2023

Cloud Project

11/6/2023

Contents

Introduction

Purpose & description

The purpose of this application is to give users the ability to understand traffic density in Queensland Australia by connecting them to the QLD traffic API, in a familiar google-maps styled interface, to help them plan their travel.

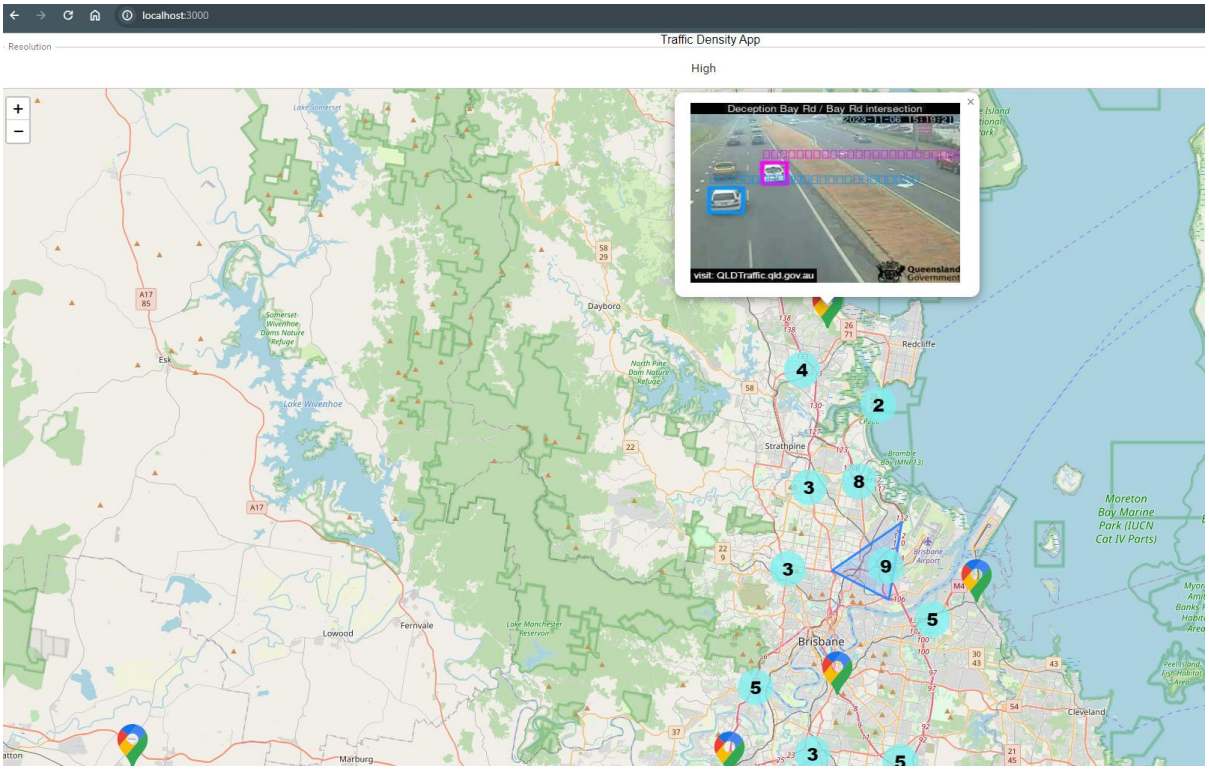


Figure 1 Traffic Densit App

This application takes still images from the QLD government traffic camera API at fixed intervals of 1 minute and applies computer vision classification to those images to count cars. The web application displays a map to the user pins denoting the locations of cameras and a popup on each pin displays the processed image including any bounding boxes showing that cars have been detected.

Services Used

QLD Traffic API

[QLD Traffic Web Camera API Documentation]

This api endpoint provides still images from various traffic cameras scattered around QLD.
Here is the API endpoint: https://api.qldtraffic.qld.gov.au/v1/webcams
To use this api you can use a public api key which is a limited number of uses or register for a key. I have registered for a key and that is what is in my assignment. This API returns some static data in a JSON format this contains a url for an image taken by a traffic camera in the last minute and is the fuel for the load generation of this project.

React-Leaflet

[React-Leaflet Documentation]

React-leaflet is a set of UI components to make a google-maps style interactive map. It includes elements such as markers, popups and clusters which I make use of in this project.

See appendix, image a) for an example from the documentation.

Tensorflow coco-ssd

[Tensorflow coco-ssd Documentation]

This api is a service that offers a node.js implementation of the Common Object in Context – Single Shot multi-box detection model. This allows me to supply it with an image and in exchange for some cpu time it will provide me with predictions of the classes of objects contained in the supplied image and the geometry required to draw a bounding box for each class. I use this to find cars in images from the QLD traffic API and then draw bounding boxes on those images.

Use cases

US 1

As a	Person who travels long distances by road
I want	I want to be able to see at a glance how dense the traffic is in a given area.
So that	So that I can better plan my routes and travel more efficiently by potentially avoiding some high traffic zones.

This element, ultimately, was difficult to implement, peripheral to the primary requirements of the project and not really supported by the API I was using for traffic information due to limited cameras of low image quality.

I did manage to collect the predictions but failed to convert them into any kind of density metric, nor any kind of representation on the map see below image:

Figure 2 Unrealized user story

It didn't seem like a good use of time to pursue this user story to its conclusion with the time constraints I was working with.

US 2

As a	As a regular commuter who is travelling home for the day
I want	I want to visually assess how bad traffic is at a specific place that I must go or along my route home.
So that	This helps me develop a sense of how long it will take to travel home allowing me to communicate my availability to family/friends expecting me.

I was able to implement this element. For each pin on the map the pop up for that pin shows either the original image from the traffic camera or the annotated image indicating detected cars. See below (and please ignore the persistent font issue):

Figure 3 A traffic image displayed as a popup

Technical breakdown

Architecture

The way this application is structured in broad terms is as follows.

There is a client-side server hosting a react js web application and back-end. Then there is an autoscaling group of worker nodes that are sent jobs to process by the back-end server.

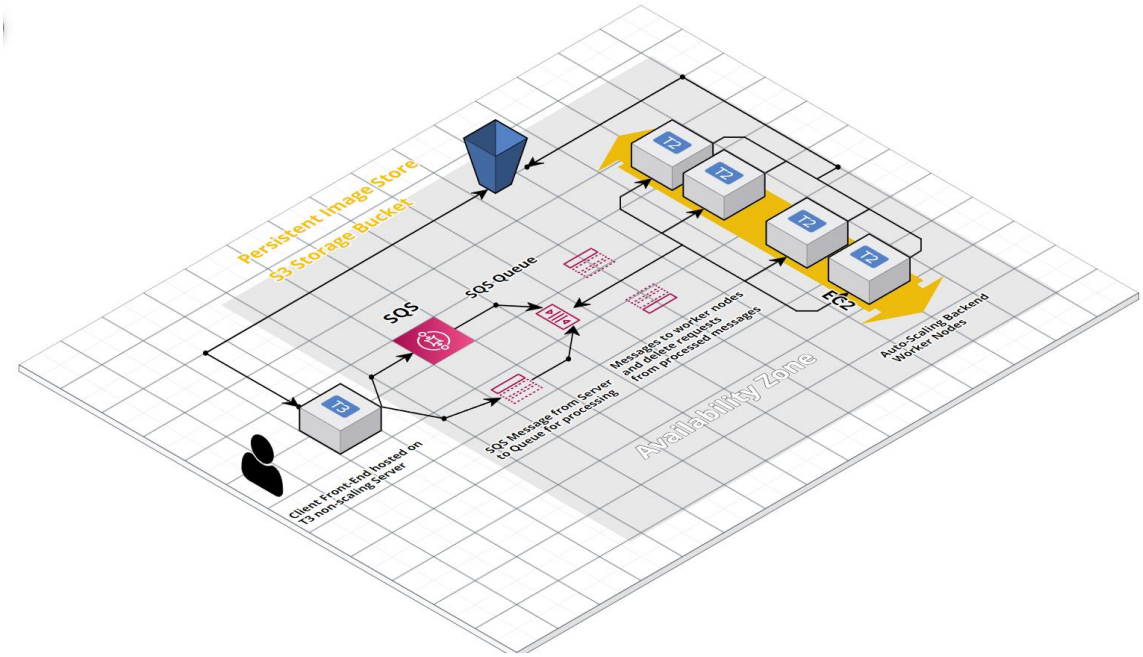


Figure 4 Architecture of the Traffic Density App

Client / server demarcation of responsibilities

First there is a user facing T3.Micro Instance which hosts an express.js server.
That server hosts a react.js single page web application and a server API that feeds the web application. Pictured below is the app.js file. Highlighted is the usage of the static deployment of the react.js front-end app.

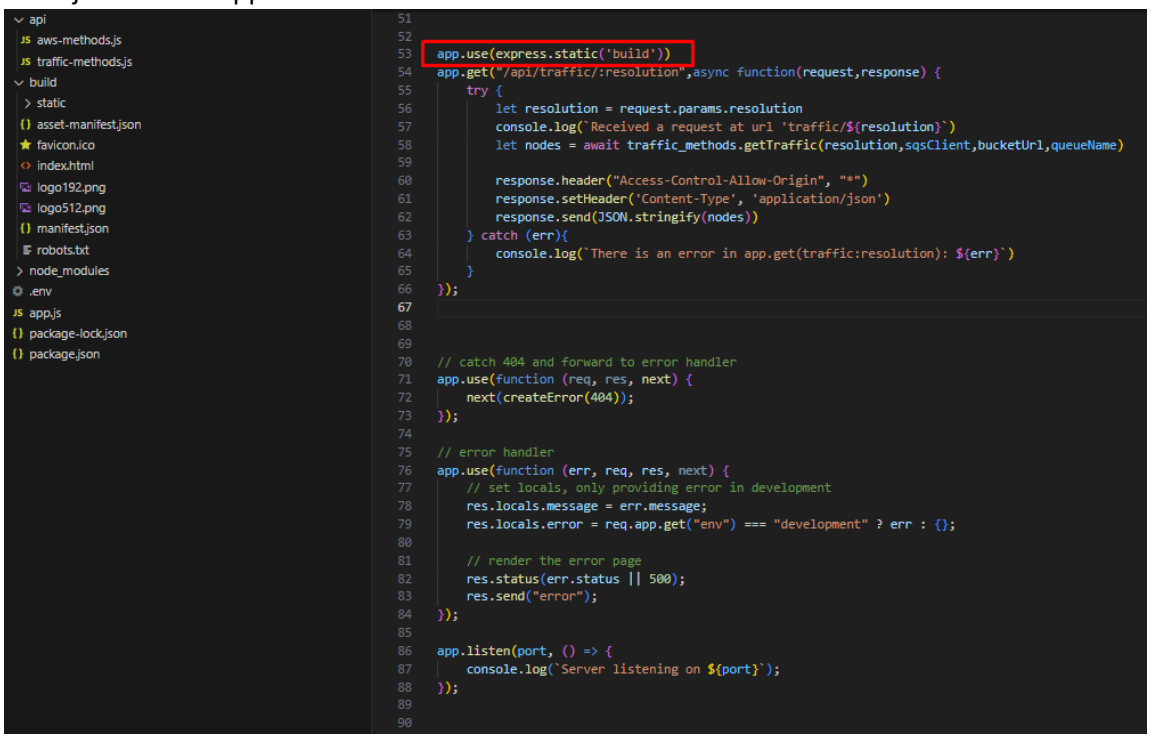


Figure 5 Highlighted static react.js app in Express Back-end

The server makes API calls to the QLD traffic API to fetch JSON data and, crucially, image URLs. This JSON object is reformatted into something more usable. For each resolution a certain number of

retrieved “trafficNodes” are sent for processing. For “Low” resolution 5 nodes are sent, for “Medium” resolution 30 nodes are sent and for “High” resolution all nodes are sent.
The image URLs are sent as messages using AWS SQS to a queue that is accessed elsewhere (see below).

```
function trafficNode(data){
    let node = {
        id:data['properties']['id'],
        geometryType: data['geometry']['type'],
        longitude: data['geometry']['coordinates'][0],
        latitude: data['geometry']['coordinates'][1],
        description:data['properties']['description'],
        suburb:data['properties']['locality'],
        postcode:data['properties']['postcode'],
        image:data['properties']['image_url'],
    }
    return node
}
```

```
async function getTraffic(resolution,sqsClient,bucketUrl,queueName){
    let trafficKey = process.env.PRIVATE_TRAFFIC_KEY
    let hostName = "https://api.qldtraffic.qld.gov.au"
    let requestURL = `${hostName}/v1/webcams?apikey=${trafficKey}`

    console.log(`Accessing url: ${requestURL}`)
    const result = await axios.get(requestURL);
    const queueUrl = await GetQueueUrl(queueName,sqsClient)

    if(result.status ===200){
        let events = result.data.features
        console.log(`Status code: ${result.status} reponse is ${result.statusText} with ${events.length

        let resolutionLookup = {
            Low:5,
            Medium:30,
            High: events.length
        }
        events = _.sampleSize(events,resolutionLookup[resolution])
        let nodes = events.map((event=>{
            try{

                let node = trafficNode(event)
                node.id = createId(node)
                SendMessageToQueue(node.image,node.id,queueUrl,sqsClient)
                node.image = bucketUrl+node.id+'.jpg'
                return node
            } catch(err){
                console.log(`There was an error mapping over the nodes in getTraffic(): ${err}`)
            }
        }))
        console.log(`Added ${nodes.length} jobs to processing queue!`)
        return nodes

    } else{
        console.log(`Status code: ${result.status} reponse is ${result.statusText}`)

    }
}
```

Figure 6 How traffic data is retrieved and handled.

The queue is then accessed by a worker T2.Micro Instance in an Auto-Scaling Group. The worker awaits new messages from the queue, collecting a small batch of 10 (the maximum) to work on. For each URL sent in the queue the image is downloaded and cached in an s3 bucket as a placeholder (see below). (The S3 bucket is instantiated by the client back-end server).

```
(async ()=>{
  const bucketName = "[redacted]-s3-cloud-project";
  const queueName = "[redacted]sqs-cloud-project"

  const configObject = {
    region: "ap-southeast-2" ,
    // credentials:{
    //   accessKeyId: process.env.AWS_ACCESS_KEY_ID,
    //   secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
    //   sessionToken: process.env.AWS_SESSION_TOKEN,
    // }
  }

  try{
    const sqsClient = new SQSClient({region:configObject.region});
    const s3Client = new S3Client({region:configObject.region});
    const queueUrl = await GetQueueUrl(queueName,sqsClient)
    let model = await cocoSsd.load()

    setInterval(async ()=>{
      console.log(`Commencing next run..`)
      await PollingQueue(queueUrl,bucketName,sqsClient,s3Client,model)
    },5000);

  } catch(err){
      console.log(`Error in main function: ${err}`)
  }

})();
```

```
async function PollingQueue(queueUrl,bucketName,sqsClient,s3Client,model){
  // Set Interval wrap this stuff:
  let result = await ReceiveMessagesFromQueue(queueUrl,sqsClient)

  if(result.Messages !== undefined){
    console.log(`${result.Messages.length} messages were retrieved`)
    result.Messages.forEach(async (element) => {
      let jobName = element.MessageAttributes.jobId.StringValue+'.jpg'
      let jobUrl = element.Body
      await DeleteObjectFromS3(bucketName,jobName,s3Client)
      let image = await DownloadImage(jobUrl);
      await PutObjectInS3(bucketName,jobName,image,'image/jpeg',s3Client)
      let outputImage = await Run(image,model);
      await PutObjectInS3(bucketName,jobName,outputImage,'image/jpeg',s3Client)
      await DeleteMessageFromQueue(element.ReceiptHandle,queueUrl,sqsClient)
    });
  }
}
```

Figure 7 Worker node processing queue messages.

Then, that same image is passed to the Tensorflow model for processing (see below).

```
async function Run(image,model){

    let prediction = await Predict(image,model)
    let outputImage = await GenerateCanvasImage(image,prediction.tfImage.shape[1],prediction.tfImage.shape[0],prediction.predictions)
    outputName = "output"
    console.log(`Attempting to write update image with type ${typeof outputImage} to file...`)

    if(outputImage){
        return outputImage
    }

}
```

```
async function Predict(image,model){
//    let model;
  console.log("Commencing Prediction...")
  try{
    image = tf.node.decodeImage(new Uint8Array(image), 3)
  } catch (err) {
    console.log(`There was an error in predict() at the point of loading the model: ${err}`)
  }
  if (image){
    try {
      let predictions = await model.detect(image)
      console.log(`Returning the predictions:`)
      console.log(predictions)
      return {predictions: predictions, tfImage:image}
    } catch (err) {
      console.log(`There was an error in predict() at the point of generating the predictions: ${err}`)
    }
  }
}
```

Figure 8 Tensorflow model making predictions.

The image is then marked up with a bounding box as detected by the Tensorflow predictions. It is uploaded to the s3 bucket, overriding the cached image. Then the message that triggered this activity on the worker is deleted from the queue.

The front end react.js web application is pulling using a programmatically generated url to access whatever image belongs to that url at the time, be it a place holder or a marked-up image.

```
events = _.sampleSize(events,resolutionLookup[resolution])
let nodes = events.map((event=>{
    try{

        let node = trafficNode(event)
        node.id = createId(node)
        SendMessageToQueue(node.image,node.id,queueUrl,sqsClient)
        node.image = bucketUrl+node.id+'.jpg'
        return node
    } catch(err){
        console.log(`There was an error mapping over the nodes in getTraffic(): ${err}`)
    }
}))
```

Figure 9 Programmatic URL generation

Response filtering / data object correlation

In Figure 6 we can see how the JSON response from the QLD Traffic API is processed into a more convenient form. See below in Figure 10. the before and after processing of the traffic data.

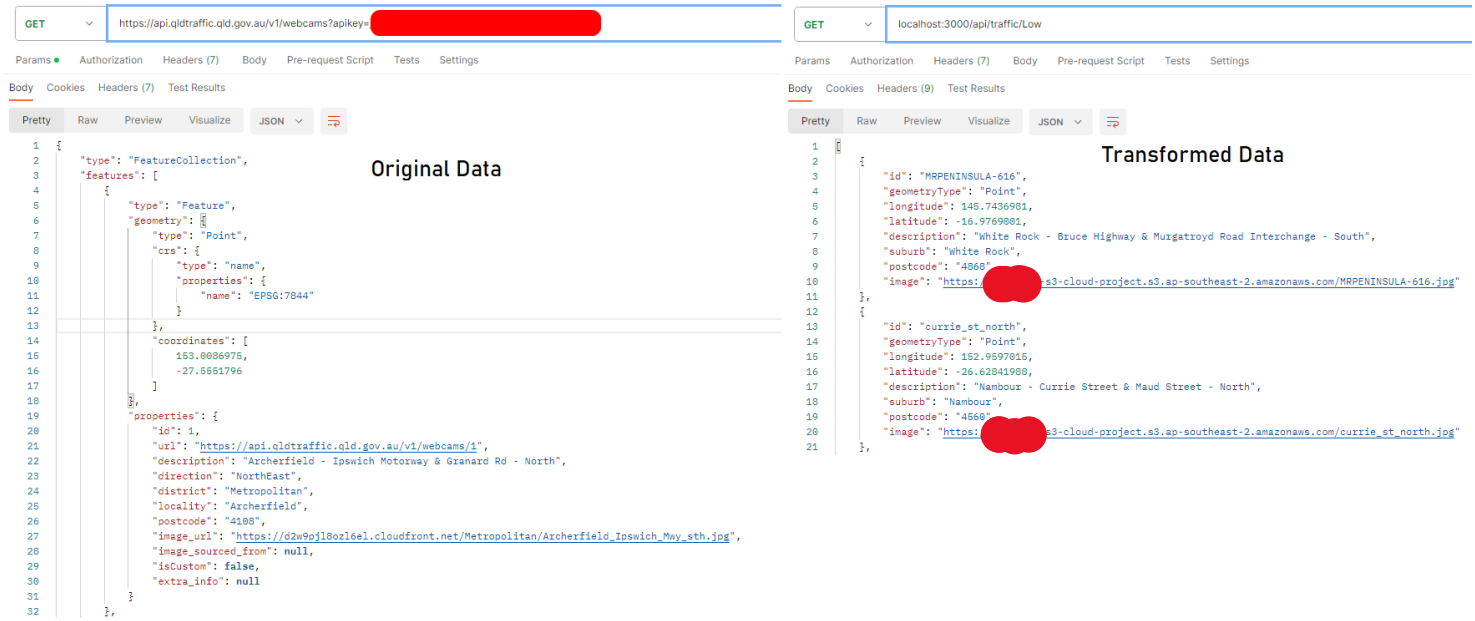


Figure 10 Before and After data transformation

Scaling and Performance

The QLD Traffic API returned at most 160 items. With updates to the image at the Url every minute. This means that I am automatically limited to 160 items of processing per minute. The react.js front end automatically requests more processing every minute generating jobs in the queue.

In my own local tests, it took a few seconds to process an image with Tensorflow. I could see the potential for generating significant load on a less powerful machine if I can dump 300-400 seconds of work on it in a single hit, with another 300-400 seconds of work to come a minute later.

See below the number of jobs thrown into the queue for processing:

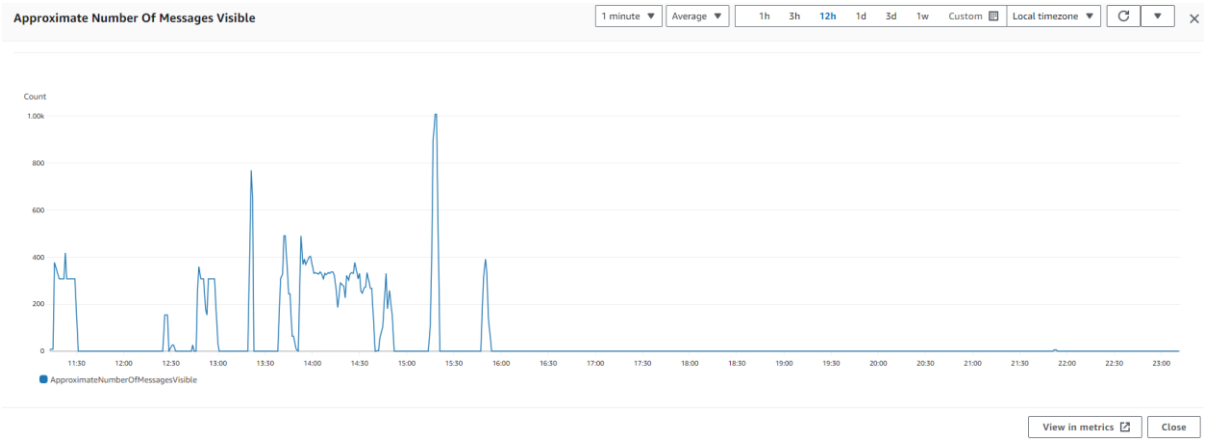


Figure 11 Queue potential message load.

For the following settings:

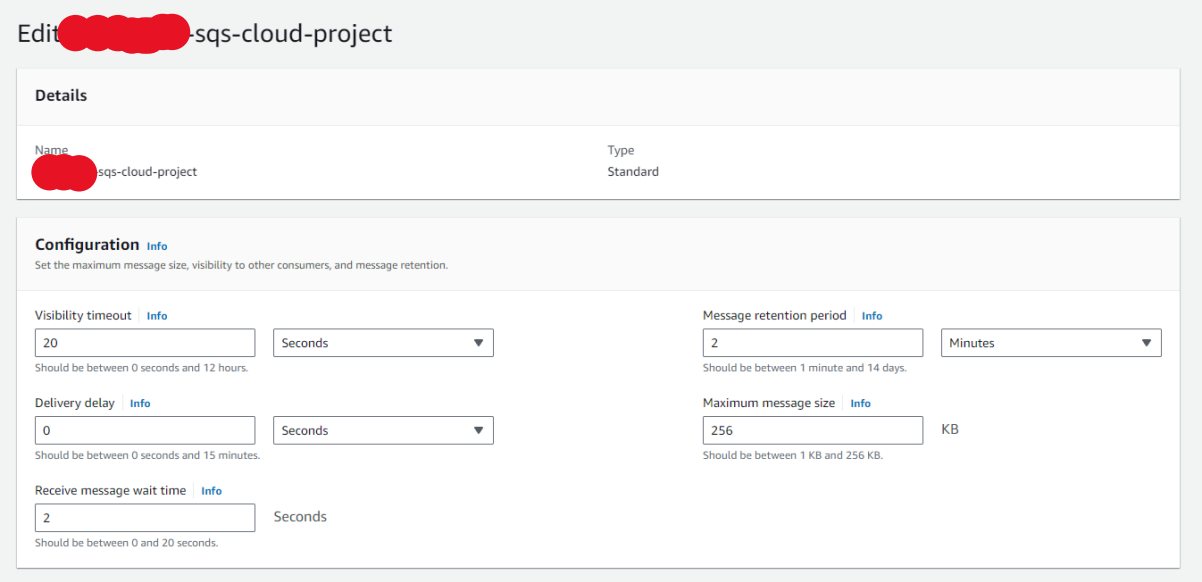


Figure 12 Queue Settings

In my auto-scaling group I have setup several alarms. Including the required 50% persistent CPU load see below:

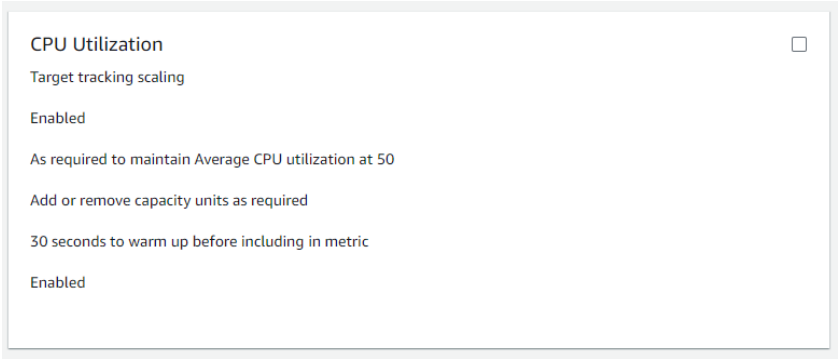


Figure 13 Authorized Policies

but also, additional unauthorized metrics inspired by various comments by the teaching team and reacting to my specific computation needs. These policies are designed to capture situations where a minute's worth of traffic data is not getting completely processed. If up to 160 nodes cannot be

processed in a minute, more will accumulate, and scaling will be required as load continues to increase.

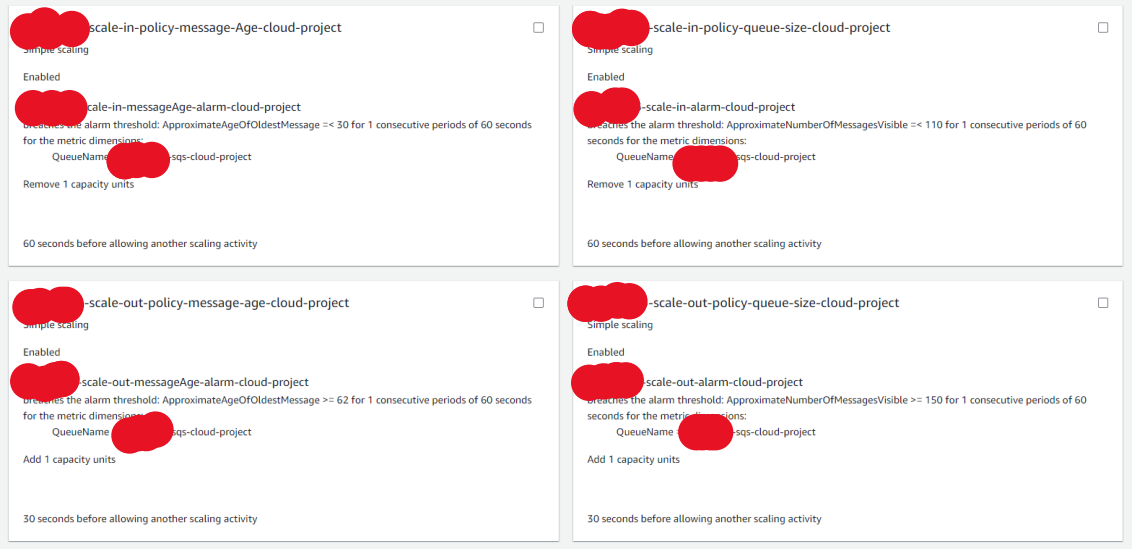


Figure 14 Unauthorized Policies

Below is a scaling even triggered by the default policy:

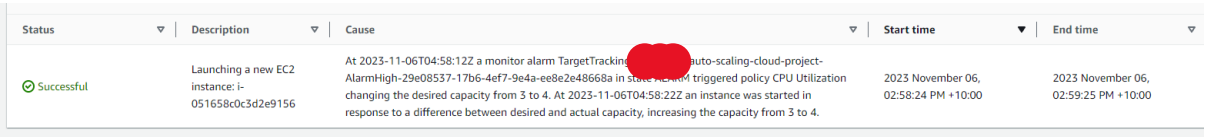


Figure 15 New instance created due to CPU load

Below is evidence of scaling in and out in response to the load generated by the application. This is a graph of the average CPU utilization across all of the instances in the group (please see dashboard dashboard-cloud-project for query). For scale out I set the threshold to be 30 seconds of sustained load, for scale in I would set the threshold to be 60 seconds of reduce load. The result of this is several skewed pyramid shapes in the below image. See also the appendix images b) and c) for the same scaling scenario with message count and message age against instance count.

Figure 16 Scaling vs CPU Util

Test plan

Task	Expected Outcome	Result	Figure/Appendix
Make request to traffic API	JSON Response	Success	Fig 10
Process Image with Cocossd	Array of predictions	Success	Fig 2
Generate Bounding Box	Image will be generated with labels and coloured bounding boxes.	Success (later partial success due to font issues with bounding box generation on vm)	Appendix d)
Make map	React-js single page map app	Success	Fig 1
Transform API request	Streamlined working data node	Success	Fig 10
Pass traffic data to react.js map	Map receives traffic data and renders it	Success	Appendix e) Fig 1.
Pass traffic data to queue	Successfully send data to queue and see message count in aws	Success	Fig 11, Appendix f)
Receive message from queue	See message and url contents.	Success	Appendix g)

Process Image from queue and delete message out of queue at completion	Write placeholder to S3, get predictions, generate bounding boxes, upload new image to S3	Success	Appendix g)
Auto Scaling worker group based on various metrics	Scaling in and out.	Success, but used multiple metrics to trigger scaling in/out	Fig 16.

Difficulties / Exclusions / unresolved & persistent errors /

Roadblocks

1. Working with canvas, javascripts fs module, buffer objects, filestream objects -> Trying to turn a string url, download that image, edit that image and then turn it back into a format that something else can download took a lot of trial and error and reading obscure tutorials about dataTypes pertaining to file storage and transmission.
2. React and useState(), useEffect(). I created a function that would fetch traffic data from the client back-end in response to a "resolution" option being selected in a drop down box. This would constantly fail and set itself to null or refuse to search for the value I had just selected, but the value that was previously selected. It all got a bit worse once I decided that I want to fetch new images every minute. So, I included a setInterval(function,milliseconds) to my useEffect function and that would introduce more issues.
3. Transferring over to AWS-SDK v3 from AWS v2 used in practical sessions. Updating the way credentials are delivered and all my methods for interacting with S3 Buckets wasn't very fun. The official v3 documentation is rich in some ways but it's too easy to fall into rabbit holes and the examples don't show enough. Attempting to google the issues leads to a myriad of shallow AWS "documentation" pages which only talk about the concepts relating to what you want and if there is any code it's in java or it's using the javascript v2 SDK in it's examples whilst helpfully pointing you to the new v3 documentation that is example-lite.
4. Cross-Origin Policy. I was having issues making API requests from my react.js app to the QLD traffic API through the client back-end. I had one of these hosted on port 3000 and the other on port 3001. This was solved by building the react app and loading self-contained deployment as a route on my express server.

Unfinished Work

One of the use cases of this task was to attempt to give glance value to users to indicate the traffic density of the area. I wanted to implement a system of normalized traffic density where after collecting the newest batch of images I would also receive the latest batch of counts of each class detected in the image. The intention was to count all instances of cars/trucks/buses etc in all images and use that value to normalize each count. Then this would somehow connect to the clustering feature of the markers so instead of the cluster displaying the counter of markers that are contained

in the cluster it would display the traffic density of the cluster. This was abandoned since it was a “nice to have” on top of the main functions added.

Differences from the original proposal

There are numerous differences from the original proposal based on the feedback I was given. Initially I wanted to pursue some kind of key,value store approach where I would pass processed image URLs and counts via redis/elasticache. My feedback rightly pointed out some issues with this and upon stewing on it for some time I bit the bullet and tried to learn how to use SQS to replace it.

Outstanding Bugs

1. The issues with useEffect and useState to update the “resolution” variable persist. The front end eventually becomes unstable after minutes of running and eventually crashes to the point of needed a refresh.
2. Not so much a bug but the Tensorflow CocoSsd model was not very good for these images. The images are quite low resolution and there isn’t enough detail for the Tensorflow model to detect cars a lot of the time. I tested with images of complete gridlock, and it would find only a single digit number of cars in the image.
3. There was an issue moving from Windows Linux Subsystem Ubuntu to AWS t2.micro Ubuntu. From the point of deployment onwards my labels for my bounding boxes return strange characters instead of the required label and score. I couldn’t make the time to fix this glaring cosmetic issue.

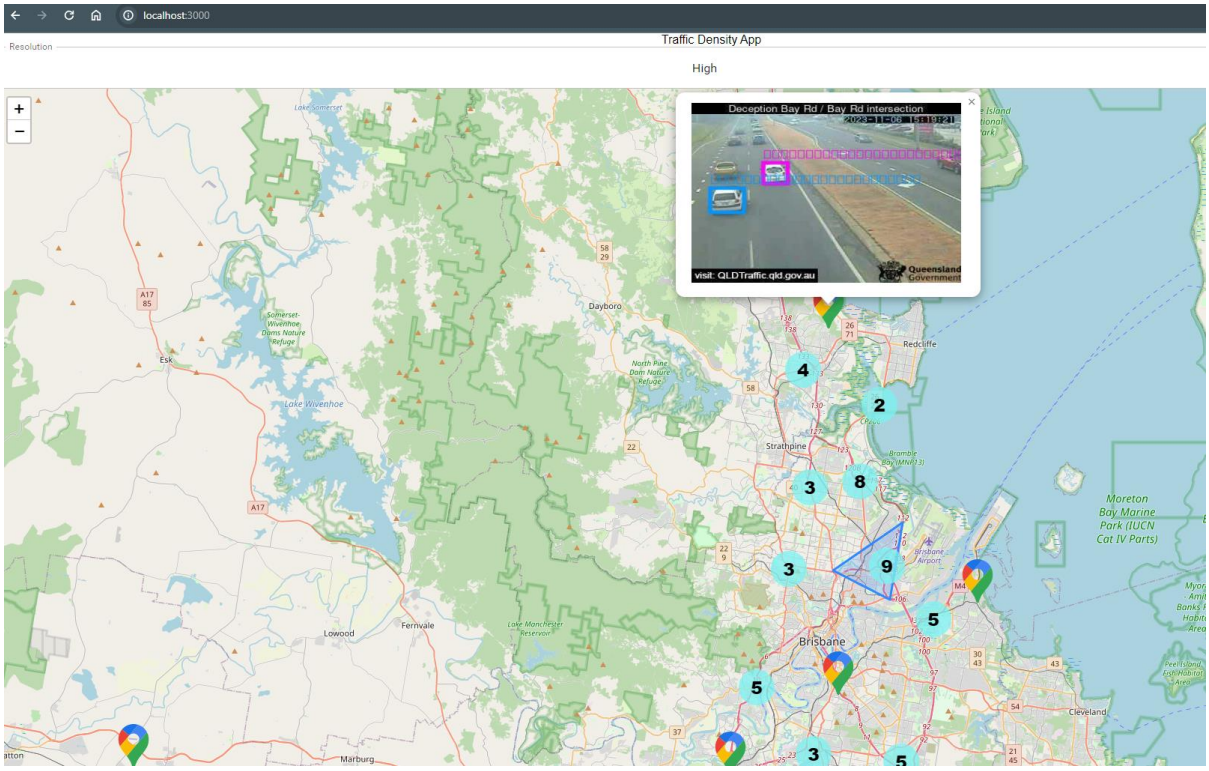
Extensions

This work could be extended by finding more cameras or video streams to incorporate into the map, particularly public/open-source camera streams. The QLD Traffic API provides URLs of still images by default so extending this work to handle video seems like a natural continuation.

User guide

To use this application simply zoom in and out and use the left click on the mouse to pan around.

If you use the “Resolution” drop down at the top several pins will begin to fill the map from all over Queensland.
If you select “Low” then 5 random pins will be placed, each with a popout showing an image from the camera at that location. The image may or may not have bounding boxes super imposed on the image if it has been processed.
If you select “Medium” then 30 random pins will be placed.
If you select “High” then all the pins available will be placed, this value ranges from 150-160.

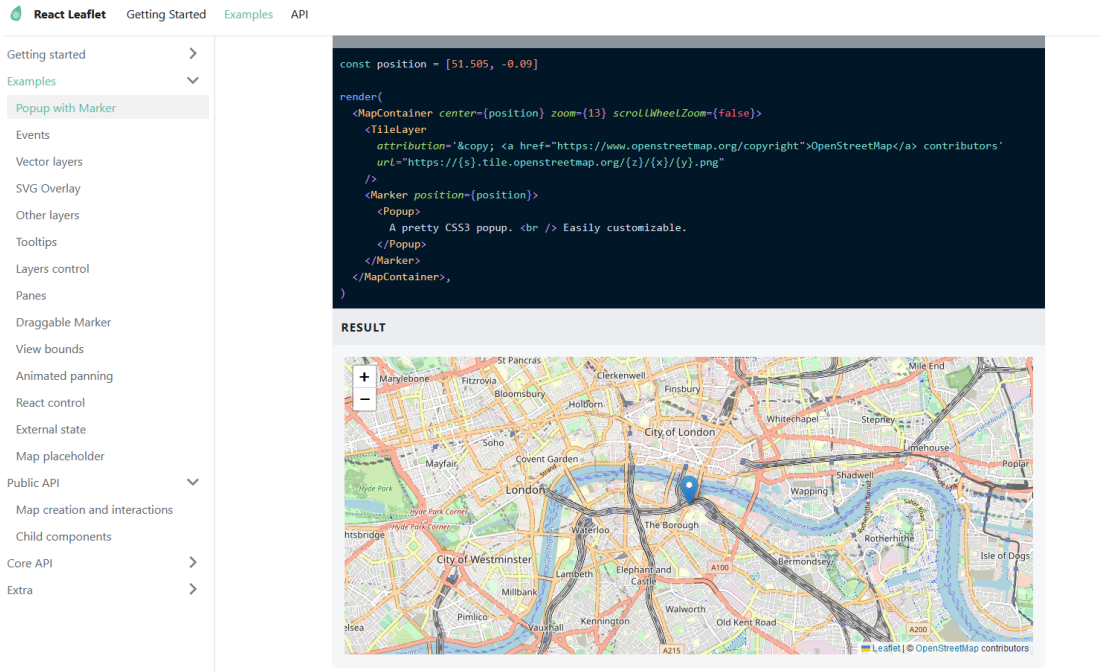


If you watch the console you will see some feedback on what is happening including any issues with the value of “resolution” and “setInterval()”.

Eventually you the app will become unstable and will need to be refreshed it won’t crash on the server but it will in the browser.

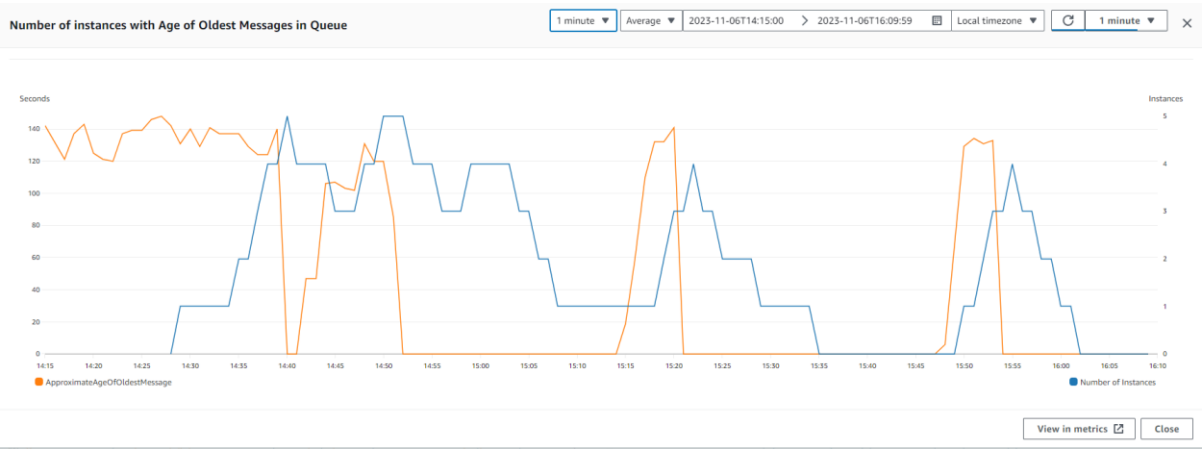
Appendices

a) React-leaflet documentation demonstrating how to put markers on an interactive map of London.



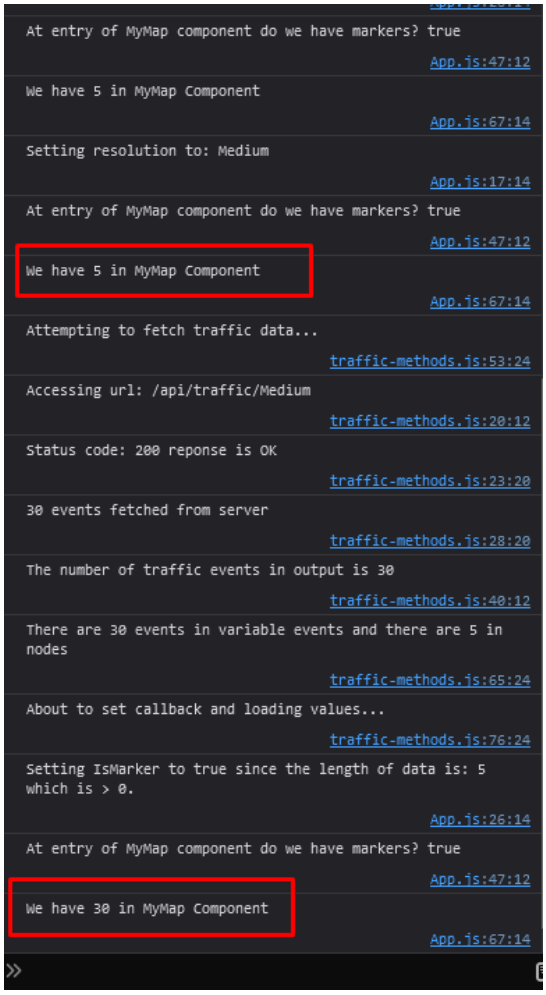
b) The following is a graph of the number of worker instances in my auto-scaling group against the number of visible messages in the SQS queue. If the number of messages is growing then the number of instances should scale up to manage the load before scaling down.

c) The following is a graph of the number of worker instances in my auto-scaling group against the age of the oldest message in the SQS queue. This shows that if tasks are not getting completed then we are bottlenecking and need to add more instances.

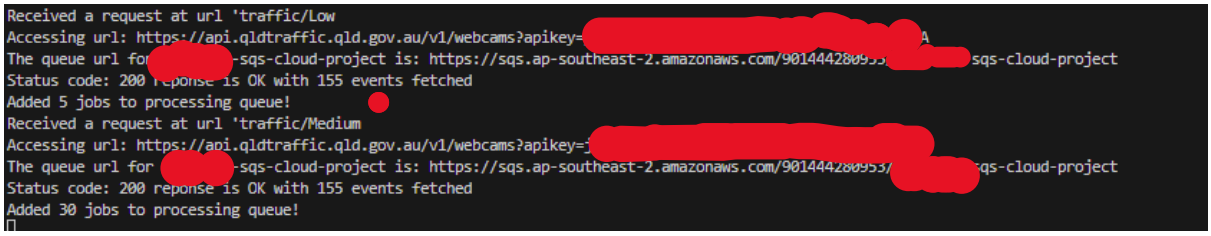


d) A clear and pretty bounding box:

e) Successfully control the number of requested traffic nodes from react js front end and receive the appropriate amount.



f) An increasing number of messages sent to the queue:



g) Jobs being retrieved from queue on back-end workers:

```
6 messages were retrieved
Redcliffe_Houghton_Hwy_Sth.jpg was deleted from s3 bucket

Beginning to download image from https://d2w9pjl8ozl6el.cloudfront.net/Metropolitan/Redcliffe_Houghton_Hwy_Sth.jpg...
alex_pde_south.jpg was deleted from s3 bucket

Beginning to download image from https://d2w9pjl8ozl6el.cloudfront.net/Sunshine_Coast/alex_pde_south.jpg...
takalvan-walker-ntheast.jpg was deleted from s3 bucket

Beginning to download image from https://d2w9pjl8ozl6el.cloudfront.net/Wide_Bay/takalvan-walker-ntheast.jpg...
The download was successful
The upload of Redcliffe_Houghton_Hwy_Sth.jpg should be complete...
The download was successful
The upload of alex_pde_south.jpg should be complete...
The download was successful
The upload of takalvan-walker-ntheast.jpg should be complete...
bargara-gahans-west.jpg was deleted from s3 bucket

Beginning to download image from https://d2w9pjl8ozl6el.cloudfront.net/Wide_Bay/bargara-gahans-west.jpg...
MRMETRO-1464.jpg was deleted from s3 bucket

Beginning to download image from https://d2w9pjl8ozl6el.cloudfront.net/Metropolitan/MRMETRO-1464.jpg...
MRSCHD-299.jpg was deleted from s3 bucket

Beginning to download image from https://d2w9pjl8ozl6el.cloudfront.net/Gold_Coast/MRSCHD-299.jpg...
Commencing Prediction...
Returning the predictions:
[]
Generating canvas and bounding boxes from prediction...
Attempting to write update image with type object to file...
The upload of alex_pde_south.jpg should be complete...
The download was successful
The upload of bargara-gahans-west.jpg should be complete...
Commencing Prediction...
Returning the predictions:
[]
Generating canvas and bounding boxes from prediction...
Attempting to write update image with type object to file...
The upload of Redcliffe_Houghton_Hwy_Sth.jpg should be complete...
The download was successful
The upload of MRMETRO-1464.jpg should be complete...
The download was successful
The upload of MRSCHD-299.jpg should be complete...
Commencing Prediction...
Returning the predictions:
[]
Generating canvas and bounding boxes from prediction...
Attempting to write update image with type object to file...
The upload of takalvan-walker-ntheast.jpg should be complete...
Commencing Prediction...
Returning the predictions:
[
  {
    bbox: [
      86.70934677124023,
      40.37306213378906,
      16.131114959716797,
      39.25372314453125
    ],
    class: 'person',
    score: 0.5811037421226501
  }
]
Generating canvas and bounding boxes from prediction...
Attempting to write update image with type object to file...
The upload of bargara-gahans-west.jpg should be complete...
```