Allert

Conor Clerkin (18773059) & David Moore (18722869)

1. Overview

Allert is an application that allows users to scan product ingredient lists and flags any products that may cause that user a specific allergic reaction. The app is cross-platform and is hosted on AWS with serverless infrastructure. The main back-end component of the application is an OCR function which utilises Tesseract and OpenCV.

2. Motivation

We chose this application and this project because of the technologies that would be involved in developing it. We each had an interest in computer vision, cross-platform app development and cloud infrastructure. This project allowed us a chance to explore these interests and develop our own skills in these areas. Computer vision in particular is an emerging technology and one which required extensive research and understanding.

We hoped the project would provide challenge and complexity that would force us to think critically and overcome obstacles. This would give us an experience comparable to working in industry where a project's progress is not linear but deals with challenge and change in a flexible way. Our aim was to develop this application in a way that would be similar to commercial software development in order to gain an insight into this workflow. We believed that a project like this would allow us the opportunity to simulate this environment to some extent. The choice to work with serverless cloud infrastructure was also driven by this desire to align with corporate software development practices. Furthermore, choosing to provision infrastructure using infrastructure as code tools was an industry standard practice that appealed to us while also providing visibility of the cloud side of the project.

Beyond the technical aspects of the application being appealing, we were motivated by the tangible nature of app development. We wanted to work on something that could provide us with immediate feedback and results. From very early in the development process we could evaluate the effectiveness of the app with regards to user experience and functionality, and we could then make decisions based on this feedback. We wanted a project that would be very end-product focused throughout its development cycle and building a mobile application gave us this opportunity and is another reason why we chose this project.

Finally, we wanted to develop an application that could help its users in some way. Initially we discussed ideas based around areas such as education and general health before eventually landing on allergen scanning. The vast majority of our ideas were based on some kind of user interaction leading to a benefit for that user. This benefit is one of the reasons we decided on Allert. We hoped that if we could visualise the app directly helping someone it would be a more enjoyable and rewarding project.

3. Research

As mentioned in section 2, one of the motivating factors behind choosing this project was the breadth of research and learning that would be necessary to implement it. This was an ongoing process throughout the development lifecycle. Before writing out our functional specification we did some broad research to gain an understanding of what the project would entail. We did this in a systematic fashion by first drafting our functional requirements, searching online for the general consensus on

how they should each be achieved, and then reading further into the methods proposed. Of course, the research process did not end here. Through each stage of development, we continued to learn. At each step we wanted to find the best possible solution for any problems that arose, and this required deepening our understanding of the technologies we were using. The following is a report of the general areas of research, what we learned and how it influenced our next steps.

3.1 Python Optical Character Recognition

We decided early on that we were going to write the back end in Python as it was a language that we have extensive experience with, has a large supportive online community and crucially has a high availability of libraries and frameworks. This led to our first research topic – "how does OCR work in Python". It was immediately apparent that the most effective and reliable solution would feature the OpenCV and Pytesseract libraries.

3.1.1 Image Reading and Manipulation

The very first step of reading an image in Python presented two options. Either OpenCV (cv2) or Pillow (PIL) could be used to read and process an image. They both performed the same functions (reading images, converting to greyscale, thresholding etc.) but in different ways. After researching the differences between the two it became apparent that either of the two could be used and would be effective. In the end the decision came down to speed. OCR is a resource intensive task, and because of this, execution time would always be difficult to keep to a minimum. We therefore had to find ways to shave it down in other parts of the OCR pipeline. Cv2 is written in C and C++ and PIL is written in Python and C. Because of the speed discrepancy between the languages, cv2 is slightly faster than PIL (1.4 times faster). We decided then to use cv2 for all image manipulation and processing prior to actually performing the OCR.

3.1.2 Pytesseract

With a way to read the image and pass it to an OCR function, we now had to research how the OCR function would work. All our reading pointed towards Pytesseract as being the standard for performing OCR in Python. With the help of some tutorials and the documentation we were able to write our first primitive OCR function. The function worked perfectly for test images generated on a computer e.g., screenshots of text. It did not however perform well at all for images of text captured in the real world. Given that this was the main functionality of our app, we began researching how we could improve the accuracy and reliability of the OCR output.

3.1.3 Thresholding

The majority of time spent researching OCR was spent at this stage. From tutorials and other online resources, it seemed that the most effective first step in preprocessing would be to convert the image to greyscale and then apply thresholding. Cv2 could perform greyscale conversion simply, but the thresholding would require more research. Luckily, we were enrolled in the module CA4007 Computer Graphics and Image Processing here in DCU and at around this time we had been learning about image manipulation. After some more research alongside extensive testing we decided to use the Otsu method of thresholding. On paper it seemed to be the best approach (identifying text as foreground objects and thresholding accordingly). In practice it also produced the best results.

3.1.4 De-Skewing

With thresholding researched and implemented we went back to the drawing board and looked for more ways to improve the output of the OCR function. De-skewing the image seemed like the obvious next step and would be very important given the input of user-generated images. We found

that a library called deskew had a function which could take an image as an input and output the degree to which an image is skewed. We then found cv2 functions in the documentation which would allow us to rotate the image.

3.1.5 Blurring and Filtering

The final piece of the preprocessing puzzle was blurring or filtering. Again, our college image processing module came in useful here in order to understand how these operations worked. We ended up focusing our research on median blurring and bilateral filtering. After an extensive testing and research period we found that applying both filters to the image before performing OCR saw a minor improvement in the quality of output, the improvement was not enough to justify the increase in execution time caused by these operations.

3.2 Amazon Web Services (AWS)

The research process for AWS was similarly a continuous process persistent through each step of the app's development. Given the number and breadth of AWS services, it would not be feasible or productive to research every resource type before going into the project. From past experience, we already had a baseline of AWS knowledge going into the project but the process of designing the architecture in its entirety was something completely new to us. We went about it by starting with a basic template of a serverless app back-end which we would research and tweak as necessary. We began with the core of any AWS-hosted serverless infrastructure – AWS Lambda.

3.2.1 Lambda

As mentioned, we had previous knowledge of what Lambda functions are and how they work going into the project, but we spent some time consolidating this knowledge and researching the features that would be specific to this project. Amazon provide extensive documentation and how-to guides on every aspect of their services, so it was relatively straightforward to perform this research. Most of the time was spent learning about Lambda layers, triggers, and destinations. Towards the later stages of the project as the library count increased, deploying Docker container images to Lambda also became an important feature to understand. We initially decided to use Lambda layers as a way of packaging the third-party libraries necessary for our functions (pytesseract, cv2). Some research provided a guide of how to do this by installing the libraries on an AWS Linux EC2 instance, zipping the content and then transferring it to an S3 bucket for use by the Lambda function. As we added more libraries to the main ocr_lambda function however, the combined layer size became too large for AWS' layer size limit. More research led us to the solution of Dockerizing the entire function alongside its libraries and storing the container in the Elastic Container Registry (ECR).

3.2.2 Step Functions

In the process of researching Lambda triggers and destinations, we began researching Step Functions as an alternative for passing information from one Lambda function to another. The main reason for choosing Step Functions over just sending output to a destination function each time was the ability to run functions concurrently. While using triggers and destinations would work, its sequential nature would badly affect our execution time. Instead, we could run both the pre-processing function and DynamoDB querying function concurrently, wait each of their outputs and cut down on the execution time.

3.2.3 Elastic Container Registry (ECR)

The ECR, as mentioned in section 3.2.1 was researched as a potential solution to a serious issue. Alongside just understanding AWS' registry we also had to do some revision of our knowledge of containers and Docker. When we had a full picture of how this service worked and how it interacted with Lambda, we felt comfortable that it fit our needs and that it would help us overcome our blocker.

3.2.4 DynamoDB

The DynamoDB research began as a more general overview of all the AWS database services. We knew the allergen matching function would require a database of allergens relevant to each allergy and that it would be static. The two main candidates were DynamoDB, AWS' NoSQL offering and RDS, their relational database. In the end it came down to DynamoDB's lower latency. In this context we just needed a simple, quick database and DynamoDB provided this to us. Minimal research was needed for the actual querying of the database as this was all handled by Boto3, an AWS library which allows resources to be accessed and modified from Python code.

3.3 Terraform

Terraform, like AWS was something we had worked with in the past so similarly just needed a bit of research to refresh our knowledge. The documentation provided by Hashicorp details every resource and all its parameters. The complexity with Terraform therefore comes from the initial architecture design rather than the infrastructure provisioning.

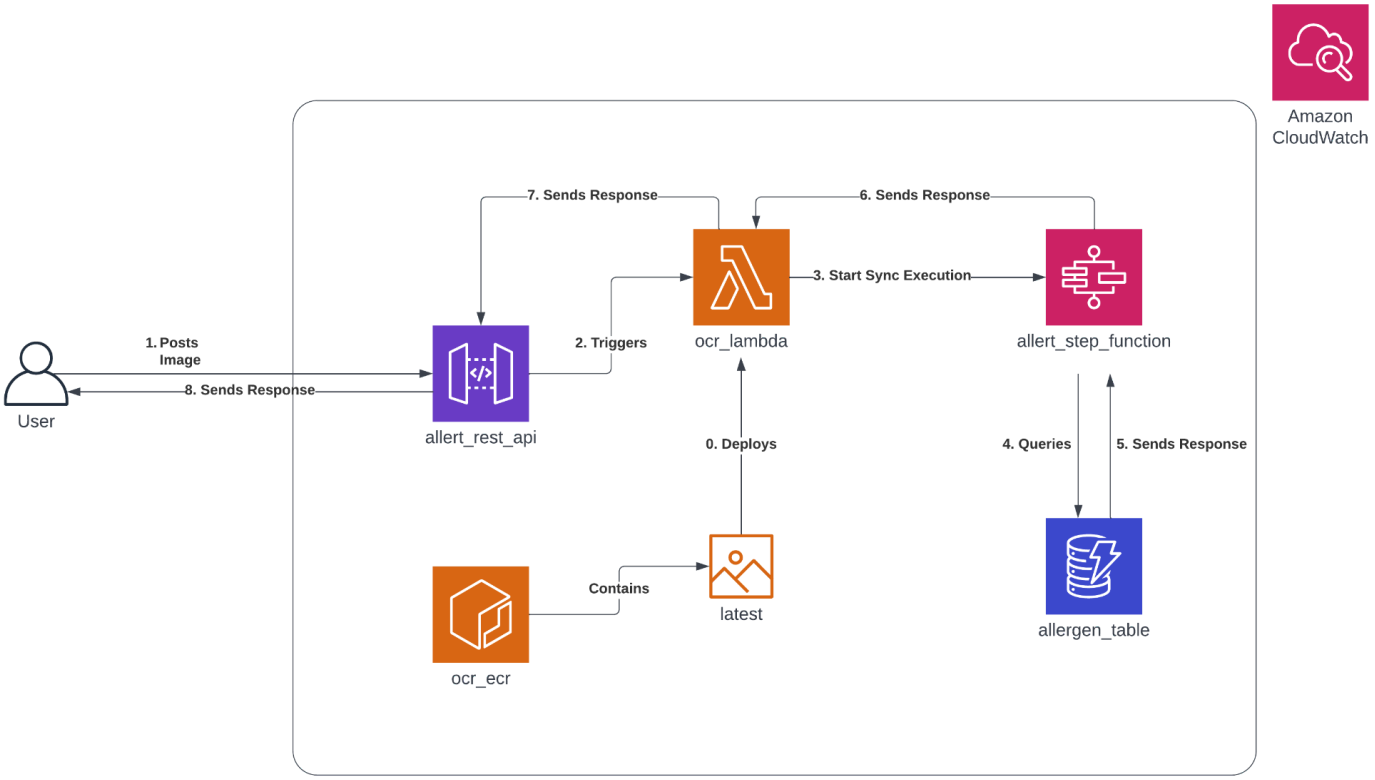
3.4 Allergen Matching

Once we had researched and implemented all the steps leading up to the allergen matching function, we then had to choose how user allergies would be compared against the output of the OCR. Here again we used a mixture of sources to come up with a list of possible functions, each of which produced the same output but in a different way. Like a number of other aspects of the application, we used our research to form a hypothesis for which would be optimal and then used tests to confirm this hypothesis. This allowed us to not only evaluate the effectiveness and speed but also understand why the results were what they were.

4. Design

The following section details some of the design choices we made over the course of this project. One of the advantages of AWS and cloud computing is that the infrastructure is completely customisable to each individual or team's needs. In this project we had the opportunity to evaluate what these needs were and how our infrastructure could best help us achieve these targets.

4.1 System Architecture



The diagram above outlines the various high-level components of Allert's system architecture and how they interact with one another. The entire sequence begins when a user takes an image of the product ingredients list they wish to scan. This triggers an API POST request to our AWS environment's API Gateway allert_rest_api. The request consists of a JSON file containing a base64 representation of the image and a list of the user's allergies. Upon receiving a request, the API Gateway triggers the Lambda function ocr_lambda. This function is the most important in the entire architecture. Here the image is decoded from base64, pre-processed, and has OCR performed on it. It is also responsible for invoking the step function within which the rest of the Lambda functions are contained. In the step function the DynamoDB Lambda queries the allergen_table for allergens associated with the user's allergy. The output of the step function is a list of user allergies which may be triggered by some of the ingredients within the product. This is collected as a response by the ocr_lambda function before being sent on to the API Gateway and back to the user's device where it is displayed. All components of the architecture are sending logs to CloudWatch each time they are invoked. The logs contain

4.2 API Gateway

The first resource in the entire back end that the user interacts with is the API Gateway. When a POST request is sent to the API endpoint it immediately passes this request and its body onto the main Lambda function ocr_lambda. This is because the API has been designated as an "AWS proxy". As mentioned, this allows the API to invoke the Lambda function (as long as the correct IAM role is attached). It also means that the API can take output from the Lambda function and convert it to a HTTP response which will be sent back to the user's device.

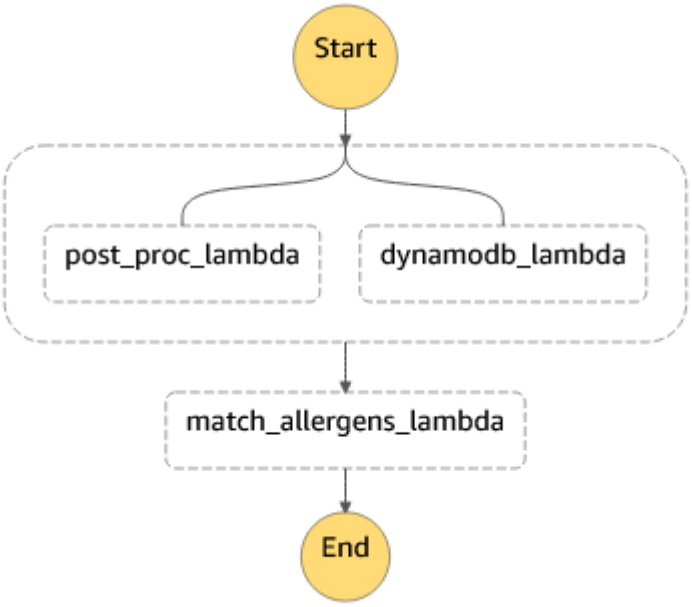
4.3 Lambda

Each Lambda function has a different configuration. They do however all share the same IAM policy and role. The policy states that any Lambda function can perform any operation (read, write, modify etc.) on the CloudWatch logs. They can also trigger other functions to start and read from the DynamoDB. The actual code for each function is stored locally and in the GitLab repository. Due to the use of Terraform for infrastructure as code, to update any lambda function, the code can be changed locally, Terraform apply run and then Terraform will make the changes for you. All functions except for the ocr_lambda function are configured this way and make use of third party libraries by wrapping them in Lambda layers. Ocr_lambda however exceeds the Lambda limits for layer size and so had to be dockerized and stored in the ECR. Any time changes are made to the ocr_lambda code, the shell script linked below is used to update the ECR image.

```
#!/bin/bash

sudo docker build -t ocr_lambda .
id=`docker images | awk '/260350295037.dkr.ecr.eu-west-1.amazonaws.com/ocr_ecr/ {print $3}'`
docker tag $id 260350295037.dkr.ecr.eu-west-1.amazonaws.com/ocr_ecr
docker push 260350295037.dkr.ecr.eu-west-1.amazonaws.com/ocr_ecr
```

4.4 Step Function



The diagram above details the components of the step function. The ocr_lambda function triggers its execution and passes input to it. At this stage it reaches the Start node and splits into two separate branches running concurrently. Each function takes a portion of the step function input and processes it. Post_proc_lambda takes the string of OCR text generated from the initial Lambda and tidies up the output, splitting it up and removing any characters which will not be useful.
On the other hand dynamodb_lambda takes the list of user allergies as input and queries the DynamoDB for all allergens associated with it/them. As the parallel portion of the step function ends, output from both functions is combined as input to the match_allergens_lambda function.

4.4.1 Post processing branch input

```
{
    "id": "5",
    "type": "LambdaFunctionScheduled",
    "details": {
        "input": "\"' [@wa/ẽwtfs  pepper sauce dry mix. wheat flour, salt, maizr; starch, potato starch, vegetable oils (palm oil, rapeseed oil, palm kerne  oil), vegetables (tomato. onion), maltodextrin, 5ng i ssfywhile plain caramel, pepper faciandcontinue (3.2%), yeast extract, _.€'i59for emulsifier (mono— and i - ' diglycerides of fatty adds) 7 ' flavourings (contain 1 celery), citric acid, garlic powder, colours (riboflavin, beta-carotene), parsley, soya flour, milk proteins.\"",
        "inputDetails": {
            "truncated": false
        },
        "resource": "arn:aws:lambda:eu-west-1:260350295037:function:post_proc_lambda"
    },
    "previous_event_id": "4",
    "event_timestamp": "1649786130675",
    "execution_arn": "arn:aws:states:eu-west-1:260350295037:express:allert_step_function:bbe3c5cc-ba89-11ec-b6b1-7239204fc541:0ee8e001-13c7-443f-97a3-60600f73da8a"
}
```


4.4.2 DynamoDB branch input


```
{
    "id": "7",
    "type": "LambdaFunctionScheduled",
    "details": {
        "input": "[\"gluten\",\"soybeans\",\"milk\",\"celery\"]",
        "inputDetails": {
            "truncated": false
        },
        "resource": "arn:aws:lambda:eu-west-1:260350295037:function:dynamodb_lambda"
    },
    "previous_event_id": "6",
    "event_timestamp": "1649786130675",
    "execution_arn": "arn:aws:states:eu-west-1:260350295037:express:allert_step_function:bbe3c5cc-ba89-11ec-b6b1-7239204fc541:0ee8e001-13c7-443f-97a3-60600f73da8a"
}
```


4.4.3 Combined output


```
{
    "id": "15",
    "type": "ParallelStateExited",
    "details": {
        "name": "Parallel",
        "output": "[[\"acid\", \"adds\", \"and\", \"betacarotene\", \"caramel\", \"celery\", \"citric\", \"colours\", \"contain\", \"diglycerides\", \"dry\", \"emulsifier\", \"extract\",
\"faciandcontinue\", \"fatty\", \"flavourings\", \"flour\", \"garlic\", \"i\", \"ifor\", \"kerne\", \"l\", \"maizr\", \"maltodextrin\", \"milk\", \"mix\", \"mono\", \"ng\", \"of\", \"oil\",
\"oils\", \"onion\", \"palm\", \"parsley\", \"pepper\", \"plain\", \"potato\", \"powder\", \"proteins\", \"rapeseed\", \"riboflavin\", \"salt\", \"sauce\", \"soya\", \"ssfywhile\", \"starch\",
\"tomato\", \"vegetable\", \"vegetables\", \"wawtfs\", \"wheat\", \"yeast\"],{\"gluten\": [\"atta\", \"barley\", \"beer\", \"betaglucaan\", \"bran\", \"bread\", \"bucket\", \"bulgur\",
\"couscous\", \"crumbs\", \"cruseli\", \"dinkel\", \"durum\", \"emmer\", \"farina\", \"farro\", \"fiber\", \"flour\", \"germ\", \"gliadin\", \"glucan\", \"gluten\", \"glutenin\", \"gran\",
\"grits\", \"groats\", \"hydrolyzate\", \"kamut\", \"macaroni\", \"malt\", \"malted\", \"matzo\", \"millet\", \"museli\", \"noodle\", \"oat\", \"oatmeal\", \"oats\", \"pasta\", \"rye\",
\"seitan\", \"semolina\", \"spaghetti\", \"spelt\", \"starch\", \"triticale\", \"triticum\", \"vermicelli\", \"wholegrain\"], \"soybeans\": [\"chiang\", \"edamame\", \"eiwothydrolysaat\",
\"glycine\", \"guar\", \"hemicellulose\", \"hoisin\", \"isoflavones\", \"kinako\", \"lecithin\", \"miso\", \"natto\", \"okara\", \"soy\", \"soya\", \"soybean\", \"soybeans\", \"starch\",
\"tamari\", \"tempeh\", \"tofu\", \"tsp\", \"tvp\", \"yuba\"], \"milk\": [\"butter\", \"butter\", \"butterfat\", \"buttermilk\", \"calcium\", \"calciumstearoylactylaat\", \"casein\",
\"caseinate\", \"cheddar\", \"cheese\", \"cream\", \"creme\", \"ghee\", \"glycomacropeptide\", \"immunoglobulin\", \"kefir\", \"lacitol\", \"lactose\", \"margarine\", \"milk\", \"milkfat\",
\"mozzarella\", \"nisin\", \"tagatose\", \"whey\", \"yoghurt\", \"yogurt\"], \"celery\": [\"celeriac\", \"celery\", \"eppe\"]}]",
        "outputDetails": {
            "truncated": false
        }
    },
    "previous_event_id": "13",
    "event_timestamp": "1649786132855",
    "execution_arn": "arn:aws:states:eu-west-1:260350295037:express:allert_step_function:bbe3c5cc-ba89-11ec-b6b1-7239204fc541:0ee8e001-13c7-443f-97a3-60600f73da8a"
}
```


Above we can see how each element is separately passed to each branch of the step function and then the output is recombined when exiting the parallel state.

5. Implementation

5.1 Ocr_lambda

The main OCR Lambda function consists of 3 sub-functions. All Lambda functions must have a handler function. This is the entry point of the code and is where the input is passed to. For the ocr_lambda function, the POST body is decoded (the API json input is base64 encoded) and then the image base64 string is further decoded. The image is temporarily saved and read using the cv2 library. The flag “0” passed to the cv2 function imread() specifies that the image should be read as a greyscale image. The image is then essentially converted to a 2-dimensional matrix upon which operations can be performed.

The first of these operations will occur when the image has been passed to the ocr() function. A threshold is applied to the image to make it easier for the OCR method to recognise the characters. The parameters are as follows; source, threshold value, the maximum value and the threshold type.

```
result = cv2.threshold(img, 0, 255, cv2.THRESH_OTSU)[1]
```

Next a deskewing function is called. This is a vital component as even a slight skew can cause a huge drop in OCR quality. The deskew function uses the deskew library to determine how much the image is skewed by, finds the centre of the image and uses the cv2 library to rotate the image.

```
def deskew_fn(img):
  angle = determine_skew(img)
  image_center = tuple(np.array(img.shape[1::-1]) / 2)
  rot_mat = cv2.getRotationMatrix2D(image_center, angle, 1.0)
  result = cv2.warpAffine(img, rot_mat, img.shape[1::-1], flags=cv2.INTER_LINEAR)
  return result
```

Finally, the preprocessed text is passed to Pytesseract's OCR function, image_to_string(). The output of this function is converted to lowercase and has any leading or trailing whitespace removed.

The result of the ocr() function can now be passed to the step function alongside the user allergies and the user ID which is only necessary for the invocation of a step function. Once a response is returned from the step function, ocr_lambda will send it back to the API Gateway and because of the proxy nature of the API, wrapping the output as HTML response will all be handled by it.

5.2 Post_proc_lambda

The purpose of this Lambda function is to perform various operations on the OCR output to make it better suited to the match_allergens_lambda function. All non-alphabetic characters are removed from the string and it is split up into a list of strings. This will make it much easier to compare the OCR output to the allergen data stored in our DynamoDB. The list is then sorted and any duplicate terms are removed. Throughout this project reducing execution time has been a focus and this is another method for achieving a more efficient, faster process. Regex and collections are used to perform non-alphabetic removal and duplicate term removal respectively.

5.3 Dynamodb_lambda

This lambda function runs concurrently alongside the previous post-processing function. It performs a simple task of iterating through the user's list of allergies and querying the DynamoDB for any allergens relating to each allergy. Once the allergies have been passed as input by the step function, this function reads it and iterates through it. For every allergy in the list, the dynamodb_query() function is called. Using AWS' boto3 library, it is possible to query the value associated with a key. In this case the key is the allergy and the value is a list of allergens. Below we can see the resource name and the actual DynamoDB table we would like to query.

```
client = boto3.resource('dynamodb')
table = client.Table('allergen_table')
```

```
resp = table.query(
    KeyConditionExpression=Key('allergy').eq(allergy)
)
```

The result is split up into a list of known allergens, the process is repeated for any more allergies and then the list is sent off to the next step in the step function.

5.4 Match_allergens_lambda

The final step is to take both outputs from the previous function and compare one against the other. Depending on how many allergies a user suffers from, this function will iterate through one or more allergies and associated allergen lists. Each iteration will call the set_compare() function which returns a boolean if any of the elements of the allergen list are found to be present in the OCR string that has been split into a list. There were a number of options of methods to achieve this but after testing all available solutions, converting the lists to sets and then finding the negation of isdisjoint() was the most consistently fast.

```
def set_compare(text, alls):
    return not set(text).isdisjoint(set(alls))
```

If this function returns True, the allergy in question is added to a list and passed back to the ocr_lambda function before being passed on to the API Gateway.

6. Problems and Solutions

6.1 Exceeding Lambda Layer Size Limit

Once we had decided on hosting our serverless backend on AWS, we began researching how we could use the third-party libraries necessary. The answer to this question it seemed was installing the dependencies in an AWS Linux EC2 directory, zipping the folder and uploading it to an S3 bucket. This would create what is called a Lambda layer. By attaching this layer to our Lambda function we could ensure that libraries that were central to our app's functionality could be imported. This worked perfectly initially when we were implementing a very basic Pytesseract function with some preprocessing in the form of thresholding using cv2. At this stage we were unaware of the hard limit of 250MB for Lambda functions. The code and the dependencies when combined together cannot exceed a size of 250MB, we had not encountered this limit as our package was just under the 250MB. However when we went to implement the deskew function that we had been working on and that was integral to reliable OCR, we found that even a library of a small size like the one we were using to calculate the deskew angle would push us over the 250MB threshold. We began to research solutions to this problem.

There were suggestions of storing the dependencies in S3 buckets not as Lambda layers but as regular files that could be imported into our function and used. This did not seem particularly appealing given the impact it would have on our execution time. Importing these libraries from S3 would be much slower than the application we had envisioned. Instead we found some suggestions that dockerizing the application along with its dependencies and storing it in AWS' Elastic Container Registry would allow us to up our combined file size to a limit of 10GB. It also did not sacrifice speed to achieve this. The docker image would be deployed to the Lambda function and therefore would not need to be calling any other services on a cold start. Although there was not much documentation online about how to employ this solution (it had only recently been developed by AWS), we stuck with it and were successful in the end. In this instance we avoided trading off speed for functionality.

6.2 API Gateway IAM Role Not Attaching

Terraform proved to be an invaluable asset to this project's development. Provisioning infrastructure as code gave us greater visibility and flexibility to adapt our resources as conditions changed. Terraform generally will not apply any changes you make if these changes will not work. For example

if I try to provision an EC2 instance with an invalid AMI, the terraform apply command will return an error detailing what is wrong and how to fix it. This is how terraform generally works and it means that you are always in control of the configuration of your infrastructure. During this project however we came across one exception to this practice. When directly declaring a resource's parameters in the resource body, checks will be made to ensure that everything is correct. However, when provisioning a resource outside the body of a certain resource and then attaching that resource, there is no feedback as to whether the two resources are compatible.
We found out about this quirk when provisioning the API Gateway. The API would need the ability to invoke a Lambda function and to do this it would also need permission. Permissions are managed by a service called IAM and can take the form of roles and policies. Policies are files in JSON format that allow users and services to perform defined operations on other resource types. The usual way of attaching these policies to resources is to first attach them to a role and then attach the role in the body of the service in question. API gateway however works differently and needs a policy attachment object to be declared separate to the API object. This policy attachment object must point towards the API in its body but as long as the API object exists, Terraform will not make any more checks for some reason.
We learned about this flaw through a lot of frustration and confusion. We initially tried invoking the ocr_lambda with the API gateway but could not get it to work. On top of this, we could not enable logging and narrow down the source of the issue without providing the API with permission to write to the logs. Eventually by exhausting all other options we came to the solution. It was disappointing that something so simple had proven to be such a blocker but that is the nature of working with any kind of third-party software. The fix in the end was a simple one so we were able to overcome the issue and move on to the next phase of the project.

7. Testing

Testing a mobile application hosted on AWS is a straightforward and continuous process but one that can be hard to formalise. Our testing process began early on as we developed the OCR component of the project locally rather than in the cloud. We ran tests on the various preprocessing elements of the OCR pipeline by iterating through increasing values of the function parameters and using cosine similarity to judge their effectiveness. We generated a number of test images and manually transcribed the ingredients and other words included on the packaging. We then made slight tweaks to the parameters of the cv2 functions we intended to use. For median blur, we incremented the ksize by 2 each iteration. For bilateral filtering we increased the sigma value. We found a function for converting two bodies of text to vectors and then measuring the similarity between them. The vectors we were comparing were the human transcribed ingredients and the OCR generated ingredients. The closer two were to each other the higher percentage the function returned. The tests would run over and over again for each image and each parameter value until we finished and generated a report of the effects. From here we could make decisions on what parameters we would employ.

In a similar vein we wrote tests for comparing the various allergen matching functions we had come up with. We wanted to find the consistently quickest means of determining whether any of the user allergen terms were in the OCR generated output. We used a bash script to run and evaluate each function in various conditions. Below you can see the script which triggered the tests.

```bash
#!/bin/bash
function run_tests() {
    set_compare=$(python3 -m timeit -s "from match_allergens_test import set_compare" "set_compare($text,$alls)")
    set_disjoint=$(python3 -m timeit -s "from match_allergens_test import set_disjoint" "set_disjoint($text,$alls)")
    if_any=$(python3 -m timeit -s "from match_allergens_test import if_any" "if_any($text,$alls)")
    if_any_lc=$(python3 -m timeit -s "from match_allergens_test import if_any_lc" "if_any_lc($text,$alls)")
}

function print_results() {
    echo "Set Compare: $set_compare"
    echo "Set Disjoint: $set_disjoint"
    echo "If Any: $if_any"
    echo "If Any List Comprehension: $if_any_lc"
    echo
```

The results were then written to a txt file and we could clearly see that set disjoint was not always the quickest but did not fluctuate in the way that if any methods did. It stayed consistently quick.

```
Match at end of alls list
Set Compare: 500000 loops, best of 5: 984 nsec per loop
Set Disjoint: 500000 loops, best of 5: 829 nsec per loop
If Any: 100000 loops, best of 5: 2.66 usec per loop
If Any List Comprehension: 100000 loops, best of 5: 2.26 usec per loop

Match at start of alls list
Set Compare: 200000 loops, best of 5: 1.01 usec per loop
Set Disjoint: 500000 loops, best of 5: 860 nsec per loop
If Any: 500000 loops, best of 5: 576 nsec per loop
If Any List Comprehension: 500000 loops, best of 5: 570 nsec per loop

Match at end of text list
Set Compare: 500000 loops, best of 5: 928 nsec per loop
Set Disjoint: 500000 loops, best of 5: 901 nsec per loop
If Any: 200000 loops, best of 5: 1.24 usec per loop
If Any List Comprehension: 200000 loops, best of 5: 1.25 usec per loop

Match at start of text list
Set Compare: 500000 loops, best of 5: 940 nsec per loop
Set Disjoint: 500000 loops, best of 5: 848 nsec per loop
If Any: 200000 loops, best of 5: 1.23 usec per loop
If Any List Comprehension: 200000 loops, best of 5: 1.19 usec per loop

No matches
Set Compare: 500000 loops, best of 5: 939 nsec per loop
Set Disjoint: 500000 loops, best of 5: 930 nsec per loop
If Any: 100000 loops, best of 5: 2.44 usec per loop
If Any List Comprehension: 100000 loops, best of 5: 2.41 usec per loop
```

We also wrote a number of end to end tests which would pass a number of JSON files to our API Gateway. These files were in the same format as expected input for our functions and contained base64 representations of ingredients list images. We judged these tests to be successful if the output was expected (the allergens were detected). We could also use CloudWatch logging to inspect how the various lambda functions worked.