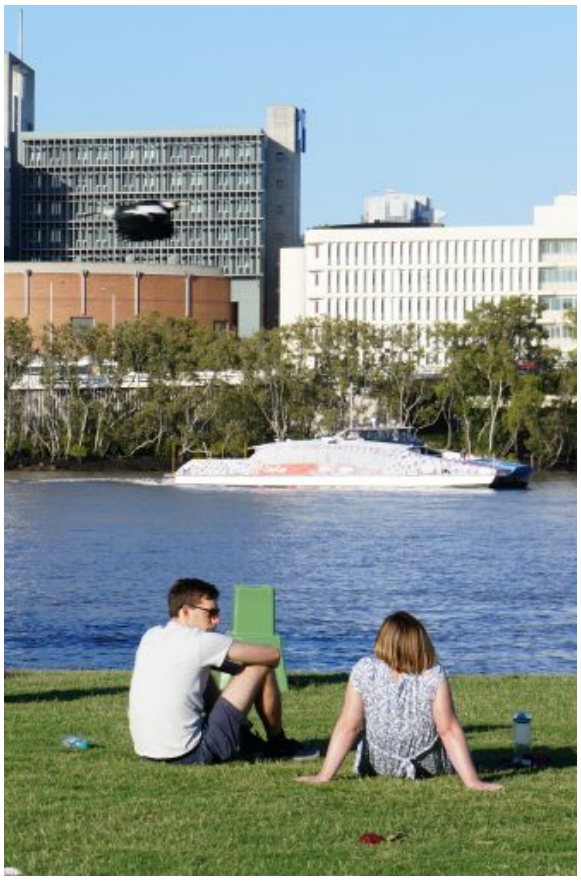
CAB 432 – Cloud Computing

Assignment 2

By Jordi Smit - 10264139



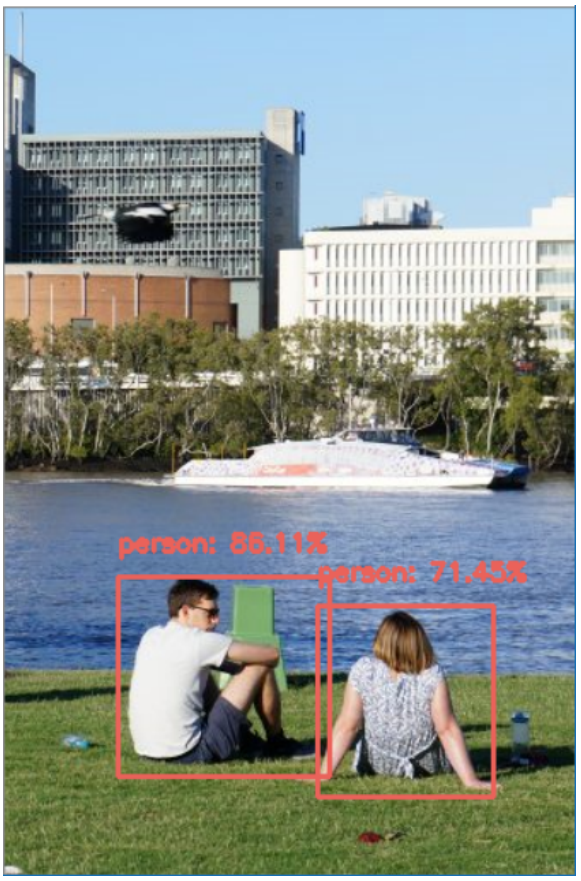


Table of content

1. Introduction

This application allows its users to analyze large amounts of images from Flicker (“Flickr Services,” n.d.) around a specific area. Using OpenCv (“OpenCV-Python", n.d.) each image will be scanned for multiple items such as people, cars, cats, etc. The results of these scans will provide the user with the locations of these items within the image and how many times these items have been detected in the image (Figure 1.a).

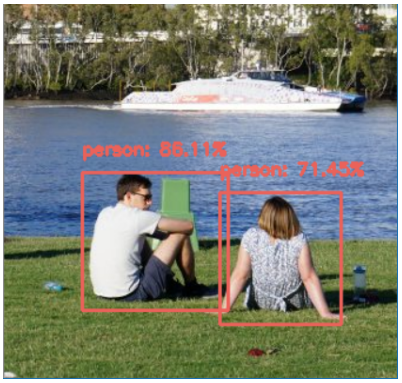


Figure 1.a Visualization of the output of OpenCv

The application has 3 pages. The index page provides the user with an overview of his previously performed searches. From this page the user can either go view an old search or start a new one. The user can start a new search using the search setting page. On this page the user will provide the application with information such as search location, how large the area should be, the maximum amount of images to scan.
Once the user clicks on the submit button, he will be redirected to the search result page. On this page the user will see the following information:

- How often has a specific label been detected;
- Which percentage of images has a specific label at least once,
- The response time per batch;
- The detection labels and locations for each image;

The search will be performed asynchronously in the background. This will allow the user to immediately go through the intermediate results, while new results will continuously be added.

The idea of this application is based on scenario 2 (Webcams) (“Website,” n.d.). However the image dataset from the webcams proofed to have a resolution that is too low for my OpenCv classification Neural Network (Rosebrock, 2017). That is why I decided to use Flickr as a datasource.
The analysis of a single image takes between 500 ms and 1000 ms depending on the resolution and the latency between the server and the client side. Combining this with the extremely large data set of Flickr, we should be able to create a significant load.

2 User stories

2.1 Use case A

As a user I want to know how often specific types of items (such as cars, people, cats, etc) are being shown in Flickr photos that have been take around a specific area.

The user starts by specifying its search parameters on the search settings page. The user will be able to specify search parameters such as the location of interest, the area around this location, the maximum amount of images to search, etc.

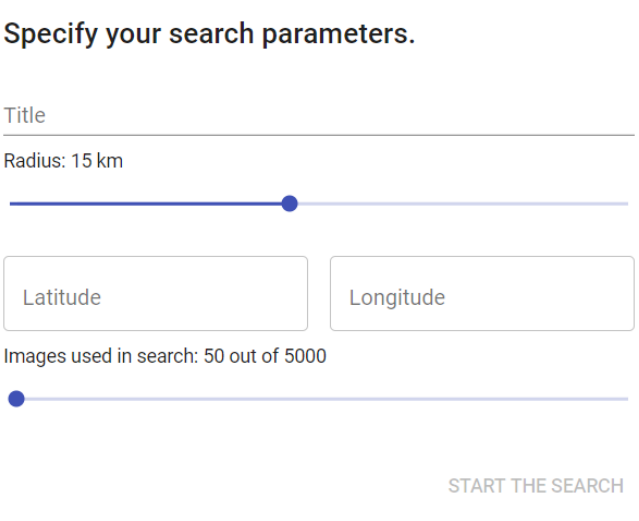


Figure 2.1.a The search form

The user can specify the search location using a leaflet map (“Leaflet — an open-source JavaScript library for interactive maps,” n.d.). When the user click on the latitude or longitude field a map will automatically open. The user only has to click on the map to specify the search location (Figure 2.1.b).

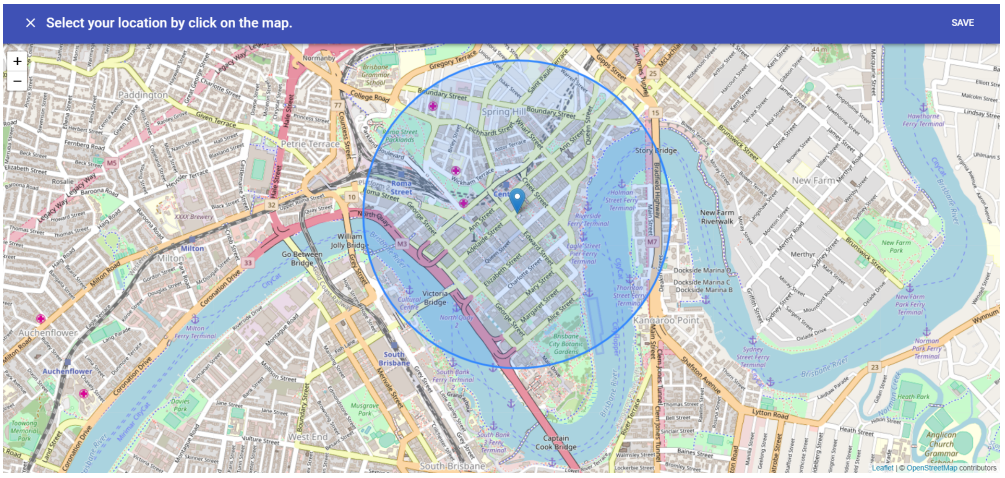


Figure 2.1.b Selecting the search area using leaflet

Once the user has specified his search parameters the application will start the analysis. While the application is performing the analysis the user will be provided with the intermediate results on the analytics page. This page will contain a bar chart that will display how often a specific class of items has been detected (Figure 2.1.c).

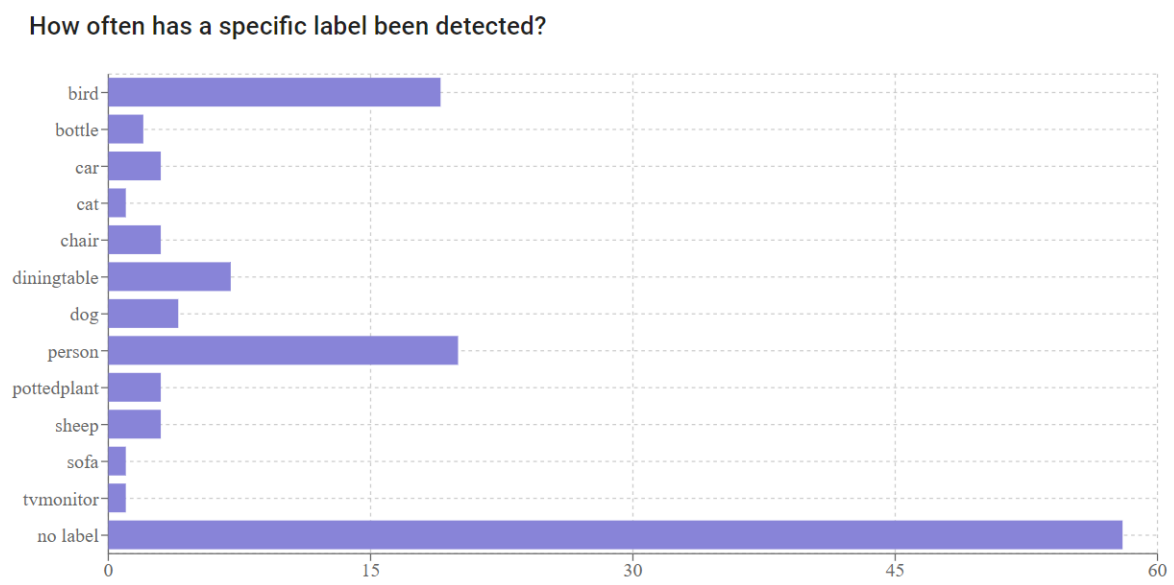


Figure 2.1.a An example of a result bar chart generated using rechart (“Recharts,” n.d.)

This use case makes use of the AWS load balancer to redirect the user’s request to an available server instance. This use case also makes use of AWS Auto Scaling Groups to handle the scaling out when the load increases. This use case also makes use of the Flickr API as a data source.

2.2 Use case B

As a user I want to know which Flickr photos in a specific area contain specific types items such as cars, people, cats, etc.

The user starts a search (as specified in 2.1) which will redirect the user to the analytics page. On this page the user will be provided with a list of images that have been scanned. For each of these images the page will display which type of items have been detected on it.

Figure 2.2.a The results for each image overview.

The user can see the results of a specific image by clicking on it. This will open a modal with a larger image. This image will contain rectangles that indicate the areas of each label. These rectangles will also contain the label and how confident the neural network is about the classification (Figure 2.2.b).

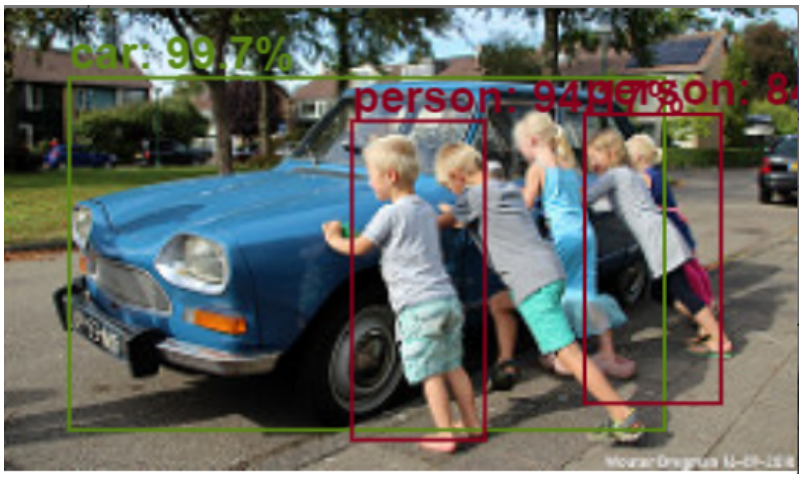


Figure 2.2.b The results for specific image.

This use case makes use of the AWS load balancer to redirect the user's request to an available server instance. This use case also makes use of AWS Auto Scaling Groups to handle the scaling out when the load increases. This use case also makes use of the Flickr API as a data source.

2.3 Use case C

As a user, I want the system to stay responsive even if it is also performing other searches or if a sub service is temporarily unavailable.
When a user starts a search (as specified in 2.1) the system should be able to handle this. The image scanning task is the most computationally expensive. That is why the image scanning servers are in there own autoscaling group while all the other requests are in an other autoscaling group. This will ensure that all the other requests will keep working with the same speed as before. If the image scanning load increases the autoscaling group will

automatically scale the amount of available servers while the load balancer will distribute the work over them. This will ensure that the application will stay responsive while the search queries are being performed.
It is possible that some of the scanning operations will fail during the automatic scaling process or are not divided equally over the growing cluster. The application handles these failures by sending the images in batches and using retries with exponential back-off on failure.This approach allows us to reschedule a batch if something goes wrong on the server side. If all else fails the server also makes backups of the search results which are stored in S3. This allows the search to resume from the last saved checkpoint if needed.

This use case makes use of the AWS load balancer to redirect the user's request to an available server instance. This use case also makes use of AWS Auto Scaling Groups to handle the scaling out. Finally this use case also makes use of AWS S3 storage service.

2.4 Use case D

As a user I want to be able to store my search results, such that I can look at them again in the future without having to perform the entire search again.

On the index page the user will be provided with an overview of his previously performed searches (Figure 2.4.a). These searches are stored in S3 as json files. On this page the user can either choose to delete them or view them.

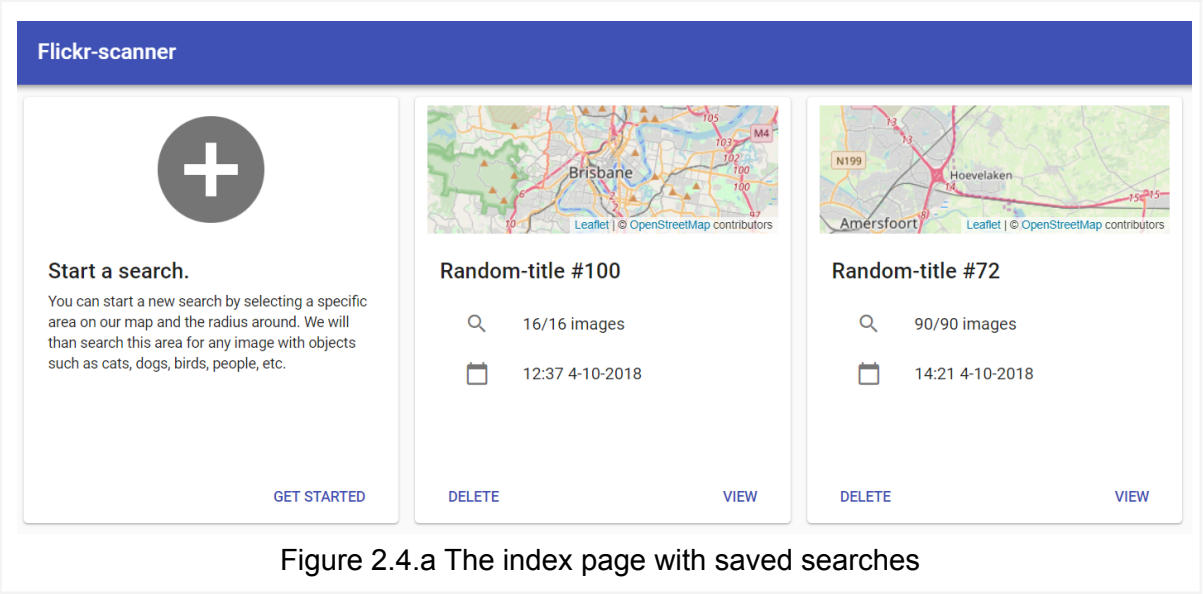


Figure 2.4.a The index page with saved searches

When the user clicks on the view button he will be redirected to the search result page. On the search result page the user will be provided with an summary of the search parameters (Figure 2.4.b). This is summary card also has a save button. Using this button the user can manually save the results. However it is not necessarily needed to do this manually, since the search automatically saves the results after performing an x amount of image scans.

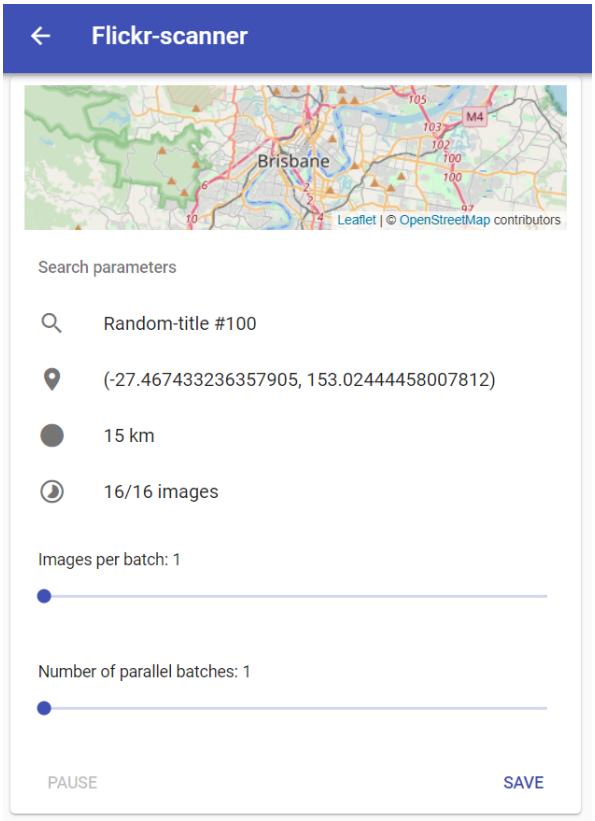


Figure 2.4.b The save option on the search result page

This use case makes use of the AWS load balancer to redirect the user's requests to an available server instance. This use case also makes use of AWS's S3 to retrieve and store the persistent data.

3 Technical Breakdown

The application architecture consists of 5 parts: The client side (3.1), the 2 auto scaling groups (3.2), the load balancers (3.3), and the S3 presistances services (3.4). The general architecture can be seen in figure 3.a. Besides this main architecture we also make use of some external services (3.5).

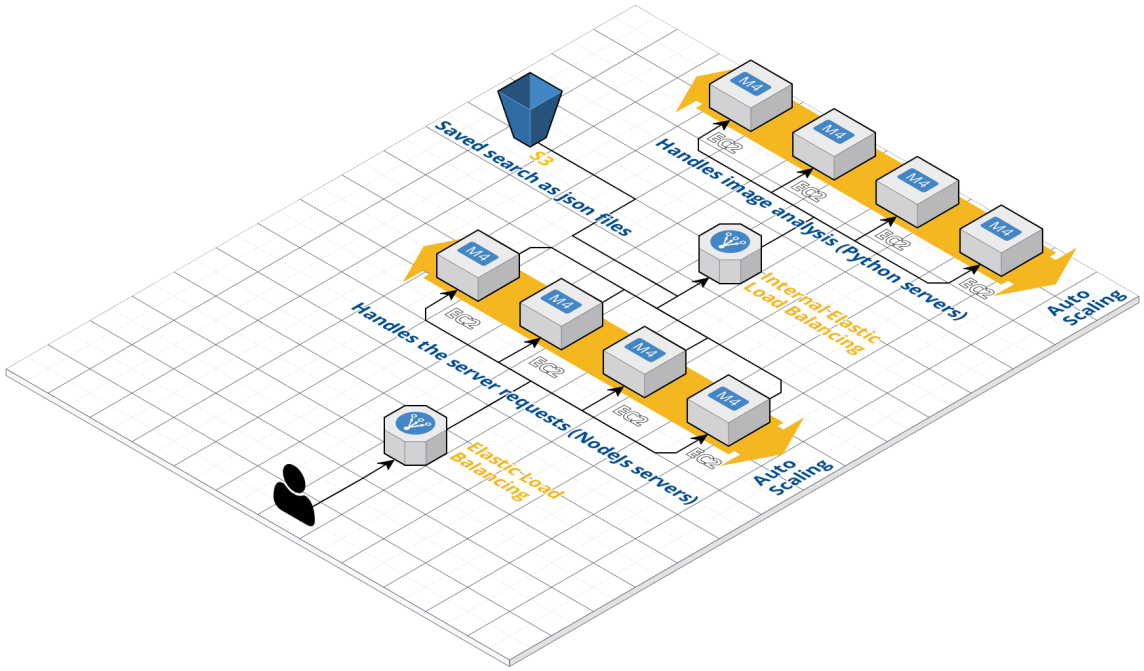


Figure 3.a The general architecture

3.1 Client side

The main responsibility of the client side is to display the data and to take user inputs. The analysis is computationally expensive and will take a while, so in order to keep the UI interactive the data has to be combined on the client side. So its secondary responsibility is to combine the different scan results. Luckily this combining part of the data (storing it in memory, summing, formatting for a bar charts, etc) is relatively simple and can easily be done on the client side.

The API is REST based and completely stateless. This means that in order for the search to continue the client has to keep sending data to the server. The client side does this by selecting batches of images and sending them in parallel HTTP requests. It then awaits these responses and updates the state and the view on the client side and continues to the next round. Every X amount of rounds the client side sends the current state to the API in order to back it up. The client side keeps doing this until all the images have been analyzed. The main advantage of doing it this way is that the design is entirely stateless, which means that theoretically we can send as much parallel requests as we want. While keeping the response time relatively low depending the amount of input per request.

The client side has been built using the ReactJs framework (“React – A JavaScript library for building user interfaces,” n.d.). This framework allows developers to create interactive UIs without the need to reload the entire page. When the user navigates to the website he will be served with the entry point of the ReactJs application. The React-Router will read the current url and will serve the defined view for this url (or a 404 page). React-Router also allows the user to navigate between urls without reloading the entire page. Each view will request the data it requires from the server, on a as needed basis, using AJAX to ensure that the application runs as fast as possible.

3.2 Auto Scaling groups

The application has 2 separate EC2 autoscaling groups. The first autoscaling group is responsible for handling the simple requests such a serving the page (3.2.1), while the second autoscaling group is specifically responsible for handling the image scans (3.2.2). The image scan is computationally much more expensive than the simple requests. This is the reason why the image scan service has its own auto scaling group. The image scan autoscaling group is not exposed through the outside world and can only be accessed through a simple requests server.

3.2.1 Simple requests autoscaling group

The main responsibility of the servers in this autoscaling group is to serve the single page ReactJs application and exposing the REST API. The client side can access the api on the /api path using AJAX. These servers do not do much heavy work, they merely accept requests and transform the input data and use it as input for other APIs such as Flickr, S3 or the image scan servers.
These servers are written in NodeJs and will be responsible for handling the requests from the client side such as for example serving the webpage, retrieving data from the Flickr API, retrieving data from S3 and redirecting the image scan requests.
The load on these servers is relatively low, so most of the time a single instance will be sufficient.

3.2.2 Image analysis autoscaling group

The image analysis autoscaling group is solely responsible for performing the image analysis. The task lives in its own autoscaling group since it is computationally much more expensive. If we would run this task on the same server as the API, it would slow done all the other requests.
The image analysis uses OpenCv to perform the analysis. This server has been written in Python using Flask because OpenCv is not as far developed in NodeJs as in Python.
The servers in this autoscaling group are not meant to be accessed on their own. That is why each EC2 instance has a private IP which can only accessed through their load balances by EC2 instances of the simple request servers.
The amount of instances in this group depends on the amount of images that needs to be processed. Each server can process a single image in between 100ms and 500ms.

3.3 Load balancers

The architecture has 2 load balancers because we have 2 auto scaling groups. The first one is an externally exposed load balancer and the second one is an internal load balancer. The externally exposed load balancer is responsible for gathering all the requests from the client side and distributing them over the API servers in the simple request autoscaling group. The internal load balancer distributes the image scan requests from the API servers over the server instances in the image analysis group. Both the API server and the image analysis server have been designed to be completely stateless. That is why each request can be send to any server. When persistence is needed the server will read or write to a AWS S3 file in order to stay stateless (3.4). This allows us to run as many server instances in parallel as we need.

3.4 S3 presistances services

In order to keep the servers stateless while having persistence, we use AWS S3 to store the persistence data as json objects in json files. We use a single S3 bucket in order to store all these files. Each time we create a persistent entity, we give it a UUID. When we need to store the persistent entity we write it data into '<UUID>.json'. When we need to access the data of this entity we can simply read the file with this ID. We can also find all the persistent entities by simply listing all the file names inside the bucket. When we need to update the entity we can simply overwrite the file.
During development we used LocalStack in order to mock the S3 services without having to pay for it (localstack, n.d.).

3.5 External services

This Application only uses the Flickr API as an external API. This API is used to search images that have been geotagged around a specific point. These images will be used as the data set to generate the required load. Depending on the selected location the API can provide us with a dataset range form a couple of 100 images to more than 100.000 images.

4 Development

In the development process I was mainly concerned with designing the application with scalability (4.1) in mind and designing and deploying the architecture (4.2).

4.1 Scalability

The image analysis part of the application scales really well because it is designed completely stateless and is computationally expensive. In order to analyse a specific image, we send the server a request with a pointer to the image to analyse on the Flickr server. The server reads the image into RAM from the Flickr server and analyse it. The server then forgets about the image and returns the results of the analysis as response.
Thus the query results depend only on the input, which means that we can split all the analysis requests and perform them on different servers in parallel. Most modern web browsers are able to maintain up to 6 parallel HTTP connections. Thus for a single search we can run up to 6 batches in parallel and have all of them return their results within roughly the same response time.

The simple request server is also designed to be completely stateless and thus it can also be scaled out. However the responses of this server are not computationally expensive, which means that it is very unlikely that we ever need more than 1 instance of this server. Nevertheless this server lives in its own autoscaling group so that it can scale out if needed. The main reason why this server lives inside its own autoscaling group is so that its performances is unaffected by the load on the image analysis server.

4.2 Architecture creation

The two autoscaling groups have been designed to scale in and auto without personal attentions. That is why the simple request server (4.2.1) and image analysis server (4.2.1) are being deployed using scripts and auto scaling policies. The other parts of the architecture load balancers (4.2.3) and the S3 bucket (4.2.4) has been created using AWS console.

4.2.1 Simple request server

The simple request server instances are created using the script in figure 4.2.1.a. Once the EC2 instance has been initialised the script will start off by installing docker and docker-compose. The script then creates and runs a docker-compose file which specifies how to start up the docker image. After the script is done the EC2 instance will be ready and have the server exposed on the port on which the load balancer is listening.

```
#!/bin/bash
DOCKER_USERNAME=""
DOCKER_PASSWORD=""
file="docker-compose.yml"
composeUrl="https://github.com/docker/compose/releases/download/1.22.0/docker-compose-$(uname -s)-$(uname -m)"


sudo curl composeUrl -fsSL https://get.docker.com/ | sh
sudo curl -L  -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
sudo docker login -u="${DOCKER_USERNAME}" -p="${DOCKER_PASSWORD}"

echo 'version: "3"
services:
  node-server:
    image: j0rd1smit/cab432-scaling-node
    container_name: cab432-scaling-node
    ports:
      - "80:8080"
    environment:
      - "OPENCVHOST=python-lb-1388893829.ap-southeast-2.elb.amazonaws.com"' > "${file}"

sudo docker-compose up
```

Figure 4.2.1.a Deployment script simple requests server

The scaling policy for the simple request server group is relatively easy since it only scales rarely. The group always has at least 1 running instance and has at most 2 instances. The group scales in and out its instances based on the policies in figure 4.2.1.b.

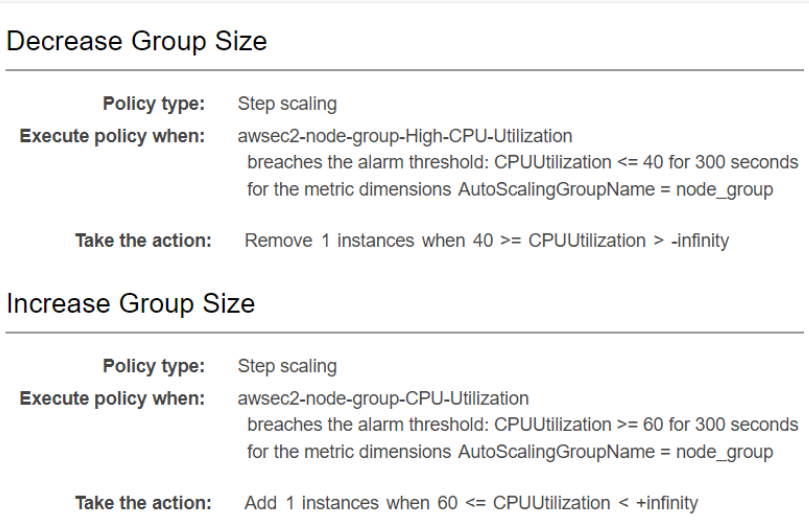


Figure 4.2.1.b Scaling policy for the simple requests server

4.2.2 Image analysis server

The image analysis server instances are created using the script in figure 4.2.2.a. Once the EC2 instance has been initialised the script will start off by installing docker and pulling the required docker image. The script then runs the docker image and exposes its port on the port on which the load balancer is listening.

```
#!/bin/bash
DOCKER_USERNAME=""
DOCKER_PASSWORD=""

sudo curl -fsSL https://get.docker.com/ | sh
sudo docker login -u="${DOCKER_USERNAME}" -p="${DOCKER_PASSWORD}"

sudo docker pull j0rd1smit/cab432-scaling-python
sudo docker run --name=cab432-scaling-python --publish=5000:5000 j0rd1smit/cab432-scaling-python
```

Figure 4.2.2.a Deployment script image analysis server

The image analysis scaling group always has at least 1 running instance and has at most 6 instances (the max amount of parallel HTTP connections for a single browser). The group scales in and out its instances based on the policies in figure 4.2.2.b. The amount of instances that will be added or removed depends on the average CPU utilization such that the scaling rate depends on the current load.

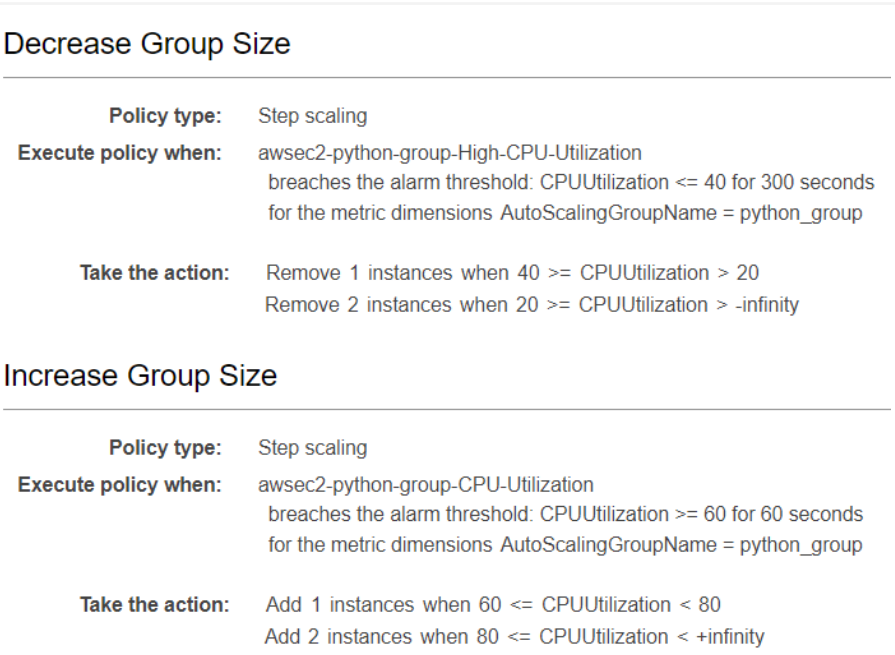


Figure 4.2.2.a Scaling policy for image analysis server

4.2.3 Load balancers

The application has two load balancers. The externally facing load balancer for the simple request server has been created with the settings in figure 4.2.3.a. This server receives requests on port 80 and redirects them to the port 80 on one of the instances. The health check interval has been chosen in such a way that the instances have enough time to initialise.

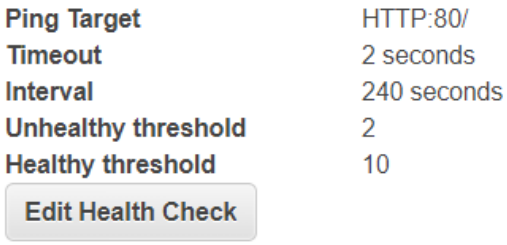


Figure 4.2.3.a Load balancer health check for simple requests server

The internal load balancer for the image analysis server has been created with the settings in figure 4.2.3.b. The parameters have been chosen in the same way as before. The only differences is that this load balancer redirects the traffic to the port 5000 on the server instance.

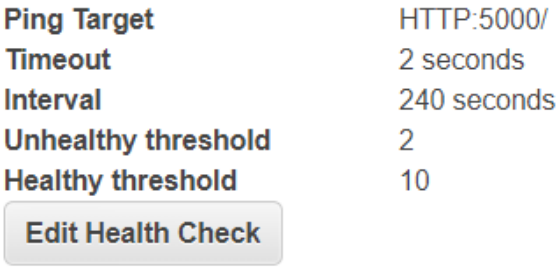


Figure 4.2.3.b Load balancer health check for image analysis server

4.2.4 S3 bucket

The application uses a single S3 bucket called *'cab432-scaling'*. This bucket has been created using the AWS console (Figure 4.2.4.a). The bucket is located in Sydney in order to minimize the data latency.

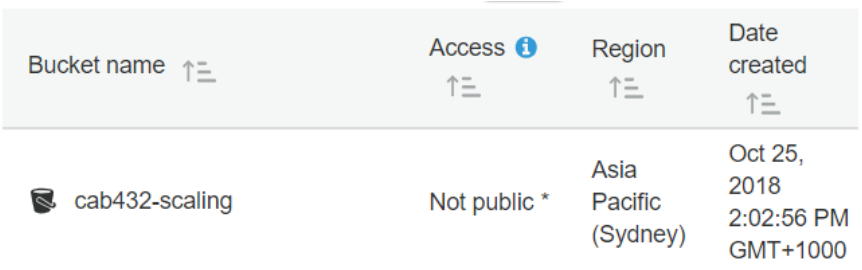


Figure 4.2.4.a The S3 bucket of the project

5 Performance

5.1 Load variations options

We can vary the load we place on the application in two ways. Firstly, we can send more images per batch. This will cause the CPU on the image analysis server to be busy longer with a single requests, sinces it will takes between 100ms and 500ms to run a single through

the neural network. The second way is to send more batches in parallel. Flask servers create a thread for each requests so that they can handle multiple requests at the same time. The image analysis server has to download the image into RAM over the internet before it can analyse it. During the download time the CPU will be idle. So by sending multiple requests in parallel we ensure that the CPU will be busy with the neural network on one of its threads. This will help to keep the average CPU utilization high.

The application will automatically find and send images to the server using Flickr. The user only has to specify how much images per batch to send and how much batches to send in parallel (Figure 5.1.a). The maximum value for the amount of images per batch has been set to 40, which will roughly take a single server 30 seconds to process when handling six batches in parallel (the max load). By setting the maximum to 40 we ensure that we never run into the time out limit of HTTP which occurs at 60 seconds. A normal value would be 5 which will only take a few seconds.
The maximum value for the amount of parallel batches has been set to 6 since most modern browsers can have at most only 6 parallel HTTP connection open.

Figure 5.1.a User input options to vary the load.

5.2 Scalability effects

The application can search through 5000 images in 5 to 10 minutes (whereby 6 parallel batches of 10 images will continuously be send). When we start this search we only have a single image analysis server instance active. The search will quickly move the CPU utilization of this instance to roughly between 80% and 90%. Based on this CPU utilization the scaling policy will add two instances, which brings the total amount of instances up to 3. If you look at figure 5.2.a you see that these instances roughly become active around batch number 330. Before this moment the average response time per batch is roughly 9 second. After this moment the response time drops to roughly 3 seconds. Which means that the load is nearly perfectly distributed over all the parallel instances.

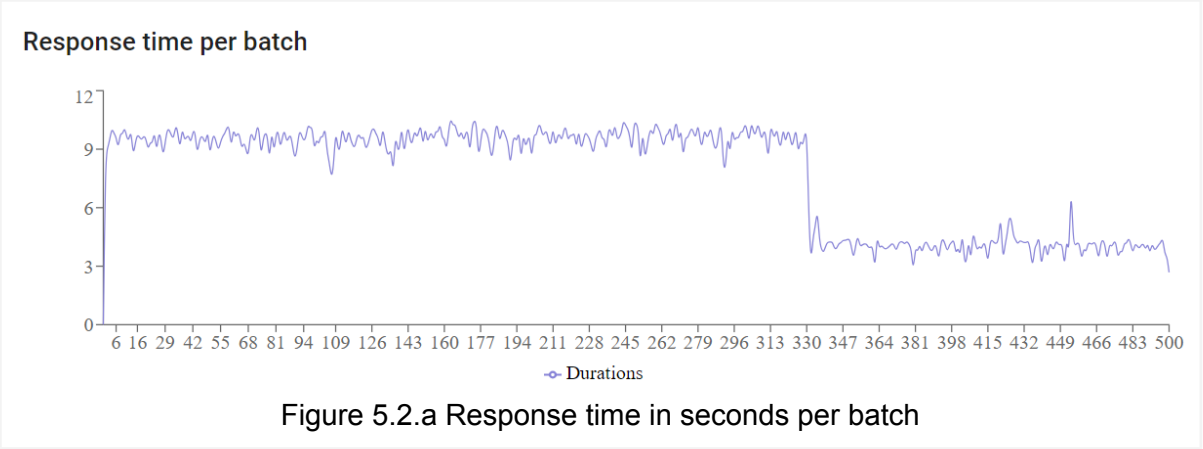


Figure 5.2.a Response time in seconds per batch

6 Testing

The application's architecture depends on multiple sub services. If for some reason one of these sub services is temporarily unavailable we still want the application to function as best as possible. That is why the system has been designed with fault tolerance in mind (6.1) and why it has been tested against specific use cases (6.2).

6.1 Fault tolerance

We introduced fault tolerance into the application by using retry on failure (6.1.1), using non blocking reads/writes (6.1.2), batches queries (6.1.3) and catching error as soon as possible (6.1.4).

6.1.1 Retry on failure

All ajax requests are being performed using a fetch function that on failure will automatically retry with exponential backoff (figure 6.1.1.a). The first time it waits 225 ms then 900 ms then 2025 ms etc. By performing the ajax requests using this method we ensure that a single network error or temporary outage cannot cripple our application. I tested the application for fault tolerance by introducing a bug that rejects 1 out of 10 requests randomly. Even with this bug the application was still useable, although it was a bit slower.

```
export async function fetchWith404(url: string, tries: number, init?: any): Promise<any> {
    if (tries > 3) {
        throw new Error( message: "Max retries is 3");
    }
    const localSignal = signal;
    try {
        await preformFetch(url, init)
    } catch (e) {
        if (tries <= 0 || localSignal.aborted) {
            throw e;
        } else {
            await sleep(Math.pow( x: (3 - tries) * 15,  y: 2));
            return await fetchWith404(url,  tries: tries - 1, init);
        }
    }
}
```

Figure 6.1.1.a Fetch using exponential back off.

6.1.2 None block reads and writes

All reads and writes have been implemented in a non blocking manner (figure 6.1.2). If for some reason S3 will become unavailable, the application will still function. In this scenario the user will not be able to see his previously performed searches or save his current search. All other functionalities will keep working properly.I tested this scenario by turning off S3 in the localstack and as expected the application keeps working properly.

```
public createNew = async (newSearchEntityData: INewSearchEntityData): Promise<ISearchEntity> => {
    const id = `${uuidv4().toString()}`;
    const detections: IOpenCvDetectionResponse[] = [];
    const countPerLabel  = defaultCountPerLabel;
    const countPerImage = defaultCountPerLabel;
    const durations: number[] = [];
    const updateAt = new Date().toString();
    const createAt = new Date().toString();

    const searchEntity = {
        id,
        ... newSearchEntityData,
        countPerLabel,
        countPerImage,
        durations,
        detections,
        updateAt,
        createAt,
    };

    this.save(searchEntity)
        .catch( onrejected: (e: any) => console.log(e));

    return searchEntity;
};
```

Figure 6.1.2 Create a new search entity does not wait for the S3 response

5.1.3 Batch queries

The image analysis are being performed in small batches. These batches make it easier to detect faulty responses. If a batch returns a 40x response or takes to long we can simply resend it and the load balancer will most likely send it to another server. If this fails, we can always drop the batch and try another one as a last resort.
If all the batches return a 40x response the server is most likely temporarily unavailable. In this case the application will wait 5 second and retry again (figure 6.1.3.a). Either way even if the Image analysis servers are unavailable the users can still view his previously performed searches normally.

```
private preformStepBatch = async (): Promise<void> => {
    if (this.state.searchEntity !== undefined) {
        try {
            const searchEntity = await this.runBatch();
            await this.preformAutoSave(searchEntity);
            await this.setStateAsync({
                searchEntity,
                isReadyForNextBatch: searchEntity.reamingPhotoData.length > 0,
            });
            return;

        } catch (e) {
            await sleep( ms: 5000);
            const {searchEntity} = this.state;
            const isReadyForNextBatch = searchEntity !== undefined
                && searchEntity.reamingPhotoData.length > 0;
            await this.setStateAsync({isReadyForNextBatch});
        }
    }
};
```

Figure 6.1.3 Retry after 5000ms if all batch fail

6.1.4 Uncaught error on the client side

It is still possible that something we have not thought about might go wrong. If this happens we don't want the user to see a frozen white screen. Therefor if there is any uncatched error (Figure 6.1.4.a) on the client side the user will be redirected to the 404 page (Figure 6.1.4.b The 404 page).

```
public componentDidCatch(error: any, info: any): void {
    location.href = "/404";
}
```

Figure 6.1.4.a The general error catch

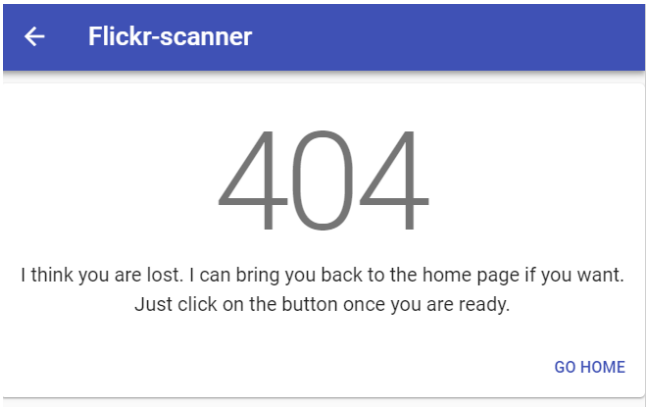


Figure 6.1.4.a The 404 page

6.2 test cases

Task	Expected outcome	Result	Visualization
Create and start a search using the search form.	A search should be create in S3 and the user has been redirected to the search result.	Pass	Appendix I
View the resulting images of a search.	After the user has started a search he should be provided with a list of images that have been analysed.	Pass	Appendix I
Store a search in S3.	After the user saved a search he should be able to close and reopen his browser and result should still be there.	Pass	Appendix I
Delete a search.	After a search has been delete an users should no longer be able to find any remaining data of it.	Pass	Appendix I
Visit a saved search.	After click on the link to a saved search the user should see the exact same data as when he saved it.	Pass	Appendix I
Use the application when S3 is unavailable.	The user should still be able to use all the other features except the persistent features of the application without any problem.	Pass	5.1.2
Use the application when the image analysis server is unavailable.	The user should still be able to use to view saved searches.	Pass	5.1.3
Use the application with varying amount of load.	When the load on the application changes the waiting time for requests should not slow down the application.	Pass	Chapter 5
Auto scaling out with increasing load.	When the load on the application increases the scaling groups should scale out.	Pass	Appendix II
Auto scaling in with decreasing load.	When the load on the application decreases the scaling groups should scale in.	Pass	Appendix I

7 Issues and limitations

During the project I encountered many issues and limitations such as the maximum amount of parallel connection (7.1), slow API response (7.2), long running promises (7.3), OpenCv problems (7.4), presitanted data storage problems (7.5) and classification problems (7.6).

7.1 Maximum parallel TCP connections

When I first started with this project I didn't know the maximum amount of parallel TCP connections a browser could have. I thought it was surely much higher than only 6. That is why the original design was sending each image by itself to the server. I thought I could increase the load by simply sending more requests in parallel. The load that was being created in this way was thus much lower than was expected. So I decided to solve the problem by sending multiple images per requests in order to increase the load.

7.2 Slow Flickr search response

In order to create a large load and keep it up we need thousands of images. Flickr can provide us with a list of pointers to these images. However the main problem is that Flickr only sends at most 250 image pointers per response and these responses take between 1 and 2 seconds.This means that we can at most fetch 750 images per second using all parallel TCP connections.
The main problem with this is that this would temporary decrease the load on the image analysis servers. We can prevent this by prefetching them before the search starts. However this introduces a small waiting time before the search can start. So we have to make a trade off. Because we want to keep the load on the image analysis servers as high as possible we decided to go for the prefetch option.

7.3 Canceling background promise

When we use the maximum load per query we end up with large response times around 20 seconds. This large response time can cause problems for us, since we are using a React single page application. The async response code can change the client side state much later or block other requests, even when its results are no longer needed. So we needed to introduce a way to cancel these long fetches when they are no longer needed. We eventually solved this problem using the native AbortController (“AbortController,” n.d.). We use the it to cancel all the open fetch requests that are no longer needed.

7.4 Opencv does not work on NodeJs

The very first problem I runned into was the fact that OpenCv is very poorly supported for NodeJs. Luckily its very well supported for Python. I quickly found a Python example with a working CNN solution for my problem (Rosebrock, 2017). However I have never created a Python webserver so I had to learn Flask (“Welcome | Flask (A Python Microframework),” n.d.) in a very short time. I managed to create a working web server but it was not as well

designed as my NodeJs server due to my lack of experience with Flask. Luckily this server only has to handle a single type of requests so I managed to get away with my lack of knowledge about Flask.

7.5 Dynamodb could not handle the object size

In my original design, I planned to store the search results into a Dynamodb table. The main problem with this approach is the large amount of data that is being produced during a single search. Dynamodb items have a maximum size of 400KB. The size of a single search result is at least 500 KB and can grow up to 1 MB depending on the amount of images that have been analysed. So the only way to store it in Dynamodb was to normalize the data but this would result into a very high burst of reads and writes (which can be very costly). So eventually I decided to store the search results in JSON files in S3. This way we didn't have to take the data size into account.

7.6 Neural network misclassified

There are many pre trained CNNs available for OpenCv. However much of these CNNs can be very slow. So I had to make a trade off between speed and accuracy, since this CNN has to run on a web server and should return a response in reasonable amount of time. The CNN (Rosebrock, 2017) I ended up using is pretty fast (100 ms to 500 ms depending on the image) but is accuracy is not always as great. For example it sometimes confuses drawings of people and real people (Figure 7.6.a).

Figure 7.6.a Incorrect classification

8 Possible extensions

During the creation of this project 2 possible extensions came to mind. Firstly, provided the users with more freedom by allowing them to upload their own images (8.1). Secondly, change the data schema in order to allow for user authentication (8.2).

8.1 Upload your own pictures

One possible extension to the application is to allow users to upload their own pictures. This would give users much more freedom. This feature could also give the user the ability to get statistical insight in the images he or she has taken over the years.
The extension can easily be created by adding an additional endpoint to the rest API of the image analysis server.

8.2 User authentication

Currently the application shows all the saved search results and anyone can see or edit any search result. User authentication was outside the scope of this project, however user authentication is a must before this application can be used by multiple users.

Bibliography

AbortController. (n.d.). Retrieved November 4, 2018, from https://developer.mozilla.org/en-US/docs/Web/API/AbortController

Leaflet — an open-source JavaScript library for interactive maps. (n.d.). Retrieved November 1, 2018, from https://leafletjs.com/

localstack. (n.d.). localstack/localstack. Retrieved November 1, 2018, from https://github.com/localstack/localstack

React – A JavaScript library for building user interfaces. (n.d.). Retrieved August 23, 2018, from https://reactjs.org/index.html

Recharts. (n.d.). Retrieved October 8, 2018, from http://recharts.org

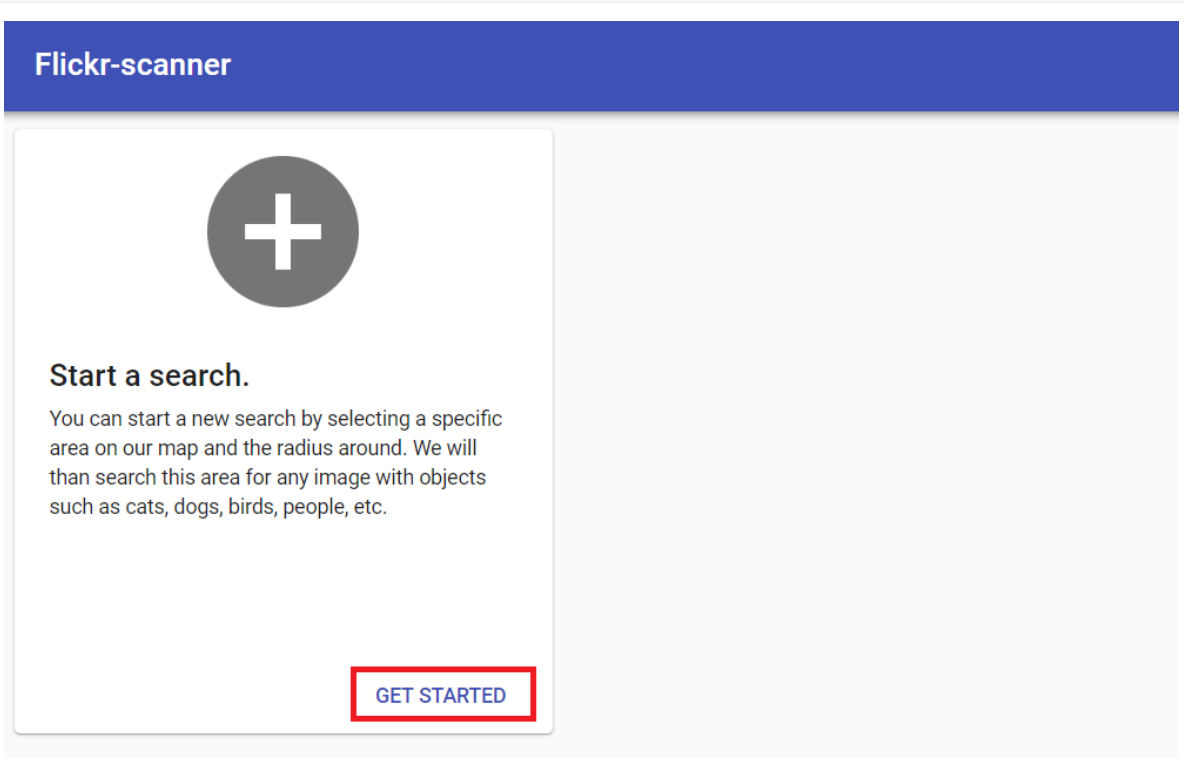
Rosebrock, A. (2017, September 11). Object detection with deep learning and OpenCV - PyImageSearch. Retrieved October 8, 2018, from https://www.pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/

Website. (n.d.). Retrieved November 1, 2018, from https://blackboard.qut.edu.au/bbcswebdav/pid-7583513-dt-content-rid-19231750_1/courses/CAB432_18se2/Assessment2Cloud%282%29.pdf

Welcome | Flask (A Python Microframework). (n.d.). Retrieved November 4, 2018, from http://flask.pocoo.org/

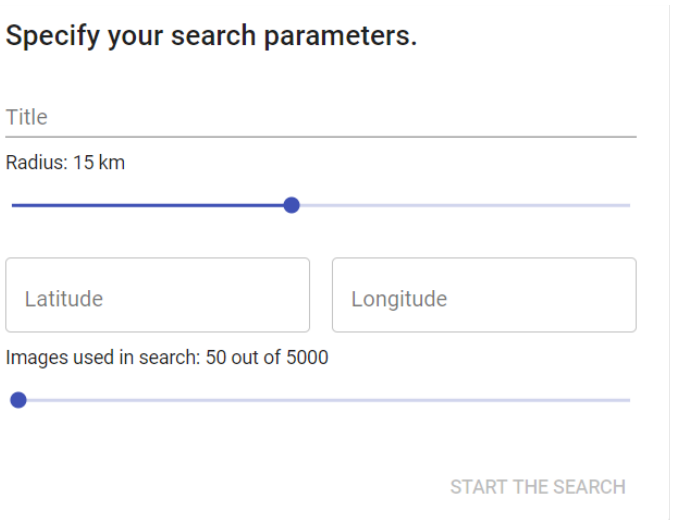
Appendix I User guide

Step 1 Go to search form

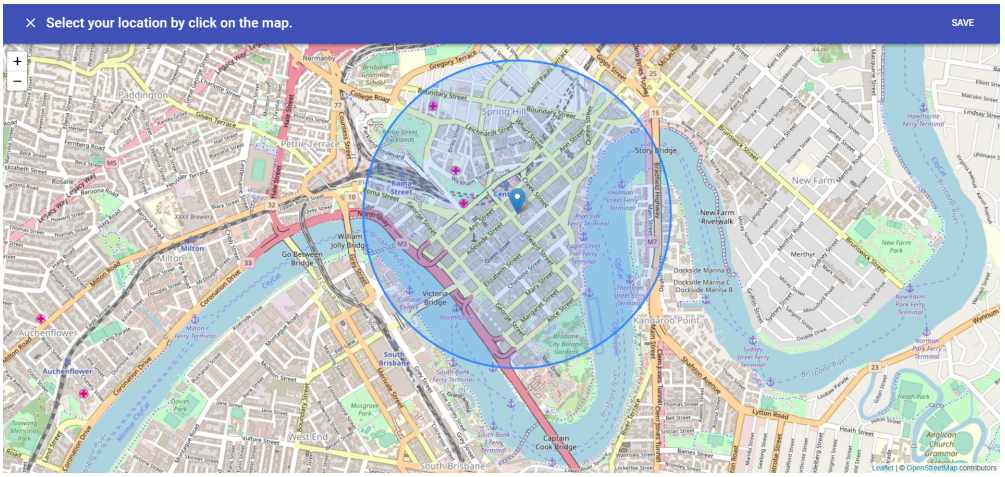


Click on the get started button.

Step 2 Select a search area

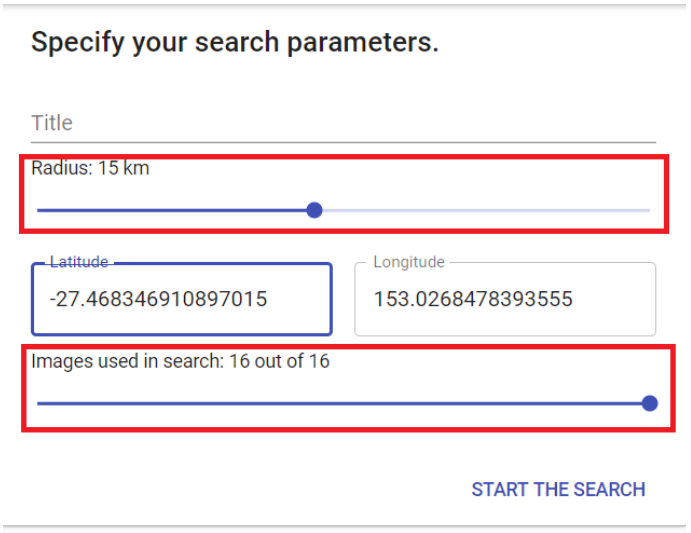


Click on the latitude or longitude field



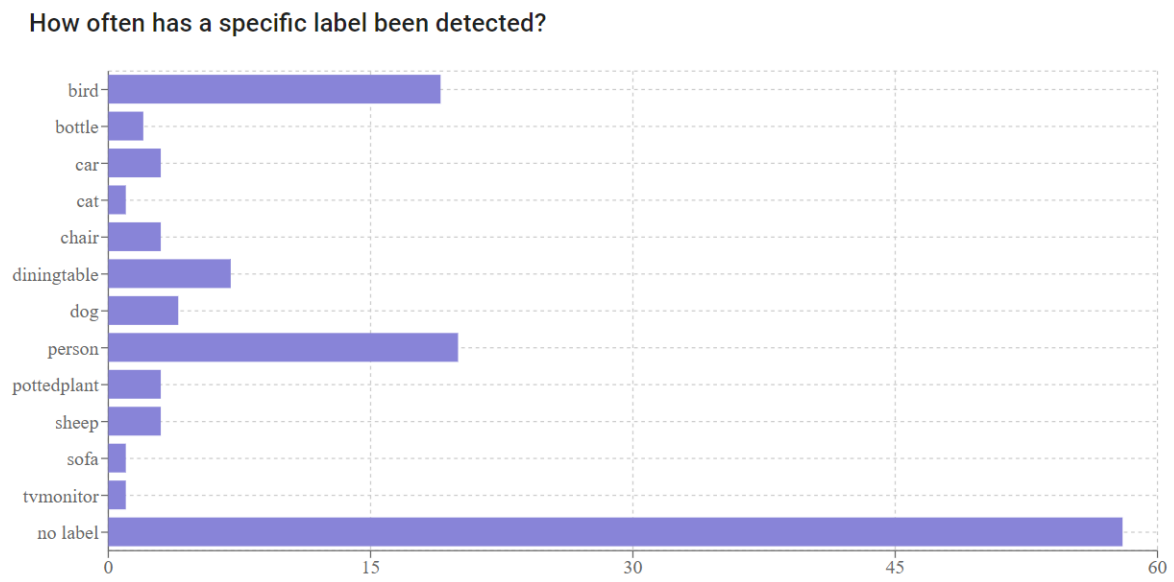
Select a search area by clicking on the map. Then save it using the save button in the top right corner.

Step 3 Specify a search radius and the amount of images and start the search



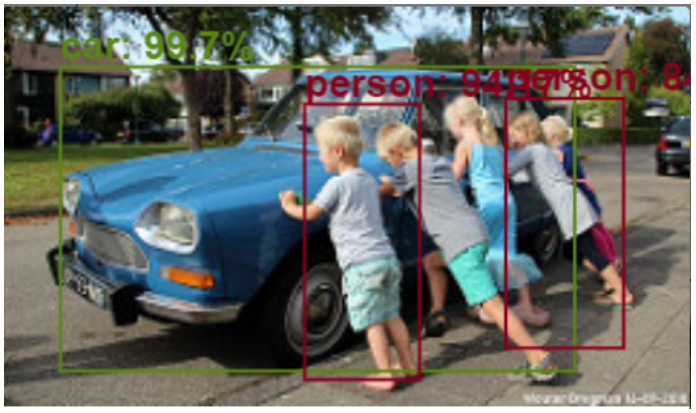
Drag the slider for the radius and amount of image to the desired values and click on start the search button.

Step 5 Let the search run while looking at the search results



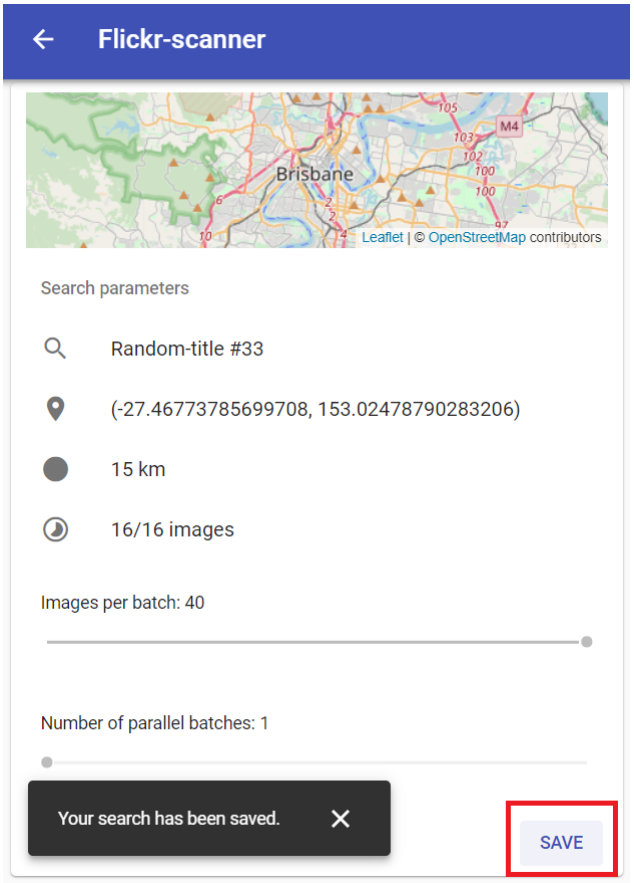
A summary of the results

The results for each image.



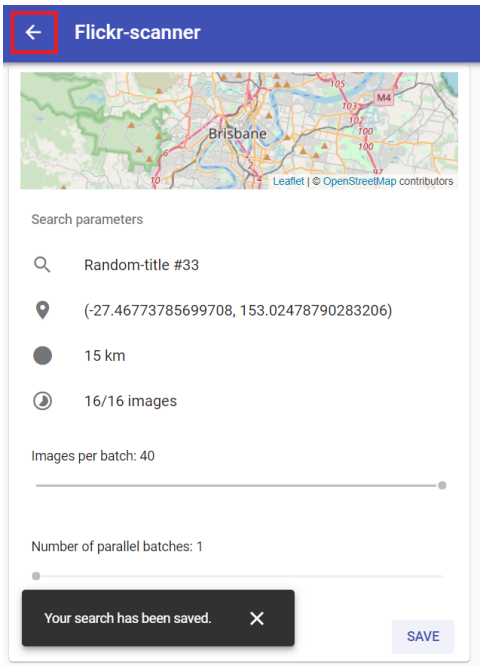
The results for specific image.

Step 6 Save the results

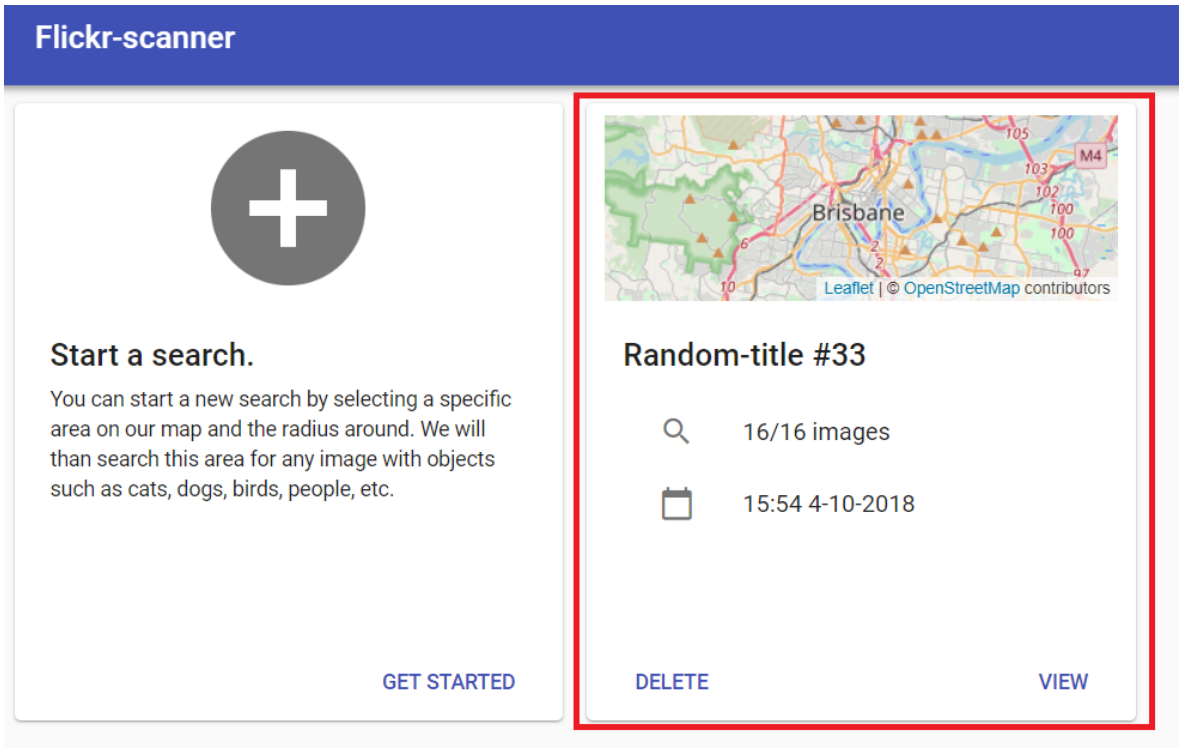


Save the results by clicking on save button.

Step 6 View the saved searches.

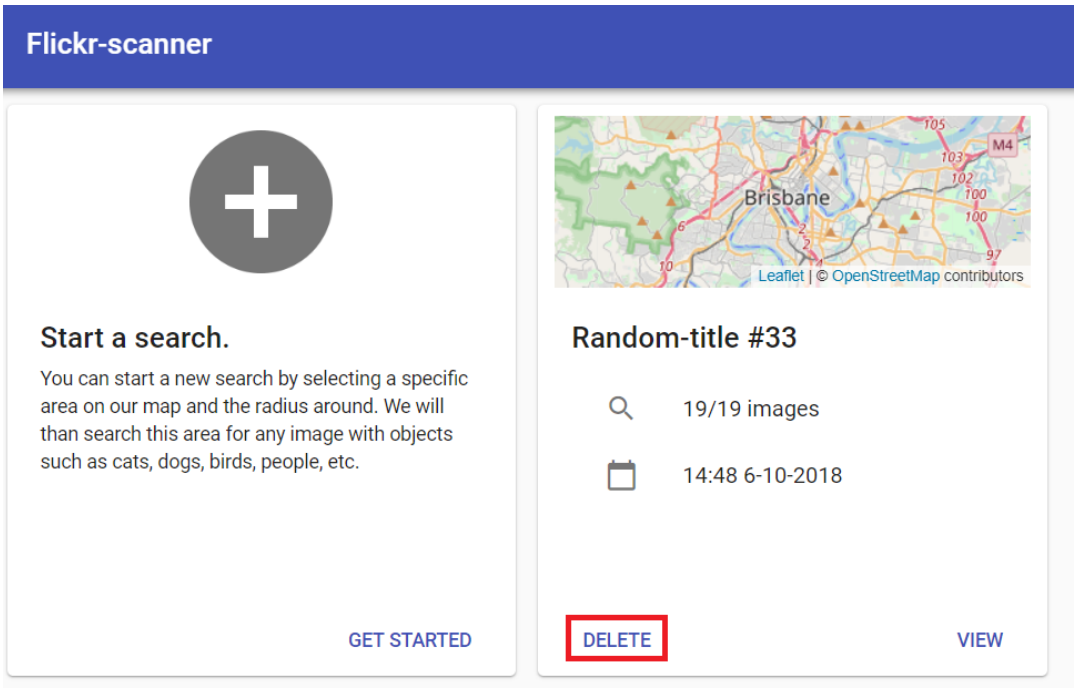


Navigate back to the index page using the button in the top left.

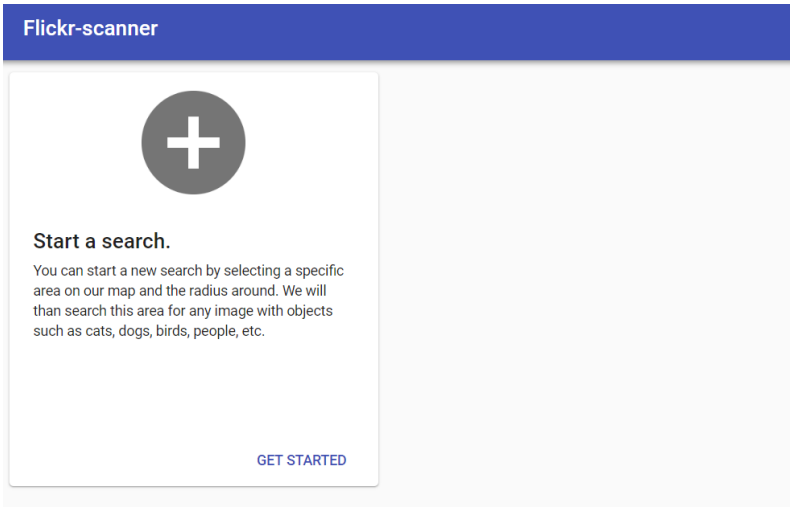


The search overview now contains the saved search result.

Step 6 Delete the saved searches.

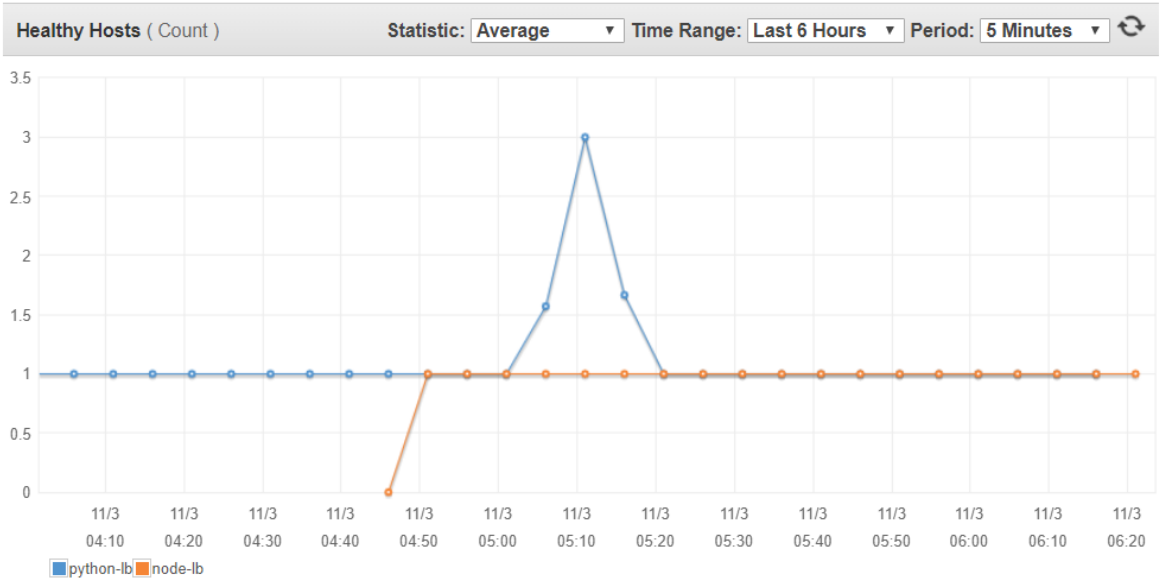


Delete the search using the delete button



The search overview now no longer contains the saved search result.

Appendix II Cloud watch graphs



Amount of instances in load balancer during search

Status	Description	Start Time
Successful	Launching a new EC2 instance: i-058e67edc42e575d3	2018 November 3 13:37:59 UTC+10
Successful	Launching a new EC2 instance: i-0613638492d611f52	2018 November 3 13:34:55 UTC+10
Successful	Launching a new EC2 instance: i-0eced36364f926fdc	2018 November 3 15:00:51 UTC+10
Successful	Launching a new EC2 instance: i-0fadec454b4f1cff7	2018 November 3 15:00:51 UTC+10
Successful	Terminating EC2 instance: i-058e67edc42e575d3	2018 November 3 15:14:33 UTC+10
Successful	Terminating EC2 instance: i-0613638492d611f52	2018 November 3 13:44:33 UTC+10
Successful	Terminating EC2 instance: i-0eced36364f926fdc	2018 November 3 15:15:34 UTC+10

Auto scaling Image analysis server instance. A search cause the creation of 2 more instances.

Status	Description	Start Time
Successful	Launching a new EC2 instance: i-0752cdfa0e66b70c7	2018 November 3 14:45:26 UTC+10

Auto simple request server instances. A search can be done with a single instances.

Appendix III Deployment instructions

API keys

The application need two API keys which should be stored in json format in credentials folder:

- flickrCredentials.json
- awsConfig.json

```
{
  "api_key": "",
  "secret" : ""
}
```

Example flickrCredentials.json file

```
{
  "accessKeyId": "",
  "secretAccessKey": "",
  "region": ""
}
```

Example awsConfig.json file

Deployment steps UI server

Step 1: Add the API keys to the credentials folder.

Step 2: Build the docker image using:

```
docker build -t <IMAGE_NAME> .
```

Step 3: Start the server using:

```
version: "3"
services:
  node-server:
    image: <ImageName>
    container_name: <ContainerName>
    ports:
      - "80:8080"
    environment:
      - "OPENCVHOST=<ImageAnalysisUrl>"
```

Docker Compose Start up

Deployment steps Image analysis server
Step 1: Build the docker image using:

```
docker build -t <IMAGE_NAME> .
```

Step 2: Start the server using:

```
docker run --name=<CONTAINER_NAME> --publish=<PORT>:5000
<IMAGE_NAME>
```

Deployment steps S3
Step 1: Create a S3 bucket called cab-scaling