

## TagStore

### A Modern Image Storage on the Cloud



#### Team Members

**Kishore Babu Sab (29604885)**

**Sheridan Arvind Filomeno (30356660)**

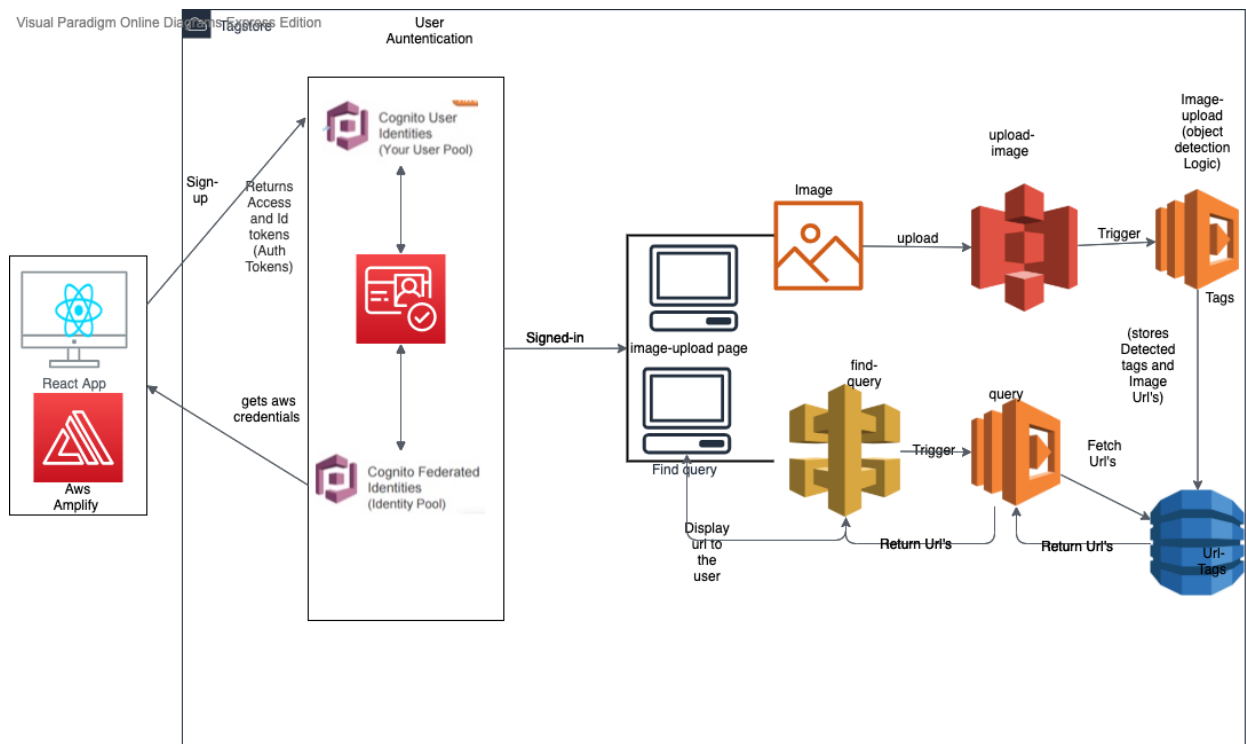
**Prathamesh Inamdar (30227704)**

**Bo Wo (29360390)**

## Introduction

This Report aims to discussing the process of building a cloud application, that allows users to store the images and retrieve the images based on the auto-generated tags. The serverless application built on AWS, allows the client to upload their images to public cloud storage. After the image upload, tags of the image with objects detected in it are stored. Further clients can query images based on tag of the object; The Cloud application will return List of Image URL's that include the queried tags.

## system design and architecture



Visual Paradigm Online Diagrams Express Edition

Figure 1: AWS System Architecture

Our Application can be accessed through following link: <https://tagstore.netlify.app/>

### **The System Architecture:**

For the Tagstore cloud application, we have used 2 Lambda functions" upload-image" and "query". The former one does the object detection and latter one helps to search the tags (Findquery). Lambda functions form the Core-business logic. There are S3 buckets," imguploads3-bucket" for user to upload their images." objectdetection-yolo-files" bucket was used to store yolo-configuration files, which will be used in the object-detection." opencv-layers-lambda" bucket was used to deploy open-cv, as layers into the "upload-image" lambda function. Open-cv is a python package, which also is used for object detection. One Dynamo-DB table named "Url-Tags" is created and all the detected object tags along with the image URL's will be stored in this table. An API gateway Named "find-query" was created in order to take the request from the user, this api gateway will trigger the "query" lambda function and will return the Image Url's to the user. A user-pool named "Tagstore-user-pool" and "tagstore-identity-pool" was created to allow only authenticated users to use the application. React Native was used to build the user interface's Amplify was along with React to Authenticate the users.

AWS Amplify is Developmental platform that helps in building scalable and secure Web applications. It makes it easier for the developers to authenticate users, securely store data and user metadata, authorize selective access to data, integrate machine learning, analyze application metrics, and execute server-side code.

### **Business Logic**

The user will be able to upload to the images to the S3 bucket. And users can also see the tags present and the URL's of the image tags. After the user is successfully signed in, he can perform two things. Image upload and find query. In image upload. The user will be able to upload an image to the S3 bucket. And second one will be he will be able to search for tags.

### **User interface:**

The user interface for the app was developed using React. React is a JavaScript library and web framework used for developing high fidelity user-interfaces. Our react app is structured to work with AWS by using AWS Amplify to communicate with the various aspects of AWS needed to make the app functional such as AWS Cognito, AWS API gateway and AWS S3. To run AWS amplify first we needed to install it using npm install aws-amplify, then communication was facilitated using a config.js file containing for S3 REGION and BUCKET name, for API gateway REGION and URL and Cognito REGION, USER\_POOL\_ID, APP\_CLIENT\_ID and IDENTITY\_POOL\_ID. To use Amplify it needs to be imported using Amplify from aws-amplify. After which we used Amplify. Configure to configure the different modules involved in our app such as Auth, Storage, API.

```

export default {
  MAX_ATTACHMENT_SIZE: 5000000,
  s3: {
    REGION: "us-east-1",
    BUCKET: "tagstoreimgupload"
  },
  apiGateway: {
    REGION: "us-east-1",
    URL: "https://p1o4zvmoe3.execute-api.us-east-1.amazonaws.com/development"
  },
  cognito: {
    REGION: "us-east-1",
    USER_POOL_ID: "us-east-1_aJ0Mzloxm",
    APP_CLIENT_ID: "488ipu8jmnst22qn6arn0au8el",
    IDENTITY_POOL_ID: "us-east-1:783e3ab4-5d96-4070-8733-269a4150a2bc"
  }
};

```

Figure 2. Amplify. Configure

After the initial integration with Amplify, the login.js and signup.js react pages were created. These pages handle the authentication of the user by communicating with Cognito using Auth from Amplify, for signup we use Auth.signUp(params) and for login Auth.signIn(params) is used (params refers to email, password, first name etc.). After the user has successfully authenticated, they are redirected to the homepage, which is handled using Routes.js in the react App.

The Image upload page communicates with S3 to allow the user to upload images and store them on S3, which is done using Amplify. A Js file was created in the lib folder to handle this communication, which is done using Storage from Amplify, an async function called s3Upload(file) which accepts the image as the parameter uses Storage.put(filename, file) to upload the image on S3 after which the key is returned. s3Upload is called in the ImageUpload.js page.

The search functionality of the app is implemented by calling our find query API created using AWS API gateway. Which is done using API from Amplify, this allows us to call the GET method that will return the list of links for the searched object. API.get( API name, path, { query string parameters: { name: searchText}, }); query string parameters refers to the search text that will be used in the GET request. The response is returned in a JSON array this array is then used to display the links below the search bar, using .map to parse the array for the individual links. After which S3Image from 'aws-amplify-react', we use S3Image to display the images below the search bar. Using <S3Image imgKey= {key}> key is retrieved by splitting the link received from the API.

### Image-upload

An s3 bucket named “imguploads3-bucket”, is created. And all the uploaded images will be stored in this bucket. A lambda function named “upload-image” is created and will act as the core- business logic function for the application. Whenever the user uploads an image to imguploads3-bucket, image-upload Lambda function will be triggered. This lambda function will detect all the tags in the uploaded image, and stores. Tags of the uploaded image along with the URL of that image.

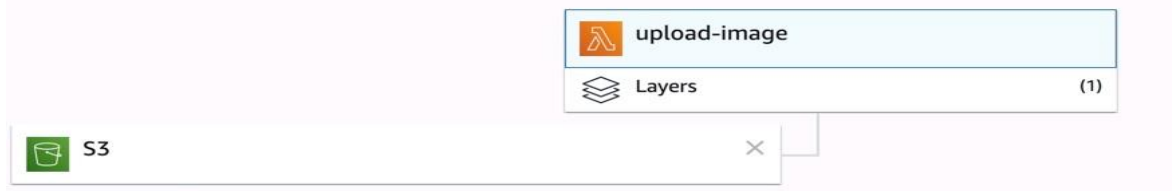


Figure 2: Image Upload and Object detection.

The S3 bucket is set as the Trigger for the Lambda Function. in order for “Upload-image” Lambda function to access the images uploaded in the s3 bucket. an IAM role named “UploadImageRole” was created.

### Object Detection

Object detection can be done in many ways. Or this assignment. We have yolo-opencv-object-detection. In order to support yolov3-object detection we need Yolo Configuration files. Which are “yolov3.cfg”, “yolov3.weights”, “coco.names” as the training set. These files are stored in the S3 bucket named “objectdetection-yolo-files”, the lambda function “Upload-image”, downloads this file from s3 bucket and uses it in the function to train the model. (**s3. Download\_file()**) method was used to download files from s3 bucket to the lambda function. OpenCV is a library of programming functions mainly aimed at real-time computer vision. Unlike yolo, OpenCV-python package. Which by default is not available in lambda. In order to import OpenCV package, we have used to through layers. An Ec2 instance was sinned-up, and OpenCV package was downloaded and zipped to a package. And this package was installed on the layers for lambda function to use the openCV package. Once the necessary files are imported or downloaded, object-detection logic is being written,

after the object detection. The detected objects called as tags, along with the image URL of the uploaded image will be stored in the dynamo-DB named “Url-Tags”. a sample of dynamo-DB storage can be seen in figure 3.

Scan: [Table] Url-Tags: url ▾ Viewing 1 to 22 items

url ⓘ	data
<input type="checkbox"/> <a href="https://imguploads3-bucket.s3.amazonaws.com/000000007454.jpg">https://imguploads3-bucket.s3.amazonaws.com/000000007454.jpg</a>	person,skateboard
<input type="checkbox"/> <a href="https://imguploads3-bucket.s3.amazonaws.com/000000023401.jpg">https://imguploads3-bucket.s3.amazonaws.com/000000023401.jpg</a>	orange
<input type="checkbox"/> <a href="https://imguploads3-bucket.s3.amazonaws.com/000000026923.jpg">https://imguploads3-bucket.s3.amazonaws.com/000000026923.jpg</a>	aeroplane,aeroplane
<input type="checkbox"/> <a href="https://imguploads3-bucket.s3.amazonaws.com/000000028124.jpg">https://imguploads3-bucket.s3.amazonaws.com/000000028124.jpg</a>	bus,car,bus
<input type="checkbox"/> <a href="https://imguploads3-bucket.s3.amazonaws.com/000000028465.jpg">https://imguploads3-bucket.s3.amazonaws.com/000000028465.jpg</a>	train
<input type="checkbox"/> <a href="https://imguploads3-bucket.s3.amazonaws.com/000000031026.jpg">https://imguploads3-bucket.s3.amazonaws.com/000000031026.jpg</a>	bird

Figure 3: “Url-Tags” DB Table displaying Image URL’s and Object tags

### Find Query

After the user has successfully uploaded the image, the user will be able to search or query for the tags. That means that if user search for “person” in find query page, the Application will return all the Image URL” s which has person in it.if the user gives “person, elephant”, all the images which have person and elephant in combination, those Image URL’s will appear on the results page.

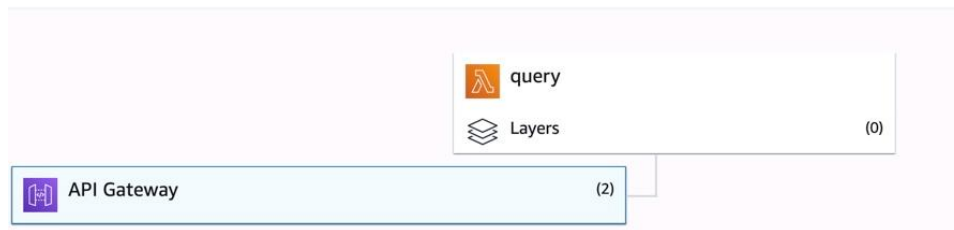


Figure 4: Find query

an API Gateway with a RESTful API was created that allow users to submit their requests, via GET or POST requests. Our application can send a list of tags via certain GET parameters through the requested URL, for example: <https://p1o4zvmoe3.execute-api.us-east-1.amazonaws.com/Develop?tag1=person&tag2=car>. Or they can send a post request, in json format like {"tags": ["person", "desk", ..."cat"]}

That’s why there are Two API gateway’s one for GET and one for POST, as soon as the user. As soon as the user gives the tags, the API gateway will fetch those tags, In the get format as the user only enters the required tags, after these tags are fetched. Used using API. get () in React. Will trigger the “query” lambda function. This lambda function will check for the tags, scan the dynamo-DB table, and retrieves all the Image URL’s which has tags mentioned by the user. The lambda function returns the Image URL’s to the API gateway, API gateway will indeed return the URL’s to the User interface results page. The logic used in the Find-query lambda function can be seen in figure 5.

Table. Scan () scans all the elements in the table. Taglist contains all the tags user wants to find. All tags are checked against the table, if the tag matches, that URL will be picked and appended to URL list and send to the API gateway.

```
scan = table.scan()
url_list=[]

for i in scan['Items']:
    if all(element in i['data'] for element in taglist):
        url_list.append(i['url'])
```

1. What are the updates you will perform to your application if you have users from all over the world?

In case we want to update the application for the user from all over the world, it is necessary to register a domain name for the website so that the users can access easily. Also, we need to think about the amount of traffic whether the hosting is scalable to handle as many users choose to use this application. Besides, web container needs to be considered because it can process multiple threads, especially for the dynamic website hosting. In order to design a great website for users we also need to build up our strong backend service. For the technical aspect, all the potential security issues could compromise with website integration.

2. What sort of design changes you will make to reduce chance of failures in your application?

There are many reasons cause the website failure such as slow speed transmission rate. For our website, it could crash the application if users upload some large size of images. To fix this issue, we can use some special tools to compress images first and then upload them to the website. When users search the image with multiple tags, the numerous incoming and outgoing links will be processed at the same time could cause the application to fail.

3. What are the design changes you will make to increase the performance of your applications in terms of response time, query handling, and loading image?

Improving server response time, we can try to optimize our database which can retrieve data as efficiently as possible. Also, rewrite the query function to return specific value instead of using loops is also necessary to speed up the loading time. In our UI design, we used React to implement all the functionality. However, react isn't fast by default, the useless renders of many components could make the application running slow. Removing unnecessary code that has no effect on performance can improve the response time.

### **Real World Application**

Object detection can be used in lot of places. The Tagstore application which will be used to upload the images and get the image URL's of tags the user wants. We can modify this application little bit to detect objects in motion which would help in the following uses. Based on the use, we can modify the application according to the user needs. Firstly, it can be used for **Optical character recognition (OCR)**. OCR is the mechanical or electronic conversion of images of typed, handwritten or printed text into machine-encoded text. One of the best examples of why you need object detection is for **autonomous driving/Self Driving**. Object detection can use to **Tracking objects**. Object detection system is also used in tracking the objects, for example tracking a ball during a football match, tracking movement of a cricket bat, tracking a person in a video. **Activity recognition** aims to recognize the actions and goals of one or more agents from a series of observations on the agent's actions and the environmental conditions

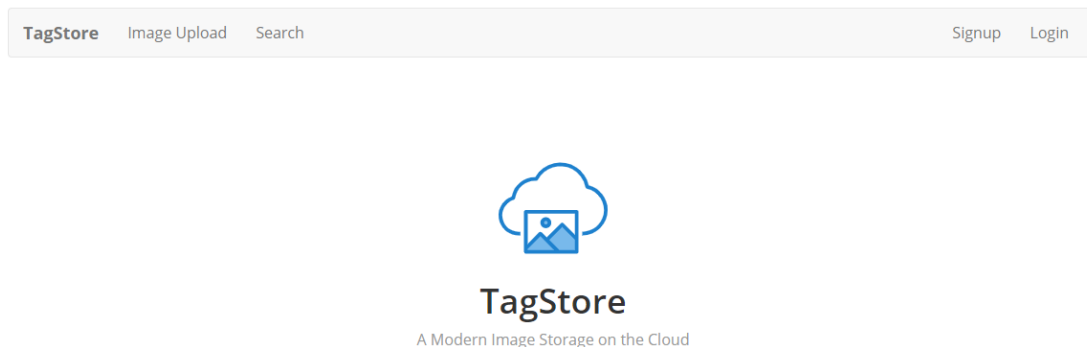
### TagStore User Manual

TagStore is web-based application can be used to upload images and retrieve distinguish images with Tag. TagStore can be accessed by clicking on following link.

TagStore: <https://tagstore.netlify.app/>

#### 1. Home Page:

When you click on above link you will be redirected to Home page of our TagStore Application as follows.




#### 2. Signup:

If you are new to our system, then Click on Signup from upper bar. You will be redirected to our signup page as follows:

Filled in your valid Email address. Remember verification code will be sent to mentioned email. Password format:

- Must be more than 8 characters.





Email

First Name

Last Name

Password

Confirm Password

**Signup**

- Must have at least one number
- Must have at least one upper case character.
- Must have at least one Special character.


After Successful completion of signup form click on Signup button. After this step you will be redirected to enter verification code which we have sent you on your email.

Confirmation Code

Please check your email for the code.

**Verify**

### 3. Login:



Email

Password

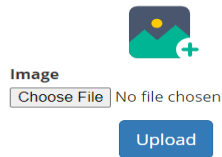
**Login**

After successful signup, Login with your Correct credentials. Type verified email and password then click on **Login** button.

If you forgot your credentials, then Click on **Signup** button from top and signup using your valid Email address as shown in step 2.

### 4. Image Upload:

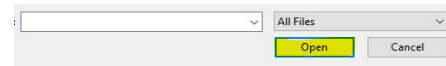
TagStore Image Upload Search Logout



- Click on **Choose File** button to choose image from Your system.
- Navigate to your image.
- Click on your image and click on **Open** button as highlighted in beside image.

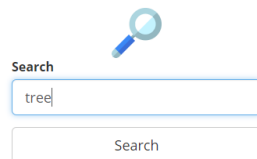


This will successfully upload your image to our cloud.



### 5. Search.

TagStore Image Upload Search Logout



- To search, enter Tag such as person, elephant etc.
- Click on **Search** button to get results.

### Team Contribution

s.no	Name	Contribution	Contribution percentage
1	Sheridan Arvind filmeno Mathew bries gomes (srie0008@student.monash.edu ) 30356660	<b>AWS Cognito, User Interface And integration.</b> He has worked on User interface and AWS amplify to generate the front end. Also wrote about User Interface in the report.	25%
2	Prathamesh Inamdar (pina0001@student.monash.edu) 30227704	<b>Aws Cognito and User Interface.</b> Worked the User Interface. He also wrote the user manual for the report. Tried Federated authentication.	25%

3	Kishore babu sab ( <a href="mailto:kbab0003@student.monash.edu">kbab0003@student.monash.edu</a> ) 29604885	<b>Image Upload and integration.</b> Object detection and tags storage were implemented. System design and Architecture was written in report.	25%
4	Bo Wo ( <a href="mailto:bwuu0013@student.monash.edu">bwuu0013@student.monash.edu</a> ) 29360390	<b>Find query.</b> API gateway was taken care by her. She also wrote the Lambda logic for find query. She also wrote the questions which were supposed to be answered in the Report.	25%

### References

1. <https://reactjs.org/tutorial/tutorial.html>
2. <https://aws-amplify.github.io/docs/js/tutorials/building-react-native-apps/>
3. <https://docs.aws.amazon.com/amplify/>
4. <https://docs.amplify.aws/ui/q/framework/react>
5. <https://docs.amplify.aws/ui-legacy/storage/s3-image/q/framework/react>
6. <https://reactjs.org/docs/getting-started.html>
7. <https://docs.aws.amazon.com/apigateway/>
8. <https://docs.aws.amazon.com/lambda/>
9. <https://docs.aws.amazon.com/db>

GITHUB source code link for the project:

There are Two repositories, one for the User interface and hosting the application using Netlify. other one is the AWS backend Code.

<https://github.com/sheridanzzz/serverless-stack-client.git> (FRONTEND)

<https://github.com/sheridanzzz/serverless-stack-backend.git> (BACKEND)