Image-Based Vehicle Traffic Monitoring Final Report

Supervisor - Francisco Benita
William Bowden 1009503

Friday 13th November 2024

Contents

1 Abstract

The goal of the work this report concerns was to create an application that allows users to initiate data collection jobs for traffic information and camera images from particular stretches of road, or ”speedbands”, across Singapore’s road network.

Using React.js [1], Amazon Web Services [2] and the Land Transport Authority’s DataMall API [3], I created an app that fulfils the requirement and collects traffic data in real time, running in the cloud, that users can download as CSV (Comma Separated Value) files.

This application acts as a proof of concept for a potential future application with a larger suite of tools, all serving the purpose of analysing traffic conditions in a region of choice, even beyond Singapore.

2 Introduction

The goal of the Image-based Vehicle Traffic Monitoring project is to use neural networks and public API data, collected from the Land Transport Authority (LTA) [3], to analyse the traffic conditions of roads in Singapore. Using this data, we can calculate an estimate for the emissions on each stretch of road based on number of vehicles and vehicle types. This data could then be used for assisting city planning by locating traffic hot spots, creating public transport incentive schemes to minimise traffic at peak times, and many more applications. In order to achieve this goal, we needed to collect data on the traffic conditions, as well as traffic camera images, for particular stretches of road (”speedbands”) for analysis. The nature of the LTA API means that we cannot obtain historical speedband data, and so to build up a suitable dataset, we have to collect data in real time and store it for later use. My task for this project was to create an application for collecting such data. Previous work by my peers included data collection from the LTA’s API, which I then built upon with a web application and cloud infrastructure to run collection jobs in real time.

Developing this project posed an interesting challenge. Architecting the backend to execute collection jobs in the cloud and store results was difficult due to the sheer number of moving parts in AWS and configuration options available, but once complete it proved a very rewarding achievement.

3 Methodology

3.1 Frontend Design

For my frontend architecture, I opted to work with Typescript and React [1] using Next.js [4] as the React framework of choice. The main reason for this decision was my prior knowledge of the technologies. As many of my previous projects have been developed in Typescript and React, I felt comfortable using them here. Beyond my experience, I also chose Typescript and React for their simplicity and the rapid prototyping that comes as a result of using them.

In line with the desire to rapidly prototype, I chose to use the Antd design library [5] for my

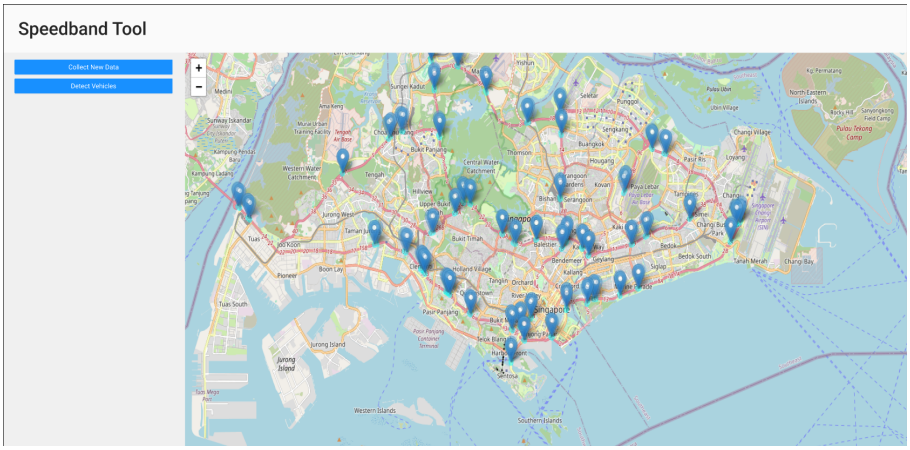

Figure 1: UI Prototype - Dashboard Page

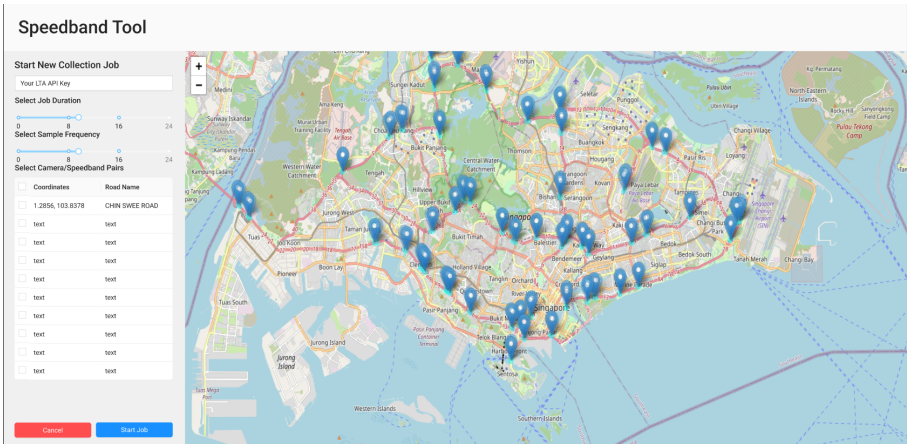

Figure 2: UI Prototype - Starting a collection job

UI components, as this spared me the effort of styling and building the UI and instead allowed me to focus on the functionality of the app.

With these decisions settled, I started by creating a UI design in Figma [6] which I reviewed with my supervisor, made some small tweaks to, and settled on a suitable version of before proceeding with development.

3.2 Backend Design

For the backend architecture, which would be responsible for running user's data collection jobs and storing the results, I decided to use Amazon Web Services [2]. Again, this was largely due to my previous experience with creating cloud infrastructure on AWS. However, I also chose AWS due to their AWS Cloudformation service [7] which allows users to define and deploy cloud infrastructure using code. This means that I could create a working cloud system as a proof of concept, and then interested parties could simply fork my repository and deploy the same infrastructure on their own AWS accounts in far fewer steps than it would take to create from scratch. Furthermore, AWS offers easy user authentication through their Cognito service [8] and simple management of user sessions and authenticated API requests using AWS Amplify [9].

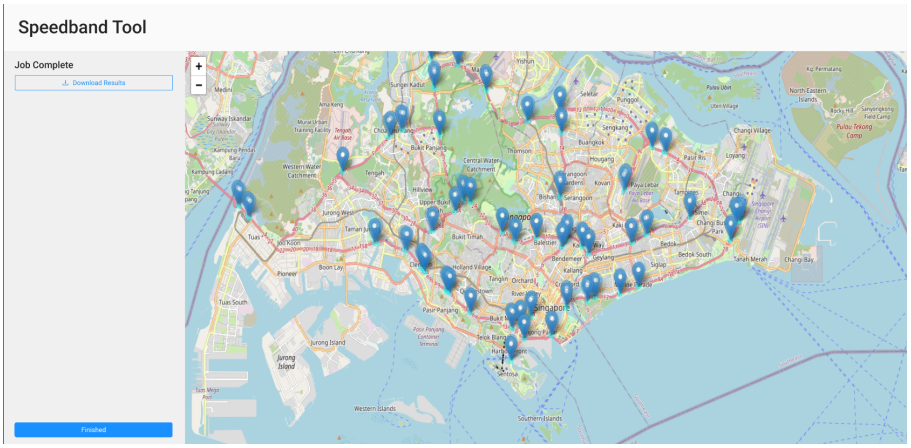

Figure 3: UI Prototype - Collection job complete

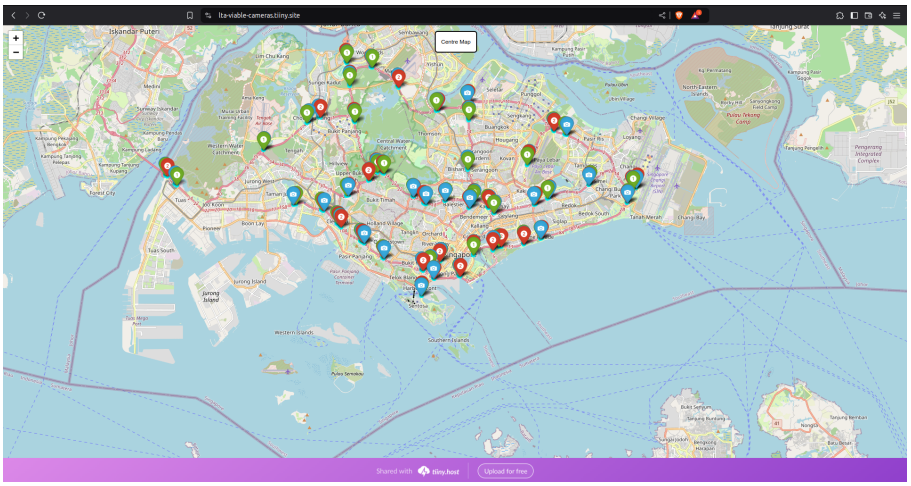

Figure 4: First Iteration - Speedband visualiser tool

3.3 Development & Iterations

3.3.1 First Iteration

Before creating the larger application and expanding the scope of this project, my initial task was simply to deduce which speedbands would be appropriate for data collection and to create a basic tool to visualise them. Building on work from my peers, I manually combed through some data and camera images from a spreadsheet to decide whether they were viable. Once I had a pruned list, I converted the spreadsheet to a CSV file, and then used a simple Python script to convert that into a series of JSON objects for later use.

Next, I built a simple web app (Fig. 4) using HTML/CSS/JS, alongside the Leaflet.js library [10] to visualise these viable cameras and speedbands on a map of Singapore. After presenting and discussing this prototype with my supervisor, we decided to expand the app's functionality to facilitate data collection jobs. In order to do this, I decided it would be most appropriate to switch from pure HTML/CSS to a web framework to accelerate development.

3.3.2 Second Iteration

The process of developing the second and final iteration began with the creation of a React app that had all the necessary UI but no interaction with any backend. Using my UI prototype as an anchor, I assembled the relevant UI components from the Antd library to make the pages necessary for the app's core functionality. The only major design decision that was made in this step was a change of colour theme. I decided to switch to a dark colour theme, which I felt was easier on the eyes. Once a React prototype was established, it was appropriate time to start work on the backend.

The next logical step was to set up user authentication. Using AWS Amplify [9], I created a Cognito [8] user pool with email authentication. Doing this on my local machine, using Amplify's Command Line Interface [11], created the files necessary for integrating user authentication in my React project right where I needed them. Then, using the Amplify React UI library [12], I added a sign up/sign in form to my app and required that users be authenticated to access it.

Once users could sign in, I began to design the cloud infrastructure which would allow the creation and fulfilment of data collection jobs. Using AWS Cloudformation [7], I defined the following AWS architecture in a YAML file:

Service	Description	Function
DynamoDB [13]	NoSQL database	Storing parameters and statuses for jobs.
S3 [14]	Data storage	Storing job results as .csv files
Lambda [15]	Serverless functions	Handle API requests and perform data collection.
API Gateway [16]	API service	Exposing API endpoints and validating requests

Once this was complete, I wrote code for the following Lambda functions in Python:

Function Name	Purpose
JobScheduler	Initiating a collection job on request of the user.
DataCollection	Performing one ”step” of a data collection job.
CheckJob	Checking the status of a collection job.
GetUserJobs	Return all of a user's jobs for display on the frontend.

After the cloud infrastructure and function code was in place, the last remaining step was to integrate it with the frontend and test. I built functionality into the pages of the web app, using AWS Amplify to send authenticated API requests to the server. I used Next.js's router to tie together the pages so that the flow of a data collection job was complete. This left me with a working version of the application, which is discussed in more detail in the sections to follow.

3.4 Feature overview

Below I will outline the features of my application and the data/control flow involved in them. Interested parties can look further into the source code and setup instructions via Appendix A.

3.4.1 Starting a new collection job

1. The user must log in to access the application.
2. From the dashboard page, the user is presented with a table of their existing/past jobs, if any exist.
3. Below the table, the user may click a button to start a collection job.
4. On the resulting page, the user must enter their LTA API key, the desired duration of the job, the desired collection frequency, and the desired camera/speedband pairs they wish to collect data for. They may then click a button to submit the job. An API request is sent.
5. In the cloud, the API request is handled by the ‘JobScheduler‘ Lambda function. The input is validated, and then a new entry in the DynamoDB table is created for the job.
6. Next, an AWS EventBridge [17] rule is set up, which will invoke the ‘DataCollection‘ Lambda function at the user-defined interval. A HTTP 200 response is returned to the user.
7. On the frontend, the user is then directed to the ”In Progress” page.

3.4.2 Performing data collection

1. The ‘DataCollection‘ Lambda function is invoked by the EventBridge rule, at the user-defined interval.
2. When invoked, it collects the necessary information from the LTA API:
   (a) The speedband information is retrieved from the LTA API. Since the API returns all of the live speedband info in chunks of 500 rows, I use a binary search-style algorithm to locate the desired speedbands in the data, since the speedbands’ ”LinkID”s are ordinal. This means a desired speedband can be located in only a handful of calls rather than combing through the approximately 58,000 rows of data available at time of writing.
   (b) The corresponding camera images are retrieved for the desired speedbands, in an almost identical manner to above.
   (c) An attempt is made to retrieve a .csv file for the job from AWS S3. If none is found, a new one is created, and the retrieved data is saved to it. If one exists, the retrieved data is appended to the end of the file.
   (d) If the job’s end time is before the current time, i.e the job has finished, the Lambda function cancels the EventBridge rule, sets the job’s status in Dynamo to ”Complete”, and removes the user’s API key from the Dynamo table entry.
   (e) If an error occurs in the process, such as the user’s API key being missing or bad, the Lambda function cancels the EventBridge rule, sets the job’s status to ”Failed” in Dynamo, and adds a ”reason” field to the Dynamo table entry to explain the failure.
3. As the function is internal to the server architecture, nothing is returned to the user.

3.4.3 Getting a user's jobs

1. When the user logs into the app and is directed to the dashboard, the app requests a list of their jobs. An API request is sent.
2. The 'GetUserJobs' Lambda function handles the API request. It retrieves the user's ID from their API request headers, meaning the user can only request their own jobs, and searches the Dynamo table to collect the user's jobs.
3. When searching the table, the "apiKey" field is omitted from the rows returned, so that in case a malefactor manages to circumvent the API authentication check, they still shouldn't be able to obtain another user's API key.
4. The user's jobs are returned to the frontend as a list.
5. The user's jobs are displayed in a table on their dashboard, with links to check their status or results.

3.4.4 Checking a job

1. When the user clicks on a "Pending" job from their dashboard, or after the user creates a new job, they will be directed to the "In Progress" page.
2. On this page, the app submits a job check API request.
3. The 'CheckJob' Lambda function handles the API request. It retrieves the job row from the Dynamo table, and returns it to the frontend. If the job's status is marked as "Complete", the function generates a pre-signed URL to access the results .csv file from S3, and appends it to the returned row. The user's API Key is omitted from the result for security.
4. On the frontend, the user is shown different content based on the status of the job:
   (a) If the job is still in progress, the user is shown a progress bar which updates every second, showing the time remaining until the job's completion. A "CheckJob" request is made every 30 seconds on this page, to ensure the job has not failed before completion.
   (b) If the job is complete, the user is redirected to the "Complete" page.

3.4.5 Job complete

1. If the user clicks a "Complete" job from their dashboard, or are directed from the "In Progress" page, they will reach the "Complete" page.
2. Users may download the job results using the pre-signed S3 URL generated by the 'CheckJob' Lambda function.

3.5 System Diagram

The diagram in Appendix B illustrates the path a user may take through my app and the interactions it makes with AWS along the way.

4 Results

The application, at time of writing, is able to perform its intended functions and execute data collection jobs. To demonstrate this, I ran a 15 minute long data collection job on my own account.

4.1 Demonstration

First, I sign into my account.

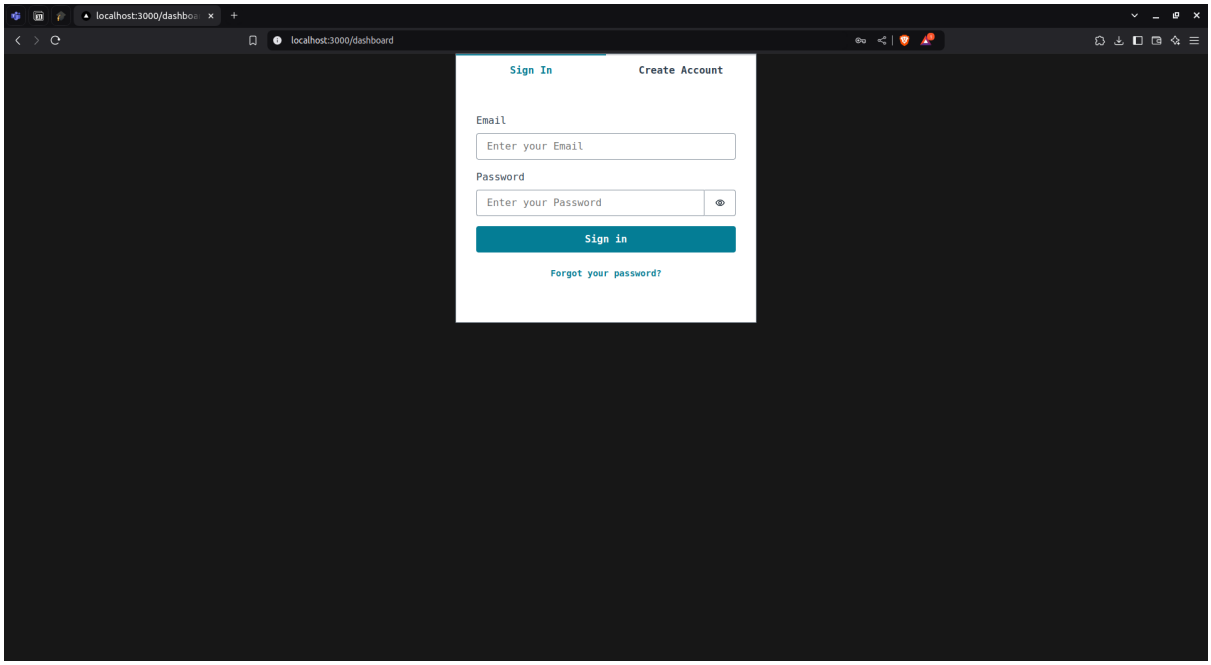

Figure 5: Results - Signing in

I am then presented with the dashboard page.

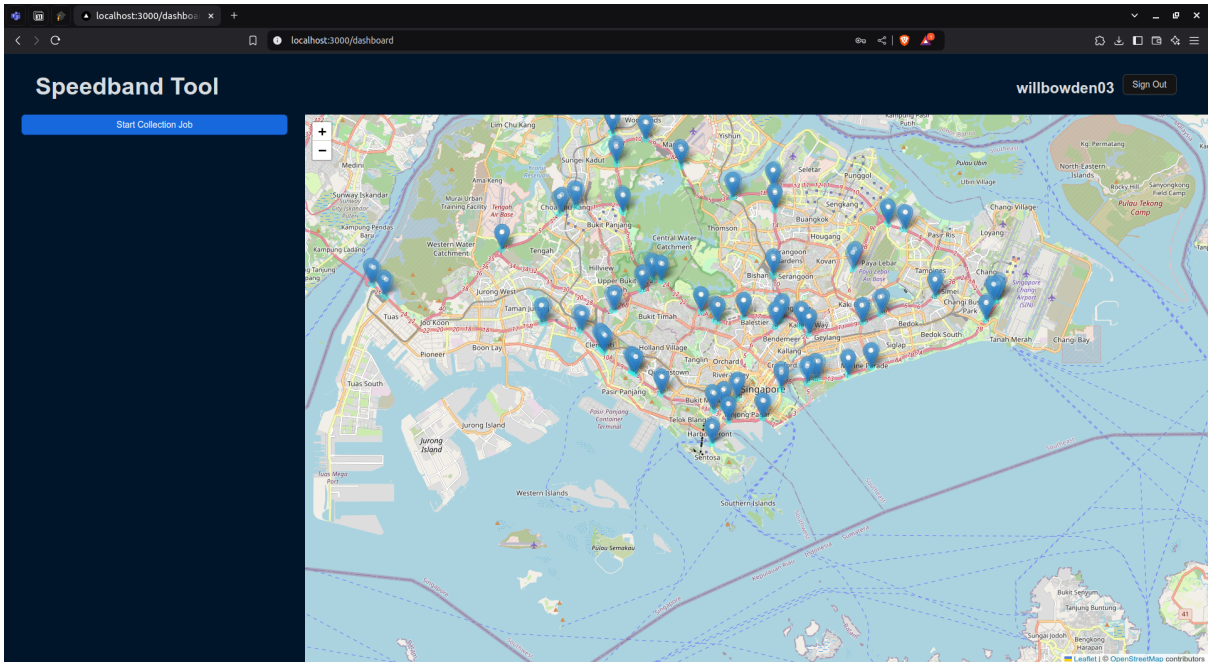

Figure 6: Results - Dashboard page

If I click ”Start Collection Job”, I’m directed to the ”Start job” page.

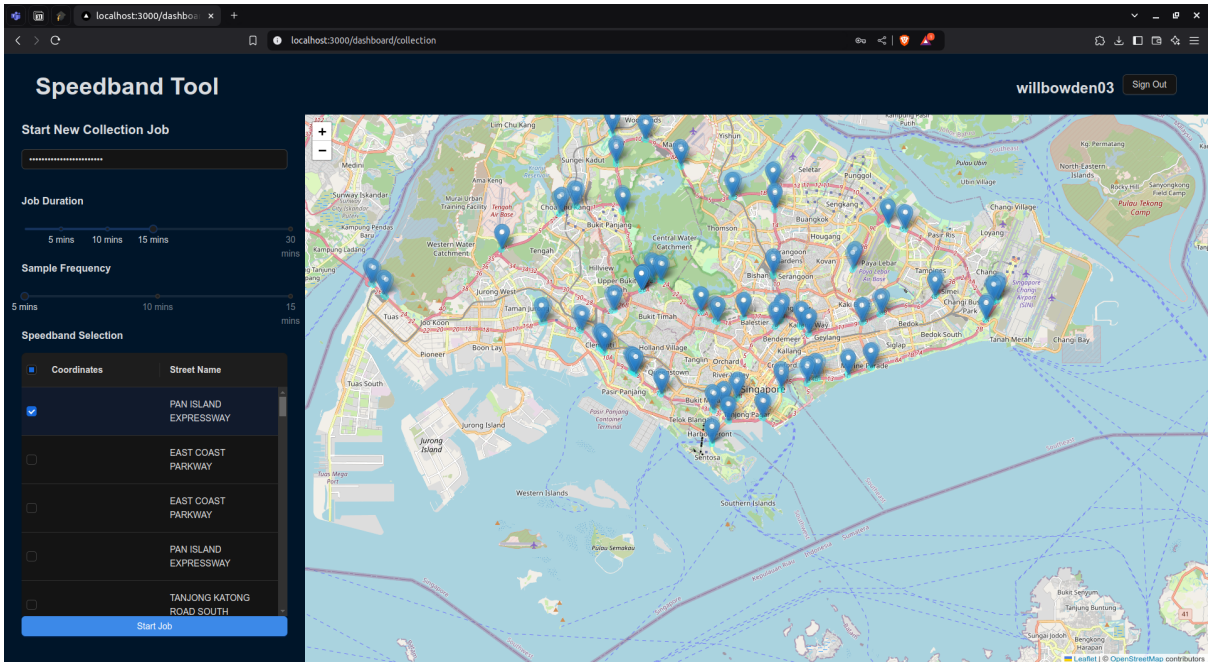

Figure 7: Results - Start job

I enter the parameters for the collection job, select a speedband either by checking the box in the table or clicking on the markers on the map, and provide my API key. Then, clicking ”Start Job” directs me to the ”In progress” page.

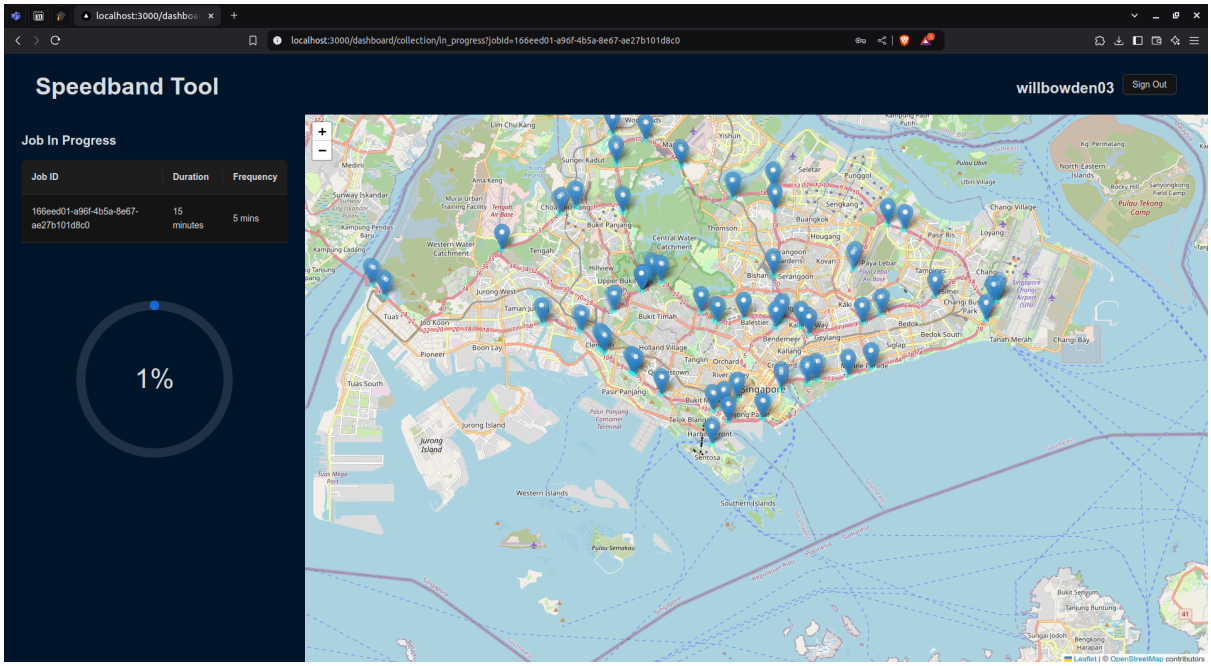

Figure 8: Results - Job in progress

If I navigate back to the dashboard by clicking the app's title on the top bar, I'm able to see my ongoing job on the dashboard screen.

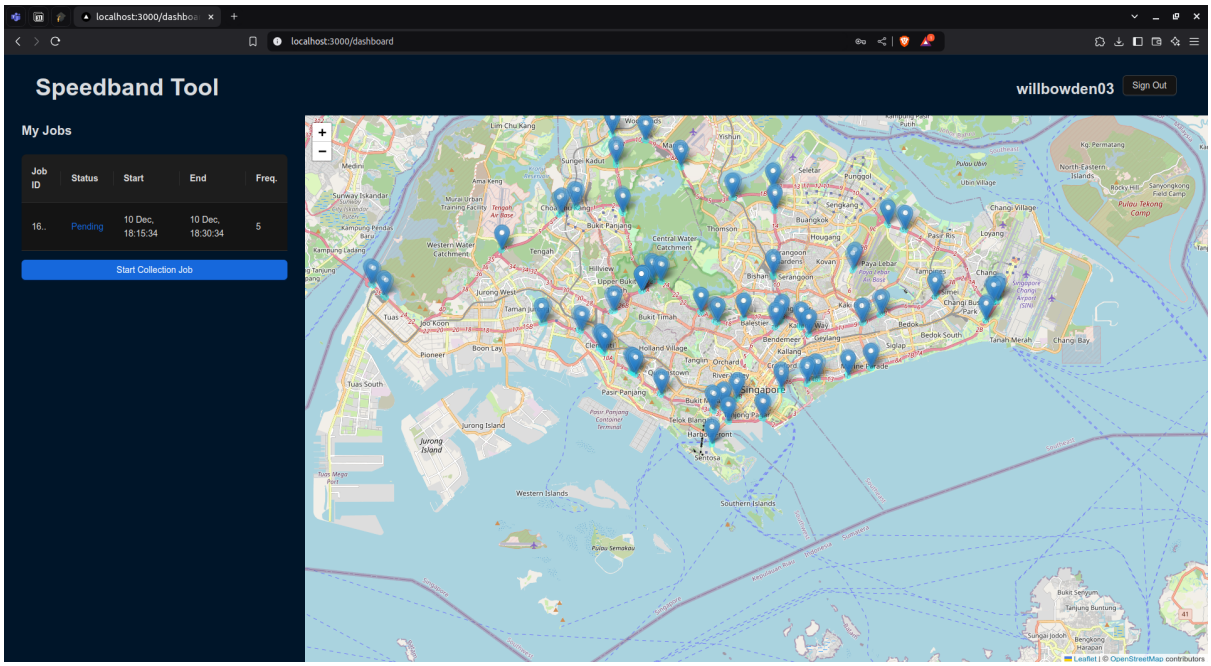

Figure 9: Results - Job on dashboard

Clicking "Pending" on the job's row in the table takes me back to the "In progress" page for the job, which is nearly done.

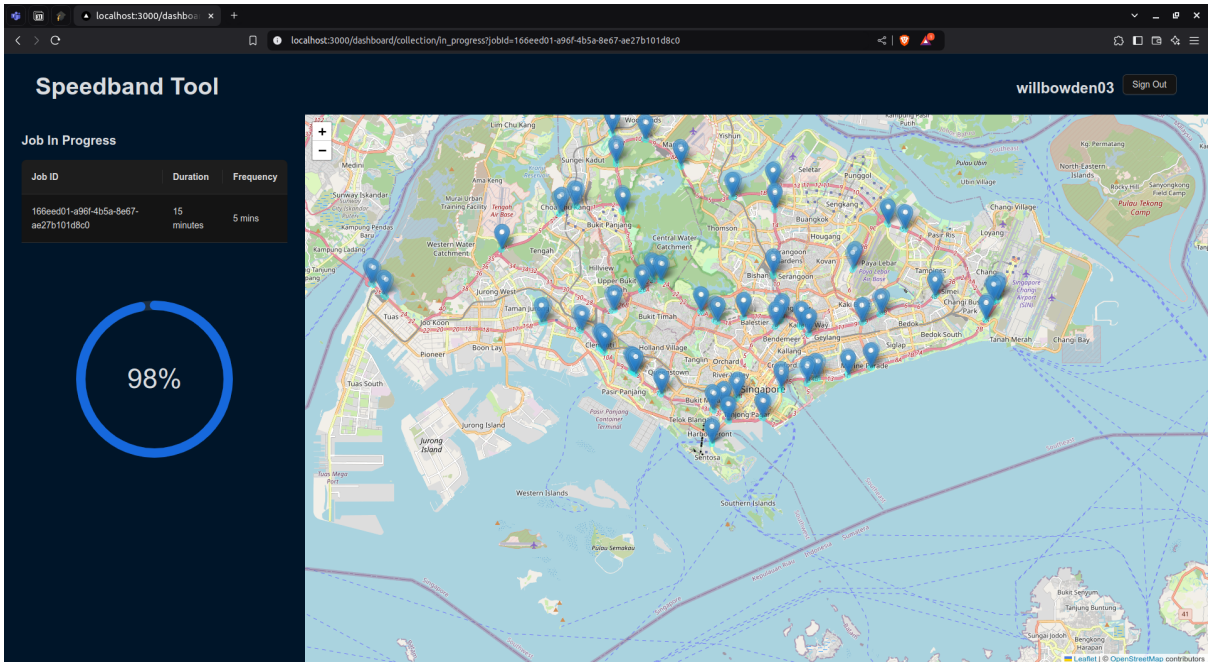

Figure 10: Results - Job nearing completion

Once the job reaches completion, I'm automatically redirected to the "Complete" page.

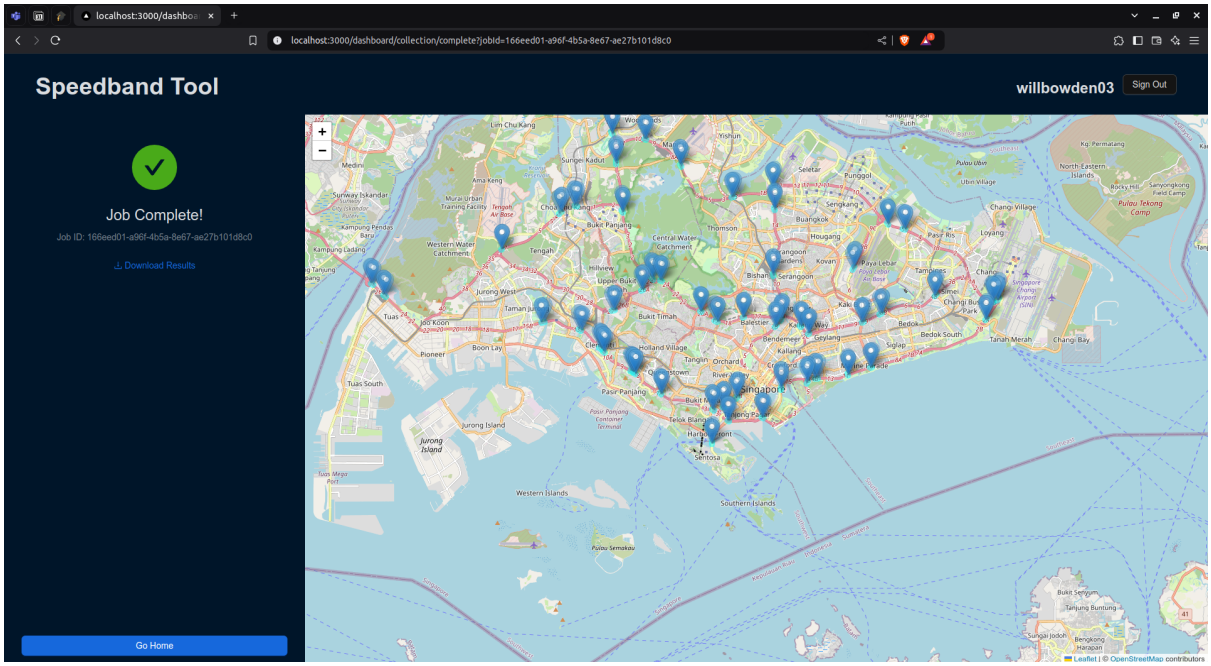

Figure 11: Results - Job complete

Clicking on "Download Results" opens my browser's download window.

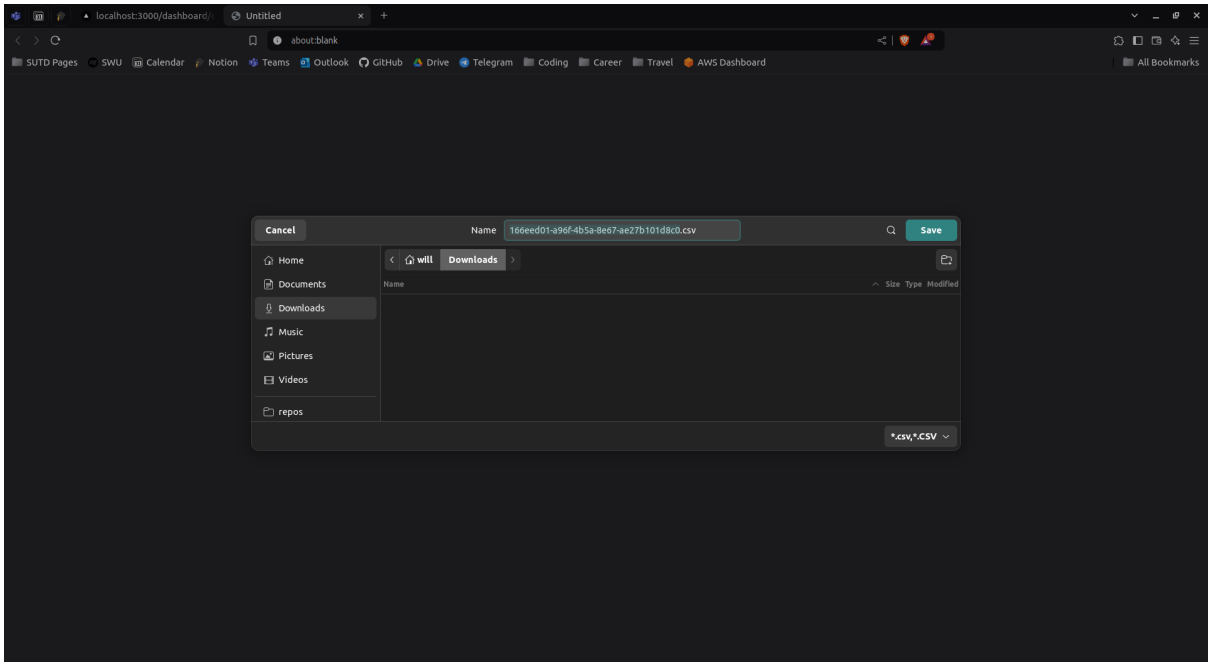

Figure 12: Results - Downloading job results

If I open the .csv file that I've downloaded, I can see I have three rows of data in it. For a 15 minute collection job at 5 minute intervals, this is correct. I can see that each row contains speedband information about my chosen speedband alongside image URLs for the chosen traffic camera.

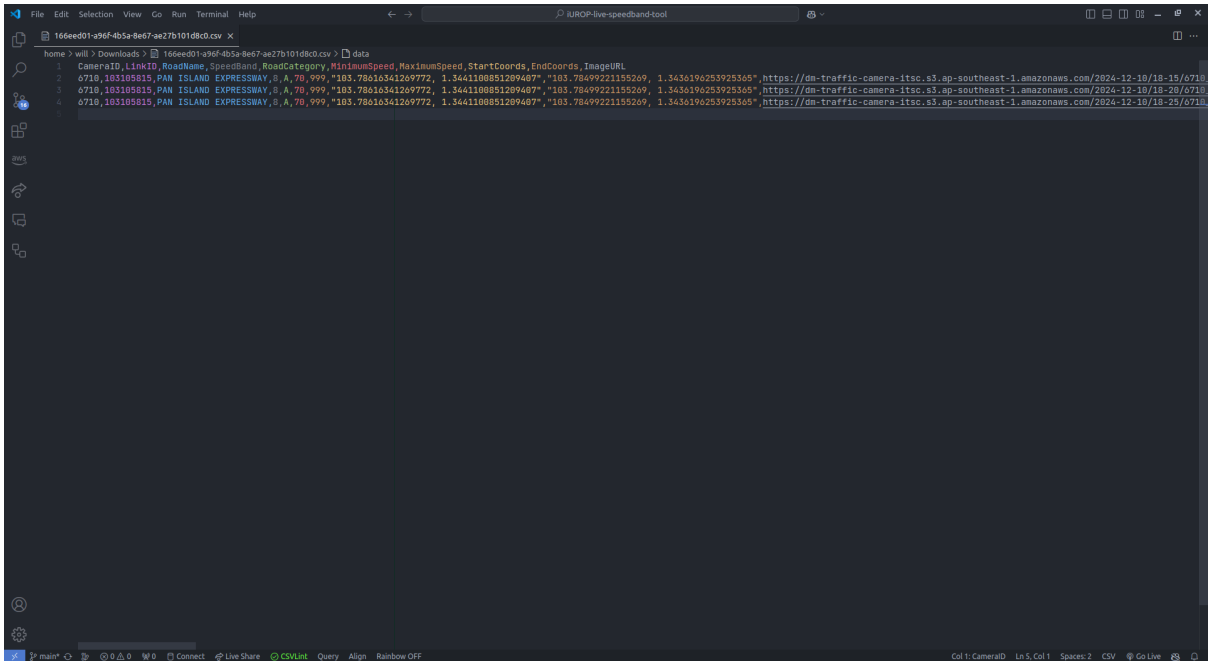

Figure 13: Results - Downloaded data

4.2 Discussion of results

The above demonstration shows that my application fulfils its goal of executing data collection jobs. It can authenticate users, connect to the backend, collect and organise data from the LTA API, and store results for users to access. This version acts as a proof of concept for potential future implementations. I took a number of steps to make it possible for future collaborators to build on top of my work.

Firstly, by choosing to develop the frontend application in a modular web framework such as React, I made it relatively simple to expand upon the app. Others can easily add new pages to the site or add new components to existing pages to expand its capabilities. One example could be to take the vehicle detection process, performed by machine learning models, and integrate it into this app. Users could start data collection jobs, and then upload their datasets to be analysed by a neural network to detect vehicles in the traffic images. I know my peers in this project have been working on such a technology, so I believe it would be simple and convenient to combine the two applications.

Additionally, building the cloud infrastructure using a YAML file to work with AWS CloudFormation [7] makes the entire backend setup highly reproducible. Others would need to run a few commands and make a few manual configurations to have an identical system to my own set up on their own AWS account. To support this, I wrote step-by-step instructions for others to recreate my cloud infrastructure setup in their own environments. Interested parties can find these instructions in the GitHub repository in Appendix A.

5 Analysis

5.1 Scalability

If this application was deployed on a large scale, it would need to be able to handle multiple concurrent collection jobs for multiple users. Fortunately, the design of my system and the nature of AWS allows for the application to be scaled quite easily.

AWS Lambda [15] is, by design, serverless, meaning multiple instances of the same function can execute concurrently. Administrators can configure a limit on concurrent Lambda executions, allowing them to find a balance between performance and cost efficiency. This would allow users of a production-level system to execute concurrent collection jobs using their own API keys.

AWS Dynamo [13], the NoSQL database used to store job information, is also configurable to support a larger number of query operations, which would enable this application to scale. However, this would likely only be necessary if the application grew to have a very large user base.

5.2 Modularity

When developing this application, I endeavoured to make it modular so that the separate parts of the code base could easily be repurposed, reconfigured or reused by others. This goal works

well with the collaborative and open source nature of this project.

All of the backend server functionality is contained in separate Lambda functions, which are simply Python scripts. This means that each section of the system can be easily reconfigured, replaced, or improved.

The ‘DataCollection‘ Lambda function is the central part of this application, as it is responsible for retrieving the data from the LTA’s API. As the other functions it is designed to be modular, and it would be relatively simple to extract the functionality and allow other developers to run collection jobs on their architecture of choice. The algorithms I implemented in that function could conceivably be built upon by others to expand the application’s capabilities.

5.3 Potential Improvements

Despite producing a functional system in the limited time available, there are some places the project could be improved.

Currently, users’ LTA API keys are stored as fields in a Dynamo table, and retrieved by the relevant Lambda functions for use in collection jobs. The main reason for this choice was to meet development deadlines, however it is not the most secure. The current system does not offer a means for external users to retrieve any API keys from the database, which it does by authenticating API requests and filtering fields in the ‘GetUserJobs‘ Lambda, and removing the API key once a collection job is finished. Despite these measures, the key is still stored in a database in plain text. This could be improved upon by using a technology such as AWS Key Management Service [18], to securely store encrypted versions of users’ keys.

Next, the infrastructure setup in this project has the AWS services defined with general-purpose permissions. For example, the Lambda functions that need access to certain databases have complete access to those services, meaning, in a large scale application, they could access database tables outside their scope, or potentially modify databases in ways they shouldn’t be able to. This, again, was done to accelerate development of the project. In a future version, it would be more appropriate to craft resource-specific permission policies so that each section of the cloud infrastructure may access only what it needs and nothing more. This would create a more secure and future proof version of this project at the expense of longer development time.

Finally, there is only limited error handling in the system. So far, if a user inputs a bad API key, the job is marked as ”Failed”. If the user then tries to check that job from the frontend, they will be presented with an error modal and forced to return to the dashboard. There is also input validation on the job creation page to ensure users provide the minimum necessary data for job execution. Beyond this, however, there is room for more error checking and handling. For example, the ‘DataCollection‘ Lambda function would benefit from logic to handle network errors and retry API calls if such errors occur. This would make the system more reliable and resilient to external issues.

5.4 Significance

This application exists mainly as a proof of concept for cloud-based data collection jobs for traffic analysis in Singapore. A future final version of this application could look different in some ways. The main way I foresee this application being transformed and deployed is as follows.

Rather than allowing individual users to submit collection jobs with fixed duration, a potential application could run one continuous collection job to amass an ever-growing historical dataset of traffic information and images. This application could then allow users to download data from the dataset in date-defined segments, or download it all at once. Since the project is of an open source nature, the deployed site could possibly allow users to donate to contribute to cloud costs. This would mean any interested parties could use the publicly available historical traffic data for their own uses without having to build up their own dataset themselves. The necessary infrastructure to change this application in this way would quite easily tack on to the existing system.

6 Conclusion

This project successfully delivered an application for collecting and storing real-time traffic data from the Land Transport Authority's API. By using React, AWS, and YAML-based infrastructure, the system is functional, scalable, and easy to deploy.

The modular design ensures future developers can easily add features like vehicle detection or advanced data analysis. The serverless architecture, built with AWS Lambda and DynamoDB, supports scalability and cost efficiency, making the system suitable for larger-scale use.

There are areas for improvement. Enhancing data security, refining permission policies, and improving error handling would make the system more secure and reliable. These changes would prepare the application for production use.

This project demonstrates the potential of real-time traffic data collection for urban planning and traffic management. It provides a strong proof of concept and a foundation for further development.

7 References

[1] Meta, "React.js." [Online]. Available: https://react.dev/

[2] Amazon Web Services, Inc., "Amazon Web Services." [Online]. Available: https://aws.amazon.com/

[3] Land Transport Authority, "LTA DataMall." [Online]. Available: https://datamall.lta.gov.sg/content/datamall/en.html

[4] Vercel, "Next.js." [Online]. Available: https://nextjs.org/

[5] Ant Group, “Ant Design Framework.” [Online]. Available: https://ant.design/

[6] Figma, “Figma.” [Online]. Available: https://www.figma.com/

[7] Amazon Web Services, Inc., “AWS Cloudformation.” [Online]. Available: https://aws.amazon.com/cloudformation/

[8] ——, “AWS Cognito.” [Online]. Available: https://aws.amazon.com/cognito/

[9] ——, “AWS Amplify.” [Online]. Available: https://aws.amazon.com/amplify/

[10] V. Agafonkin, “Leaflet.js.” [Online]. Available: https://leafletjs.com/

[11] Amazon Web Services, Inc., “”amplify cli”.” [Online]. Available: https://docs.amplify.aws/gen1/javascript/tools/cli/

[12] Open Source, “amplify-ui-react.” [Online]. Available: https://www.npmjs.com/package/@aws-amplify/ui-react

[13] Amazon Web Services, Inc., “AWS DynamoDB.” [Online]. Available: https://aws.amazon.com/dynamodb/

[14] ——, “AWS S3.” [Online]. Available: https://aws.amazon.com/s3/

[15] ——, “AWS Lambda.” [Online]. Available: https://aws.amazon.com/lambda/

[16] ——, “AWS Lambda.” [Online]. Available: https://aws.amazon.com/api-gateway/

[17] Amazon Web Services, Inc, “AWS EventBridge.” [Online]. Available: https://aws.amazon.com/eventbridge/

[18] ——, “AWS Key Management System.” [Online]. Available: https://aws.amazon.com/kms/

8 Appendices

A Source Code & Instructions

Instructions to set up the cloud infrastructure on your own account, and the code to deploy your own frontend app can be found in the public repository on GitHub via the following link: `https://github.com/willbowden/iUROP-live-speedband-tool`.

B System Flow Diagram

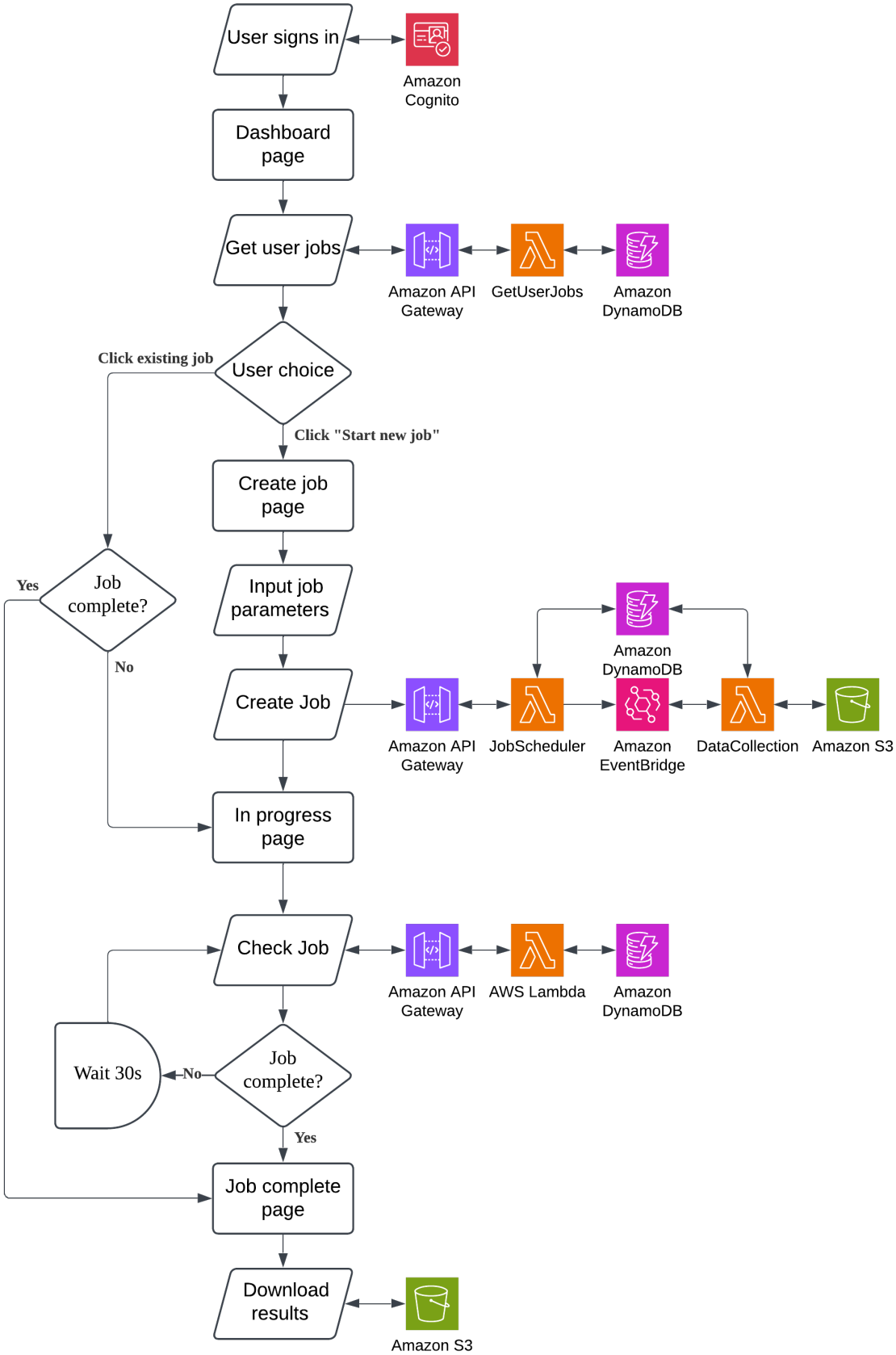

Figure 14: System flow diagram