_flag == 0):
    SUBJECT = "Error Trying to Get Your Results"
    BODY_TEXT = "There was an error trying to get your results. Please try to upload
        ↪ again!"
    try:
        email_response = mail_client.send_raw_email(
            Destination={
                'ToAddresses':[
                    RECIPIENT,
                ],
            },
            Message = {
                'Body':{
                    'Text':{
                        'Charset':CHARSET,
                        'Data' : SUBJECT
                        }
                },
                'Subject':{
                    'Charset' : CHARSET,
                    'Data' : SUBJECT
                }
            },
                Source = SENDER
        )
    except ClientError as e:
        print(e.response['Error']['Message'])
    else:
        print("Email sent!")
else:
    SUBJECT = "Your OCR Results are Available"
    BODY_TEXT = "Hello,\r\nYour OCR Word Count Reduce Results are ready! Please see
        ↪  the attached JSON File\r\nSincerely,\r\nHard Coders"

    #Temp dump
    base_name = os.path.basename(db_response['Item']['filepointer'])
    tmp_file_name = '/tmp/' + base_name
    s3_client.download_file(db_response['Item']['bucketpointer'],db_response['Item']['filepointer'
        ↪ ],tmp_file_name)
    attachment = tmp_file_name

    <...More Code...>
```

Figure 9: Creating, attaching, and sending email

Finally, in **Figure 9**, the code to send the email and the attachment is written. Due to the sheer length of the code block to create an email with an attachment, the tail portion of the code was truncated. For the full code, please reference the source code.

3.2.2 Lambda Word Count Function

```
import json
import os
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

#Import Boto3 for AWS Glue
import boto3
client = boto3.client('glue')

#Variables for the job:
glueJobName = "PySpark_MapReduce"

def lambda_handler(event, context):
    bucket = event["Records"][0]["s3"]["bucket"]["name"]
    document = event["Records"][0]["s3"]["object"]["key"]
    response = client.start_job_run(JobName = glueJobName, Arguments = {'--
        ↪ bucket_name': bucket,'--file_name':document})
    logger.info('## STARTED GLUE JOB: ' + glueJobName)
    logger.info('## GLUE JOB RUN ID: ' + response['JobRunId'])

    return response
```

Figure 10: Lambda code to invoke the AWS Glue job

The code block in **Figure 10**, is fairly simple. Its only role here is to wait for a `.CSV` to be put in the back-end S3 bucket and trigger. When it does trigger, the Lambda function passes the bucket name and the file name and invokes the AWS Glue job.

3.2.3 AWS Glue PySpark

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
import boto3

## @params: [JOB_NAME]
args = getResolvedOptions(sys.argv, ['JOB_NAME','bucket_name','file_name'])

sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

s3_file_path = "s3://" + str(args['bucket_name']) + "/" + str(args['file_name'])
df = spark.read.format("csv").option("header","true").load(s3_file_path)
df = df.withColumn("Word",lower(col("Word")))
df = df.groupBy('Word').count()
df = df.sort(desc('count'))

#We will need to change the folder name of the JSON file to the Unique identifier
#Take off the .csv from the filename. Remove last 4 characters
file_name = args['file_name']
file_name = file_name[:-4]
df.coalesce(1).write.format('json').save('s3://hardcoders-back-end/' + str(file_name) + '.
    ↪ json')

#Delete the CSV file_name
s3 = boto3.resource('s3')
object_to_be_deleted = s3.Object(args['bucket_name'], args['file_name'])
object_to_be_deleted.delete()
job.commit()
```

Figure 11: AWS Glue PySpark word count algorithm

In **Figure 11**, the AWS Glue PySpark algorithm can be seen. In this code block, we first grab the file from the back-end S3 bucket, using the bucket name and file name we passed with the Lambda function. The Glue job will read the `.CSV` file and convert all the words to lowercase. Changing all the words to either all lowercase or uppercase is crucial in performing the word count. Once we made all the words in lowercase form, we then performed a `groupBy()` command to perform our "map reducing" -like word count. Then, we sorted the count in descending order.

Now that we have the word count, we need to write the results to the back-end S3 bucket. Due to AWS Glue running multiple processes at once, we need to coalesce our output into a single file and write it as a `.JSON` file. We will still preserve the `MD5` hash value in our file name for easier processing. Unfortunately, we can only control the name of the folder of the output; however, this is not

a huge issue, as we can still trigger our other Lambda Function with a `.JSON` suffix. After the `.JSON` file is written, we delete the `.CSV` file from the back-end S3 bucket.

3.2.4 Lambda Update DynamoDB Function

```
import json
import boto3

def lambda_handler(event, context):
    #Get the Bucket Name
    read_bucket = event["Records"][0]["s3"]["bucket"]["name"] #This should be hardcoders-
        ↪ back-end

    #Get the Folder Name
    folder = event["Records"][0]["s3"]["object"]["key"] #This should be [md5][date].json
        ↪ folder

    client = boto3.client('s3')

    #Get the Folder and the Single File
    response = client.list_objects_v2(Bucket = read_bucket, Prefix = folder)

    #Grab the MD5 Value for the MD5 attribute for DynamoDB
    MD5_value = response['Prefix'].split('*')[0]

    #Create Item in DynamoDB
    db_client = boto3.client('dynamodb')

    data = db_client.put_item(
        TableName='HardCodersDB',
        Item={
            'MD5':{
                'S': MD5_value
            },
            'bucketpointer':{
                'S': read_bucket
            },
            'filepointer':{
                'S': response['Prefix']
            }
        }
    )

    return {
        'statusCode': 200,
        'body': json.dumps(data)
    }
```

Figure 12: Lambda code to add an entry to DynamoDB Table

Finally, with our last Lambda function in **Figure 12**, the function is triggered whenever it sees a new object with the `.JSON` suffix that was put by the AWS Glue job. When this function gets triggered, it takes our customized folder name with the `MD5` hash value and uses it as a new entry for our DynamoDB table.

4 Testing

In this project, we have two test cases to work with. The uploaded picture image is either a new image or a duplicate of a previous upload. We have tested our code with both test cases to verify that the intended behavior is satisfied.

4.1 New File

This test case tests the process of uploading an image for the first time to our S3 bucket.

4.1.1 Preparing the Image

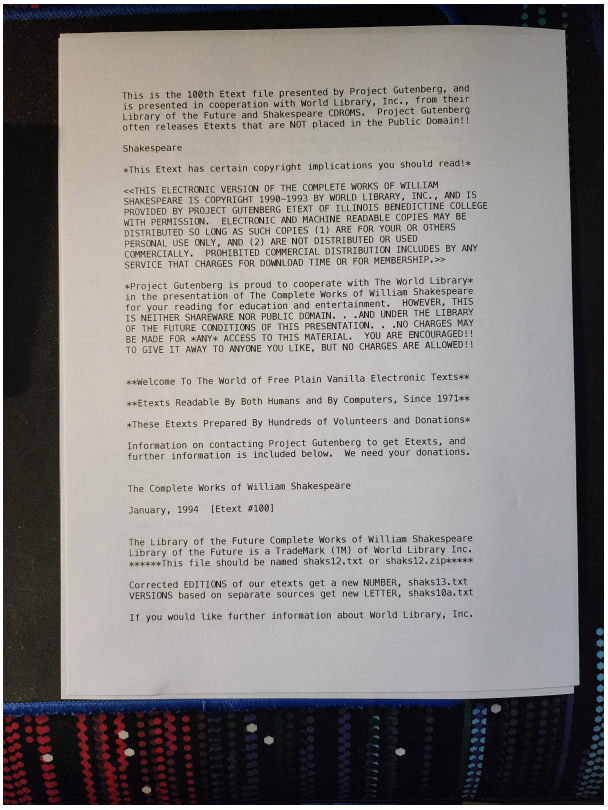


Figure 13: Sample image used for testing

In this example, we are going to upload a photo for the first time. The image of the test image is shown in **Figure 13**.

The text in the image is from Shakespeare's collection of plays and is a good test

for our serverless function. The captured image must end with the supported file formats, such as `.PNG`, `.JPG`, `.JPEG`, or `.TIFF`. Furthermore, the filename must not include any special or restrictive characters, as this will break the code.

4.1.2 MD5 Hashing the Image

We will first get the MD5 hash value of this file and check the DynamoDB table to ensure that the item does not exist. To perform an MD5 hash on macOS, perform the `md5` command, or on Linux, perform the `md5sum` command. The `MD5` hash value of the image in **Figure 13** is shown in **Figure 14**.

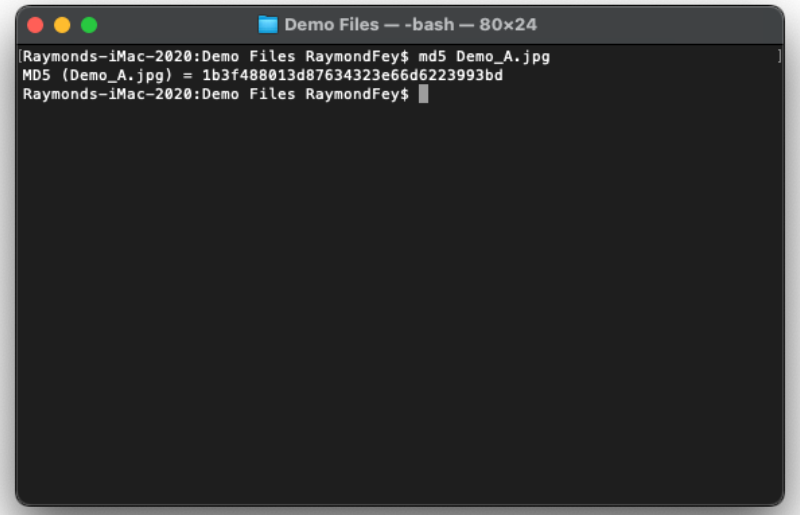


Figure 14: MD5 hash value in Terminal

As we can see in **Figure 14** , the MD5 hash value produced by the file is `1b3f488013d87634323e66d6223993bd`. Now we can go into DynamoDB and verify it does not exist as an item.

4.1.3 Uploading the Image

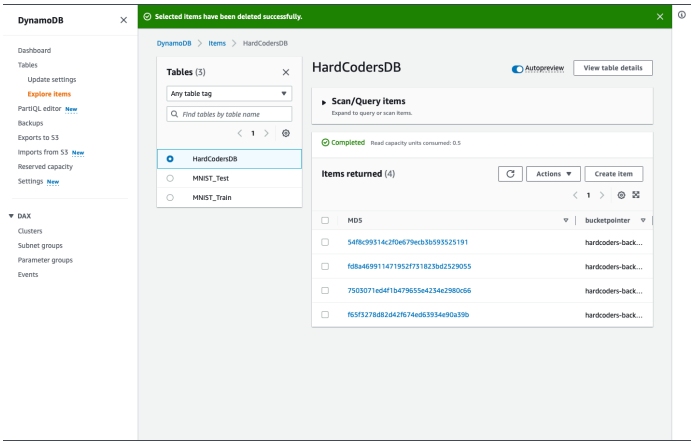


Figure 15: DynamoDB table prior to file upload

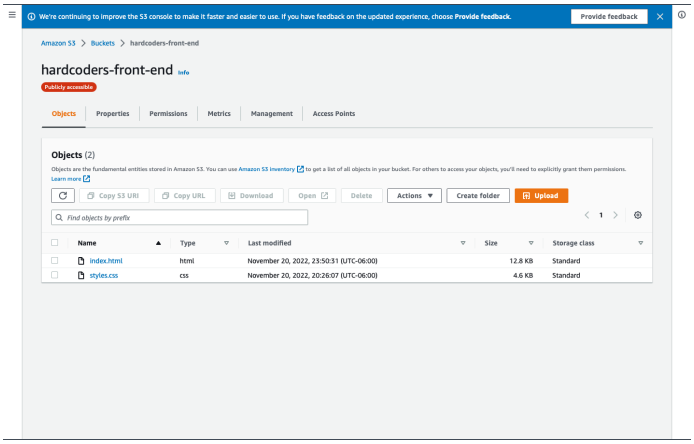
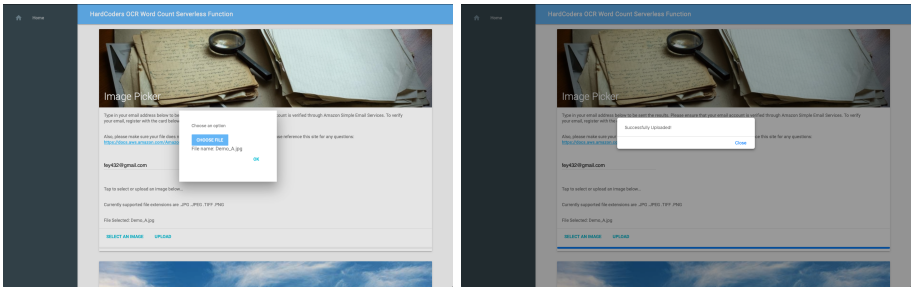


Figure 16: S3 Bucket prior to file upload

In **Figure 15**, we can observe that we do not have an MD5 hashed value entry in our DynamoDB table. With this in mind, we can indeed verify the image has not been uploaded before; therefore, we can continue uploading the image to our bucket using the front-end website. Furthermore, let us take a look at our Front end S3 bucket, seen in **Figure 16**, to ensure that no magic or cheating has occurred.

Now, we can upload the image file using the S3 statically hosted website, as seen in **Figure 17**.



(a) Selecting image to upload (b) Successfully uploaded image to S3

Figure 17: S3 Bucket front end web-page

In **Figure 17a**, we chose the file `Demo_A.png`, which is the picture we saw in **Figure 13**. Then we proceeded to type in our AWS SES verified email into the text field and pressed upload. Shortly, we will get a pop-up box that informs us that the file has been successfully uploaded to our S3 bucket.

Note: We have pre-verified our emails with AWS SES with the verification system provided on our front-end page. It is essential to verify your email with AWS SES prior to uploading any file. Not verifying your email ahead of time will result in you not receiving an email of the results; however, the processing of the text data will continue to be performed and ready for future requests.

4.1.4 Processing the Image

Now that the file has been successfully uploaded to our S3 front-end bucket, we must be swift with verifying that the file has indeed been uploaded. Our main Lambda Function quickly ingests the image file and deletes it afterward. In **Figure 18**, we can indeed verify that the image has been uploaded.

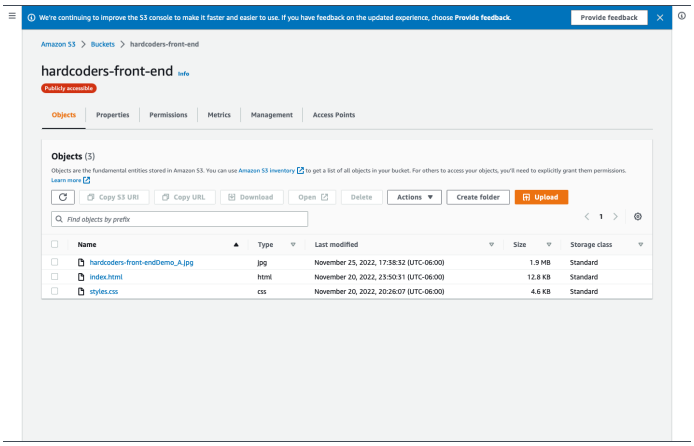


Figure 18: S3 bucket after file upload

Because this image file has never been processed before, the Lambda function will call Textract to perform optical character recognition (OCR) on the image file. We can verify these actions by going to our AWS CloudWatch logs on our main Lambda function.

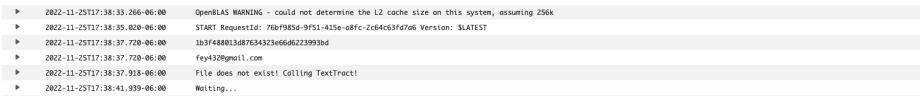


Figure 19: Textract function called in CloudWatch logs

As we can see in **Figure 19**, we can pull the MD5 hashed value of the file that AWS S3 has determined, and the user-defined metadata that provides us the user's email to send the results to. Then based on the MD5 hash value, we query our DynamoDB table to see if that MD5 hash value is an entry. As we saw earlier, this entry doesn't exist; therefore, AWS Textract will be called.

Now, because many more services need to be run before sending an email of the word count results, the Lambda function is put into a holding pattern or a while loop. The Lambda function will constantly query the DynamoDB table with the MD5 hash value, every two seconds, to see whether a new entry has been made or not. To ensure that we do not put a deadlock on our holding pattern, we have a timeout to break the while loop after 5 minutes.

After AWS Textract has finished, it will generate a `.CSV` file in our S3 Back-end bucket, as seen in **Figure 20**.

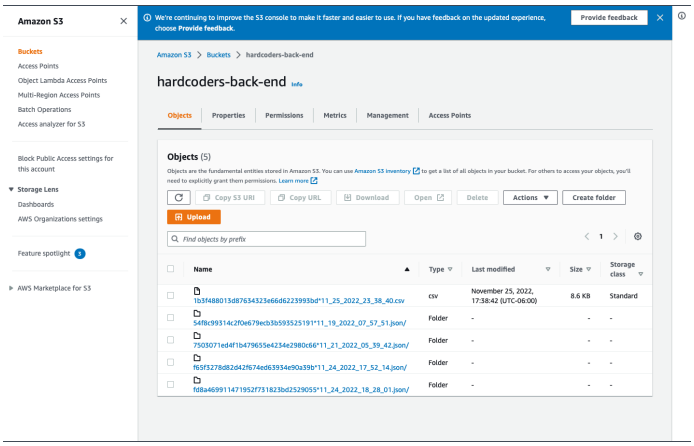


Figure 20: Creation of .CSV file

We have another Lambda Function that takes care of turning the `.CSV` file results and performing a word count. This Lambda function invokes our AWS Glue job and passes the pointer to the `.CSV` file in the S3 back-end bucket. We used AWS Glue and PySpark to perform our *pseudo* Map Reducing to get a word count. Once AWS Glue is done processing the file, it will create a `.JSON` folder with the `.JSON` file within and delete the `.CSV` file. The results are seen in **Figure 21**.

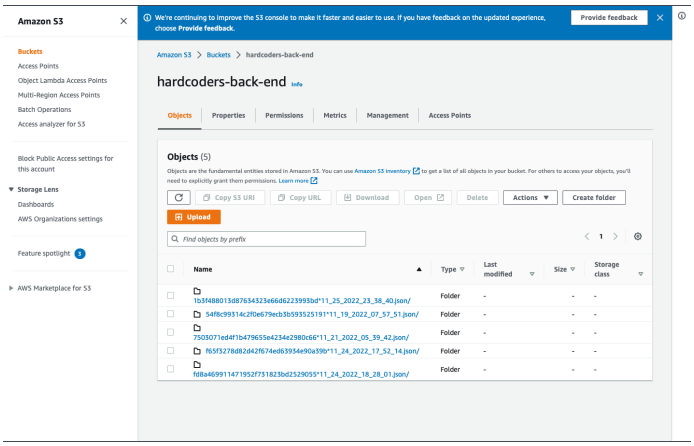


Figure 21: Creation of .JSON folder and file

4.1.5 Emailing Results

In **Figure 22**, we can see in our CloudWatch logs that after waiting a while, the entry in the DynamoDB table will be created with the bucket and file pointer. The Lambda Function will fetch the file from the S3 bucket and place the file

in the `/tmp/` directory. We can then generate an email with an attachment to the `.JSON` file, and shortly sends it out

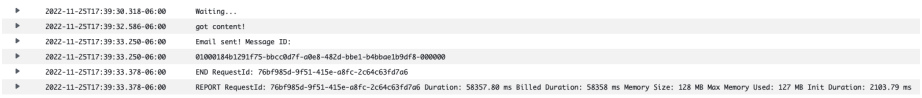
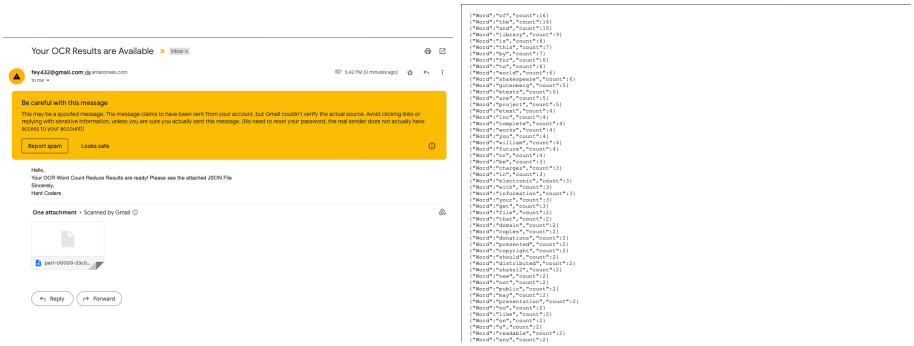


Figure 22: JSON file fetched in CloudWatch logs

The total amount of time for this test was about 1 minute long. The time it actually takes to complete is greatly dependent on how big the image file is, how many words, and how quickly the Glue Job instance can be spun up.



(a) Email that was received (b) JSON attached results

Figure 23: Email with JSON attachment

Shortly, an email with the `.JSON` results are received and the attachment can be opened, as seen in **Figures 23a** and **23b**.

4.2 Duplicate File

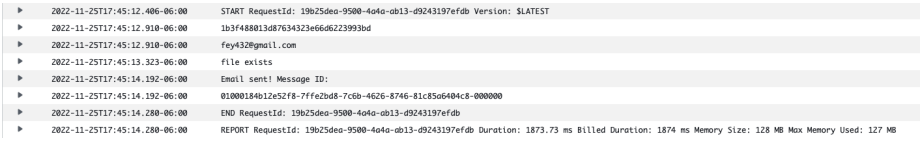


Figure 24: CloudWatch logs for a duplicate image

In this scenario, we will be uploading the same image, as seen in **Figure 13** to our S3 bucket. In this case, the amount of time to send an email out to the user is much quicker, at about 2 seconds, as the MD5 hash already exists in the DynamoDB table and can bypass the other services.

4.3 Edge Cases

There are a couple of edge cases that are present but are not tested, as we rarely expect them to happen. Some of the edge cases are if the user forgets to verify their email with AWS SES ahead of time or the AWS Glue job takes too much time that the while loop will timeout.

Figure 25: Non-verified email will result in a failed email

In the case that the user forgets to verify their email with AWS SES ahead of time, the normal process of either determining whether the image is new or a duplicate will continue as is. However, when it is time to send out the email, the email will fail to be sent as seen in **Figure 25**.

Unfortunately, with this scenario, our website will not be able to catch the error that the user isn't verified as it is a statically hosted site.

The other edge case is if the word count process takes too long and results in our while loop timeout. We can intentionally set the timeout to be one minute long, which is likely to timeout for a new image. What will result from this is that an email will be sent to the user stating `Error Trying to Get Your Results` and will ask the user to try the upload again. Even though the Lambda function has timed out, the word count process is still continuing. Most likely, by the time the user re-uploads the file, the entry will already be present in the DynamoDB table and the results will be sent swiftly.

5 Milestones and Challenges

5.1 Milestones

1. Milestone 1
    - In Milestone 1, we initially proposed to have completed the core functionality and individual resources working by October 24, 2022.
2. Milestone 2
    - In Milestone 2, we initially proposed to have completed the integration of different resources by November 7, 2022.
3. Milestone 3
    - In Milestone 3, we initially proposed to have performed refinements and added more features to our project by November 10, 2022.
4. Results
    - Unfortunately, we missed every single deadline for our milestones. Our group was only able to complete Milestone 1 by November 16, 2022, and Milestone 2 by November 20, 2022. We were not able to complete Milestone 3 of refinements and then adding additional features.

5.2 Challenges

Like with any project, we have faced several challenges that impacted our ability to complete the project. The following are the challenges we faced and how we addressed them. Some of the challenges we faced were implementation issues and some were other miscellaneous ones.

5.2.1 Implementation Challenges

1. How to code the front-end page?
    (a) Designing the front-end user interface for the website was not too much of a problem with HTML5 and CSS; however, was the actual logic and linking of AWS services that posed a bigger issue. We initially wanted to create a front-end page using React with NodeJS to allow the uploading of files to the S3 bucket. Even though this is possible, the learning curve of trying to implement it was too high. We had to attempt to use tokens through API Gateway, but the result was not consistent, as some of the times we would get a REST API return code of `404` or `5XX`. Furthermore, the website would only be hosted on a local machine, rather than being able to be accessed from a hosted site.

We even attempted to use AWS Amplify to build our website and do the authentication and S3 bucket access for us, but AWS Amplify Studio User Interface builder was severely limited and required installation of `npm` which sometimes wouldn't compile or build correctly.

(b) We resolved this issue by going to old-school HTML5 and Javascript coding, but this time putting the `index.html` page directly in the front-end S3 bucket and hosting the static from there. With this approach, we can more easily implement direct file uploads.

2. How to determine a duplicate image?

   (a) This was a trivial challenge that posed to us. Some of us took some sort of cybersecurity courses, so we knew some variation of file hashing had to be performed to determine if a file has been uploaded before. The most common hashing type is `MD5` hashing, but we were initially not sure how to get the hash value. We looked at using the `hashlib` library in Python, but could not get actually hash the file.

   The second part of the issue is where are we supposed to reference this comparison. Storing the images in the bucket forever and hashing every file every time a new picture comes in was not an efficient approach.

   (b) We looked further into the metadata of how S3 objects are generated when they are uploaded. Conveniently, AWS S3 Buckets generates a `MD5` hash value for **every** file uploaded, and stores them in the `ETag` attribute of the metadata. Now with this hash value, we are able to determine if a file has been uploaded before by comparing the `MD5` value. To address the second part of the issue, we created a DynamoDB table that stores the `MD5` hash value as the partition key, and we can easily compare to check.

   Of course, this is not a perfect solution, because a similar photo of the same document can be taken, but each would have a different hash value. However, we believe this is the best approach without calling more expensive resources such as `OpenCV` to make this decision.

3. How to know what file pertains to which request?

   (a) This issue was a genuine issue that really put a hold on our project until a hack-and-wack solution was made. Due to our separation of Lambda functions, figuring out which file belongs to which request got really messy.

   (b) We resolved this using an ugly method and some brute forcing. We put the `MD5` hashed value of the image in the file name to make it more clear for processing.

4. Who to send the email to?

    (a) This does not sound like an issue that should be brought up at all; however, was a big issue if we were creating a statically hosted site. Similarly, with the previous challenge of determining what file pertains to which question, we thought of putting the email address in the file name. This of course will not work due to naming conventions on objects uploaded to AWS S3 buckets. A typical email address can allow for many more special characters than what S3 can support. We also contemplated uploading two files, one is the original image file and the second is a supporting `.txt` file that only contains the user's email address. This proposal was immediately shot down as we did not want to increase overhead and waste resources.

    (b) The solution to this issue was to add user-defined metadata to every image uploaded to our S3 bucket. User-defined metadata on S3 objects does not have the same restrictions as the filename. Our front-end page has Javascript code that will take the email address from the text field and inject the email into our user-defined metadata. With this, our Lambda function can get the email address associated with each image upload without increasing overhead or wasting more resources.

5. How to fetch results if is not instantly available?

    (a) Now that we know whom to email the results to, we face a new issue. As discussed earlier, we have two main test cases, a new image or a duplicate image upload. The duplicate image upload was easy to deal with as the results were immediately available to reference and send to the user. The main issue is dealing with the new image uploads, as the results are not immediately available. We have contemplated creating a fourth Lambda function, but then the Lambda function would have no clue which user to email if we have to delete the photo.

    (b) Ideally, we wanted to create something similar to an event system using semaphores (similar to Operating Systems concepts) to put the main Lambda function as it is in a deadlock state and have another Lambda function raise the semaphore to take it out. This was problematic as our Lambda Function was only set up to trigger and get events on image uploads only. We resorted to a more rudimentary solution of putting our main Lambda function in a while loop with a 5-minute timeout. The while loop will query the DynamoDB table every two seconds.

        Of course, this solution is not the most optimal as it wastes resources; however, due to time constraints and limitations on the services, we pressed on with this solution. At the end of the day, the implementation we put in works.

6. How to get the word count?

   (a) The whole point of this project is to perform a word count on the text-filled image. Without it, the whole project has no meaning. We initially proposed using AWS Elastic MapReduce (EMR); however, the learning curve was too much.

   (b) Luckily, we had an assignment over AWS Glue PySpark, and we can perform the same task with it using much simpler syntax. The amount of time devoted to actually creating the word count on Glue PySpark was only about half an hour. Granted, AWS Glue is not perfect. Each time AWS Glue job is invoked, it takes a significant amount of time to spin up an instance, which can be up to 30 seconds in addition to the execution time.

5.2.2 Miscellaneous Challenges

As this is a group project and all group members are Master's students, finding a common time to meet up, allocating time to complete this project, and account access became issues. Trying to find a common time to meet up was probably the biggest issue because each member will have a different living and course schedule. The second issue of devoting and allocating time to complete this project was amplified as each member had to deal with many projects from other courses that were more challenging and required more time. Lastly, account access was an issue, because only one person can really access the account unless the password is to be shared. AWS does not have a real option for a collaborative session, and the workaround is making IAM accounts; however, the time to set it up was too much for our liking.

The real solution was to have a few meetings that had long sessions each and have one member do most of the coding and integration, while the others supported by researching questions the coder has. While distributing the work sounded like a great idea, since we were using many different AWS services, it can end up with more work, so delegating the coding to one person was the best option.

6 Lessons Learned

6.1 What have we expected to learn?

Because this is a half-semester project, we cannot ultimately set and propose a huge project goal and learn all the ins and outs of any cloud service. Furthermore, we did not want to propose a fancy project knowing that we would not complete it. Rather, we wanted to put out a more realistic project, so that we can get a better understanding of the different services offered by different services and how to integrate them together to achieve our goals and bring us some skillsets for the future.

6.2 What we actually learned?

Due to our easier project topic, we were naive into thinking that the services we had in mind would perform exactly what we had imagined. When we had to deal with actually using these services, we were faced with several limitations that required us to be very creative with how we can use these services to overcome the limitations. Sometimes, we have to research online about different ways to approach the issue, and other times it required us to bring up possible solutions and experiment with them. Furthermore, we learned that sometimes there is more than one service that can provide similar results, but requires much less of a learning curve. For example, instead of using Elastic Map Reduce for our word count, we elected to use AWS Glue and PySpark to perform the same task.

7 Conclusion

In this project, we successfully created a serverless function using Amazon Web Services that allows users to upload their text-filled images and receive the word count results through email. This project gave our group a great challenge to integrate different solutions and provide creative workarounds for certain limitations. Despite not being to implement everything we wanted and having extra nice to have features, we overcame these issues and addressed them. While this project serves its main purpose, there are many places that can benefit from improvements or changes.