Label Hub

*A cloud-based platform and market for image labelling

Tsai-Chen Hsieh
Graduate School of Engineering and Applied Science
Columbia University
New York City, United States
th2990@columbia.edu

Zehua Chen
Graduate School of Engineering and Applied Science
Columbia University
New York City, United States
zc2616@columbia.edu

Fatima Dantsoho
Graduate School of Engineering and Applied Science
Columbia University
New York City, United States
fd2508@columbia.edu

Alix F. Leon
Graduate School of Arts and Sciences
Columbia University
New York City, United States
afl2124@columbia.edu

***Abstract*—Machine learning and data-intensive methods have seen a boom in recent years. In particular, computer vision algorithms have experienced a surge in terms of the number of companies applying them to their products. From training models to automate vehicles to learning to mimic movements using graphical information, there is an increased need for properly labeled data across various industries. Consequently, companies all around the world invest heavily in the process of image recollection; however, they are not guaranteed quality labels to train models and organize their data. Furthermore, this process is costly, discouraging development and research when funds of scarce. This trend only increases with time as new techniques require significantly more data. In this project, we present Label Hub, a cloud-based distribution platform for labeling images that can reduce costs for consumers, such as companies and researchers, while at the same time incentivizing producers to upload quality data by facilitating the process of licensing these images in exchange for monetary compensation.**

***Index Terms*—distribution, labeling, images, captions, cloud computing.**

I. Introduction

Label Hub is a platform designed to license and distribute user labeled images in large quantities. This platform makes it easy to obtain good quality images with high sensitivity, that is, with a high rate of true labels assigned to each image, potentially reducing the end costs for consumers by minimizing the process of applying denoising techniques and manually labeling data.

Producers have the option to upload as many images as they want. One of the main challenges we faced when building a system of this scale is ensuring that the flow of information is not blocked or interrupted by neither consumers nor producers. For this particular reason, we decided to migrate as much of the computations as possible to our elected cloud computing service, in this case, Amazon Web Services (AWS). By utilizing AWS as our main cloud service, we can ensure up to a 99.99% up-time for all our services, with high levels of throughput even during moments of high user activity and demand. Additionally, it is a cost-effective resource due to the option to horizontally scale our services on demand.

Section II describes the project overview, section IV goes over the stack implementation and the different technologies and functions used for the project, section V goes over potential challenges, in section VI we go over the results of our experimentation with the project's stack and used services, and in section VII we discuss further work that can be implemented to improve the accuracy of the labeled images.

Link to the slides (Need Lionmail to access)
Link to the video
Link to Github

II. Project Overview

More pictures and images are being produced now than ever before due to the adoption of different mediums at an affordable cost to the public. Although it is hard to determine exactly how many are taken on a daily basis now, it is safe to assume that the number of images produced has exponentially with the increased adoption of technology and social media during the last few years.

Companies, in particular those who deal with vast amounts of data, often encounter noisy data for real-world computer vision projects. In order to obtain quality data, they often go through the process of outsourcing the labeling task to other individuals or companies; nonetheless, the process tends to be time-consuming and expensive. Label Hub aims to amend this issue by providing an easy-to-use platform for both producers and consumers.

Label Hub provides a direct interface for producers to obtain high-quality labeled images at a fraction of the cost of outsourcing the process of recollection and manual labeling. This platform removes the time-consuming process of reaching out to other individuals to complete this process and instead provides a service where producers can easily license their images to consumers. The latter is then able to download a

data set of the size of their choosing and use the images for any purpose, according to the license agreements.

This application is a platform where users can upload photos from their devices. They will then be prompted to add descriptive labels. This aggregated information will be called image metadata and will be stored in a database together with the image.

This platform will serve as a pipeline for third-party businesses and organizations looking to gather data for machine learning models in exchange for payment for licensing. After the transaction is completed, users will be compensated for their data, and we will receive a small commission.

III. Related Work

There exist services like Amazon Mechanical Turk that connect companies and researchers to individuals looking for people to complete Human Intelligence Tasks (HITs), such as labeling images. These services are often forum-like and require consumers to post requests to a platform, to then wait for their request to be picked up by another company or individual. To our best knowledge, Label Hub is the first platform directly connecting producers to consumers offering licensed labeled images in mass.

IV. Architecture and Implementation

A. DevOps

Label hub uses many components from AWS as its main computing engine. Furthermore, for this project, we took an unconventional approach to the development process. Instead of using the graphical user interface (GUI) provided by AWS to define our resources, we decided to take advantage of Amazon's Cloud Development Kit (CDK). This framework, much like AWS' CloudFormation templates, allows us to define resources in a programmatic way.

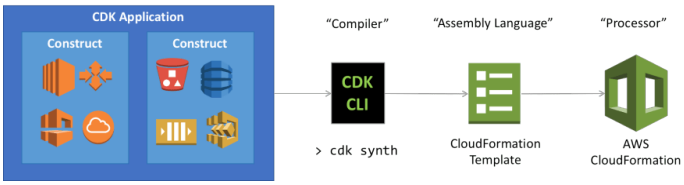

Fig. 1. The workflow of CDK. Source: dev.to

For each CDK deployment, there is App, and each App might have multiple Stacks, similar to a stack that we deploy using CloudFormation, Finally, we have Construct, which stands for the individual resources in AWS. Essentially, CDK generates Cloudformation templates from code with *cdk synth* and deploys the stacks with *cdk synth*.

The advantage of using CDK is that every resource is defined clearly, and can be deployed easily. When we pull new commits, we can clearly see what resources have been moved. If everyone is working on the same AWS account, team members might delete each other's resources, and without any backup, the progress might be lost. CDK provides us an easy way to examine the resources deployed and deploy all the resources, simply type *cdk deploy*, and updates to the resources can be done independently so that we only need to redeploy the resources that are changed.

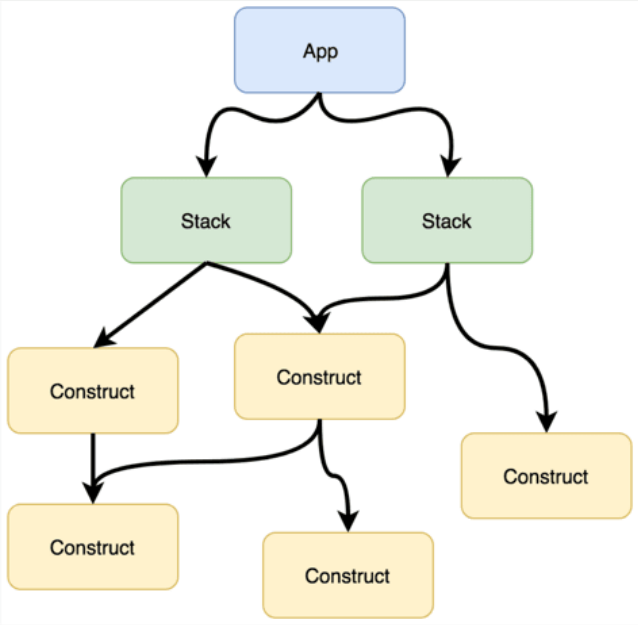

Fig. 2. The CDK code structure. Source: dev.to

Another advantage of CDK is the possibility to define the resources clearly. Some of the team members have experienced using CloudFormation, and while it does similar things in terms of deploying the resources, filling out the details can be annoying and prone to error. CDK fixes that by automatically filling out the details for us, while letting us write significantly less code. It is used via a wide variety of programming languages, so we can all apply the programming logic to it. For our development, we use Typescript on CDK since it is the most supported language.

B. Resources Used

Our architecture is based on serverless components, taking advantage of the services and operations made available by AWS. This gives us the ability to expand our resources and integrate new ones on demand. By utilizing preexisting services and platforms, we minimize the number of bugs that can be unintentionally introduced into our implementation of certain components; additionally, we have the option to scale computational resources in case of increased demand. Next, we will describe a brief overview of the resources used in this project.

- AWS S3 buckets are the main permanent store. Since our system focuses on distributing labeled data amongst many consumers, it is of utmost importance to have a reliable permanent store. S3 achieves high throughput of data over multiple connections, thus maximizing the

number of pictures that can be downloaded through our pipeline on multiple client connections. This service is also used as a hosting platform for both our front-end and back-end stack, meaning that we can deploy changes to the web immediately.

- Open-search instances are used in conjunction with Amazon S3 services to store metadata information regarding uploaded images from producers and consumer projects that can be quickly indexed and returned with minimal latency.
- DynamoDB is used as our logical schema store. Here we store information regarding consumers and consumers such as billing information, and income details, among other metrics used for quality-of-life and user experience purposes.
- API Gateway is used as our main service to connect between the front end and the back end. It helps us to manage the data sent from the front end and route them to the correct Lambda.
- We use Lambda to take care of all the computational work as well as connecting between different services. It is serverless, so we do not need to think about scaling issues, making it a great option for our project.

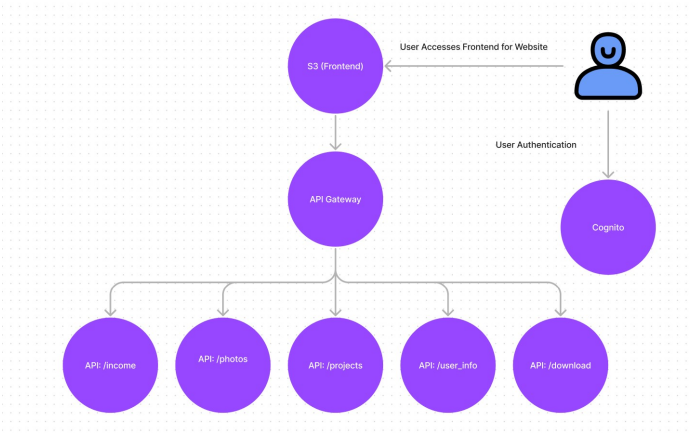

Fig. 3. Front-end and API Gateway

C. Producer and Consumer

For Label Hub, we have two roles: producer and consumer. Producers create photos that can be uploaded to the web, and the consumers can buy images. For producers, we assume that they are the people that want to upload the photos and sell them to people. We intentionally keep the producers from knowing their consumers, and the same can be said from the consumer side, We did this to ensure data confidentiality for both parties. According to the roles, there would be different functionalities provided to them.

D. Front end

Our front end is built using Gatsby, React, and TypeScript. At the built time, Gatsby generates static HTML, JavaScript, and CSS which would be stored in S3. Then at run-time, the browser would load the static HTML, and React would "hydrate" the page. React's hydration mechanism requires the static HTML and the first render of our app to produce the same result. For a site that has dynamic data like ours, this poses many problems.

One problem is the environment differences between Node.js, where the static markups are generated, and browsers, where the user uses the app. Many APIs are not available in Node.js. Bootstrap's JavaScript library would use Browser API upon import, which would break HTML generation in our case. So to work around this, we lazily import Bootstrap in an effect hook to ensure that Bootstrap code is only run in the browser.

Another issue is data fetching. Since authentication only happens in the browser, not in build time, we would only want to fetch data in the browser. To handle this we use a third-party library named "swr" which only runs the fetch function in the browser. An additional benefit of "swr" is that it caches already-fetched data so that when the user clicks the back button, the user would not see a blank screen. At the same time, "swr" triggers another fetch to update the interface with the latest data.

We use Cognito for authentication. Since Cognito does not allow non-HTTPS URLs to be used as callbacks for its hosted authentication interface. Our front end is stored in S3 but served over a cloud front distribution.

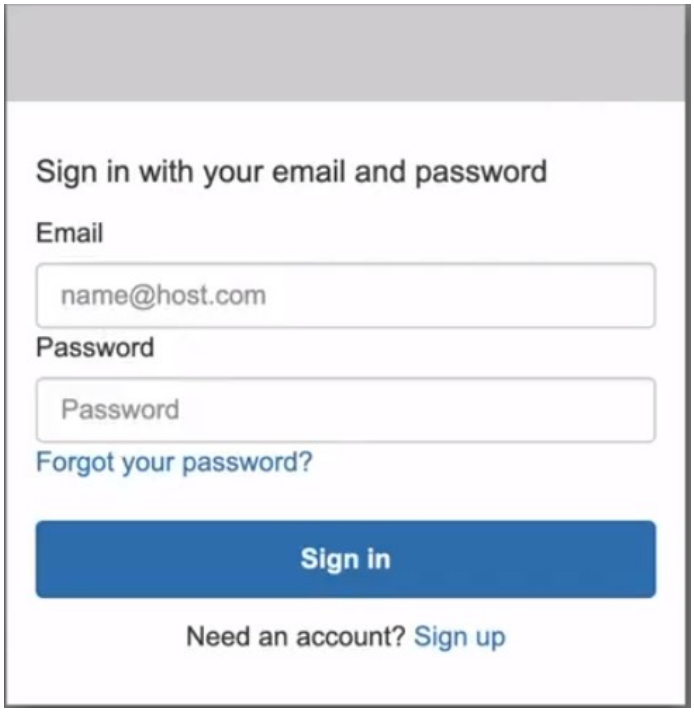

Fig. 4. Login page

When we go to the producer dashboard, it lists all of the photos that the producer has uploaded, as well as the related information. To upload an image, we click "Upload Image".

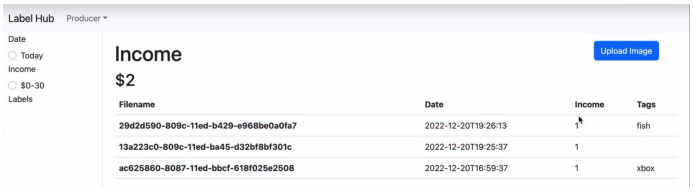

Fig. 5. Producer dashboard

At the image editor, we can upload a picture and indicate the labels. After we are done, click ”Save”, and the stored picture would appear in the producer dashboard.

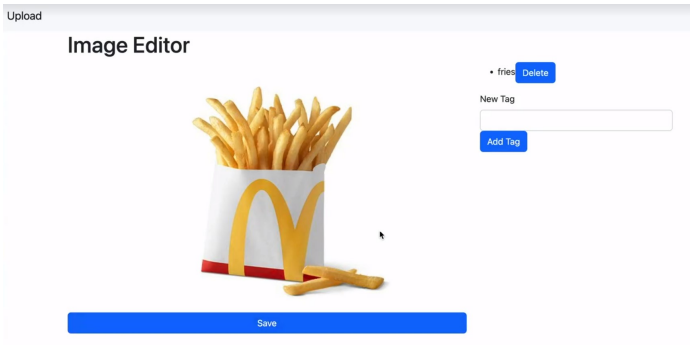

Fig. 6. Image editor

If we switch the tab, we can see the consumer dashboard. It contains all the images that the producer bought. Click ”Buy” to go to the Image Marketplace. Click ”Download” to download the image in binary format.

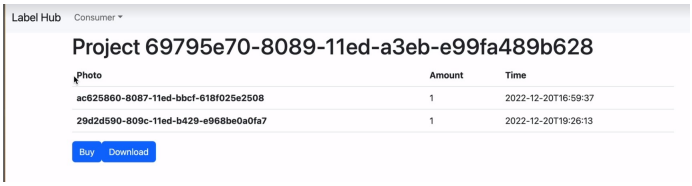

Fig. 7. Consumer dashboard

In the Image Marketplace, we search for the labels that we want. Then, click the ”Buy” button to buy the photo.

E. Storage instances

photos (S3): The bucket stores images uploaded by the producer. The key is the filename.

downloads (S3): The bucket stores images ready to be downloaded by the user. The images are grouped by projects.

producer (Open-search): The instance has the image file name as the document's id, so each uploaded photo has one document. The fields are: objectKey(filename), bucket name, created timestamp, year, month, day, producerID, price, labels.

consumer (Open-search): This instance stores information regarding the consumers transactions when purchasing a photo. Every (consumerID, filename) pair has a unique document as id. The fields are: (consumerID, filename) unique

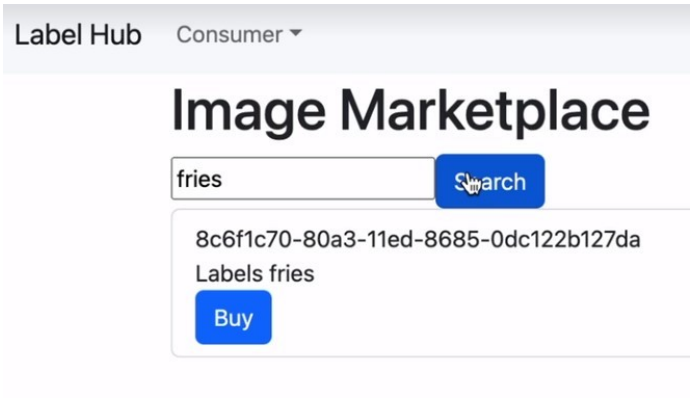

Fig. 8. Image Marketplace

name, file name, bucket, created timestamp, producerID, price, consumerID, projectID, labels

userinfo (DynamoDB): The fields are: user id(primary key), first name, last name, title, email, about me.

F. Lambda and API Methods

The back-end consists of the following methods:

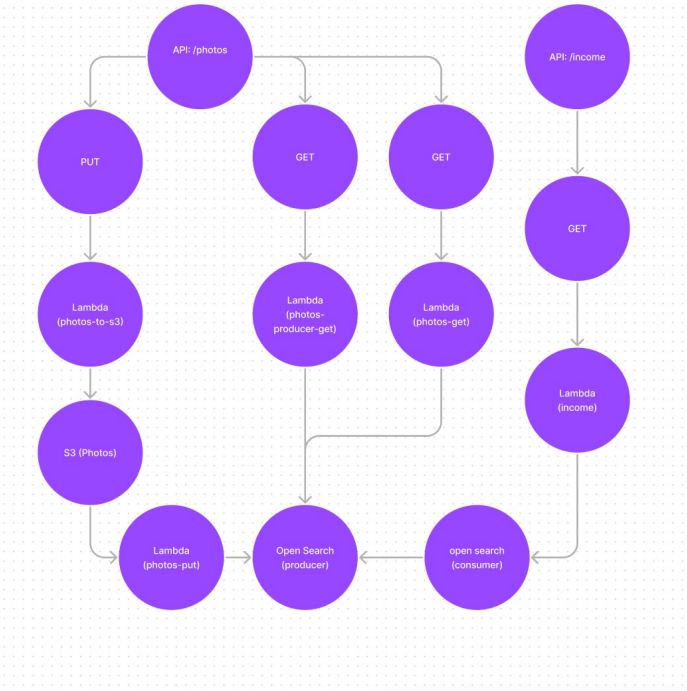

Fig. 9. Photo and Income method

GET income: The method would take the producerID as input and return the total amount of income by querying the consumer Open-search. It adds the price that appears in each document that has the requested producerID.

GET photos-get: The method would take a label as input and return the images that contain the label. We search through all possible images that have the searched label and return the photos that contain the label.

GET photos-producer-get: The method would take the producerID and filter information as input and return the information of all pictures produced by the producer. This is used when constructing the producer dashboard. The information includes: filename, time stored, the price of the picture, labels for the picture.

PUT photos-put: Once the photo is stored in the photos s3 bucket, the lambda is triggered to store the labels and other metadata to producer Open-search, which includes: objectKey(filename), bucket name, created timestamp, year, month, day, producerID, price, labels. The id of each document is the file name of the image, so each image uploaded by the producer can have a unique document.

PUT photos-to-s3: The method would take the producerID, labels, and photo to store them in the photos s3 bucket.

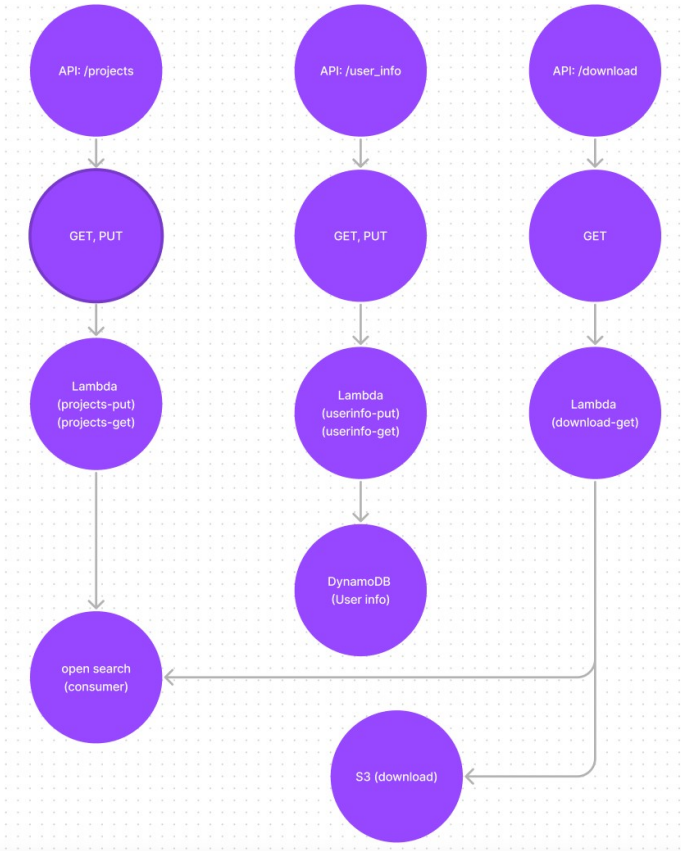

Fig. 10. Projects, Userinfo and Download

GET project-get: The method would take the consumerID and projectID as input and return all the photos that are labeled to the specific project by the consumer. It will query the consumer Open-search to get the result.

PUT project-put: The method would take the consumerID, projectID and objectKey(filename) as input, and store the information to consumer Open-search. The id is a combination of the consumerID and filename, so every (consumerID, filename) pair has a unique document. We can also view this as an action to record each transaction.

GET userinfo-get: The method would take userID as input, and return information about the user. It gets information from the userinfo DynamoDB.

PUT userinfo-put: The method would take the following fields as input: user id, first name, last name, title, email, about me. It stores the information in the userinfo DynamoDB.

GET download-get: The method takes projectID and consumerID as input and returns the binary file that allows users to download.

V. Challenges

One of the main challenges we faced when designing this project was integrating the resources using CDK. Although CDK gives full control to the developer over the creation and deployment of resources, there is a steep learning curve, as no implementation between two different Amazon services is equivalent, and furthermore, they do not share the same parameters in many cases. This development cost, however, is offset by the stability in production due to the granular control imposed over the aforementioned services.

VI. Results

Despite the fact that we have not started testing on large samples of users, after several local experimentation sessions over the project, we concluded that we can efficiently distribute and license images between producers and consumers, whilst accounting for any extra traffic without any major complications. Due to some missing components that were mentioned in this paper, this product is not ready to roll out to the public; however, our stack counts with robust components designed to be easily maintained. Moreover, our stack was developed with future work in mind, thus making the process of integrating new services a simple job.

VII. Further Work

Given the limited time, we are not able to completely realize our vision for the project. Here are a few features that we could work on in the future:

- During the planning phase, we incorporated the idea of projects, where each bought image by the consumer can be grouped together, and we also implemented the structure that supports this feature to the back end. In the future, we hope to bring the feature to the front end so that users can group the images for better management.
- Stripe API, an API for online payments, is a great way to make the project more towards real-life usage. At the current stage, we record the transactions in Open-search, and no real-life monetary transactions are involved. By adding Stripe API to the system, we allow the consumers to pay by cash, and the producers can also receive real money.
- During our investigation, we found out that providing bounding boxes on photos can be useful for machine learning projects. It provides a more precise way to let machines identify the objects in the picture, and can potentially produce better results.

- We are aware that in the current state of our design, it lacks a way for consumers to verify the quality of the labeling. It causes a disadvantage for the consumer since they do not know the quality of the labeling beforehand. One solution is to allow consumers to make manual verification before buying. Another one is to leave ratings or reviews to the producer. Even though the producer is anonymous to the consumer, the reviews can still stay for future consumers to check. In that way, we make sure that the data consumers use to train is always guaranteed to have high quality.
- In the future, we intend to implement machine learning models, also using AWS, to further increase the accuracy of the labeled images in our platform. Images uploaded via our system will be used in a feedback loop to constantly retrain and improve the classifications for the labeled images, therefore minimizing potentially noisy subsets in our data.
- In the future, we intend to deploy the front end over AWS Amplify or App Runner to take advantage of Server Side Rendering (SSR). Our current approach of Server Side Generation (SSG) has a major drawback that when user access dynamic routes like /consumer/project/id, they would see "Not Found" for a second and then see the content. Another issue with SSG is when the user first sees the interface, it would not have meaningful data. With SSR, would fetch all the data on the server side, and then render the page. This would ensure the user to see meaningful data on the first render.
- Currently, our licensing system consists of simply purchasing images hosted on our platform; nonetheless, this may prove inconvenient to some producers, thus discouraging them to further upload images to our platform. To address this issue, we are currently discussing ways to develop a robust licensing system to give consumers and producers granular control over their resources, therefore fostering a sense of control over the uploaded and downloaded images.
- We should also adopt CLEAN architecture. Currently, our lambda code is impossible to test locally, and as a result, we could not write continuous integration tests for them. By adopting CLEAN architecture, we would be able to resolve this issue.