

# Technical Report (Frontend)

## Contents

Project Overview.....	1
Problem Description .....	1
Application Overview.....	1
Feature 1: Map Page.....	2
Feature 2: Grid Page .....	3
Home Page.....	3
Repository Breakdown.....	4
Application Architecture.....	4
Repository Structure Breakdown.....	4
Frontend detailed breakdown .....	4
Dash-Application.....	5
Result-Generator .....	8
Docker-compose.yml .....	8
Design Conceptualization .....	8
Pre-Development.....	8
First phase user experience .....	9
Second phase user experience .....	10
Future Development Directions .....	11
Architecture Changes.....	11
Frontend .....	12
Customizable Grid.....	12

## Project Overview

### Problem Description

Land Transport Authority (LTA) has set up 87 cameras at key locations in Singapore, mainly across the expressways to monitor the traffic. Image data from these cameras are refreshed in LTA DataMall every 5 minutes. LTA would like to estimate the density of traffic from these images, where density is defined as the vehicles per kilometer per lane at a given instant. Together with other types of traffic data on the DataMall site, LTA would like to create a model that predict traffic jams using those variables and display the density, traffic speed and prediction on an interface.

### Application Overview

With LTA's engineers' objectives in mind, we developed the application to not only present the result, but also to improve their ease of access to the images and assist them in their analysis. Using a pre-trained YOLOX model (weight obtained from MMDet's Model Zoo) in combination with SAHI (Slicing Aided Hyper Inference), we count the number of vehicles that can be seen from the processed images of the traffic cameras (refer to figure 1 below for an example of the model's detection capabilities). The counts are used as input to the density formula to calculate the required traffic density, which is then further used to calculate the probability of traffic jam.

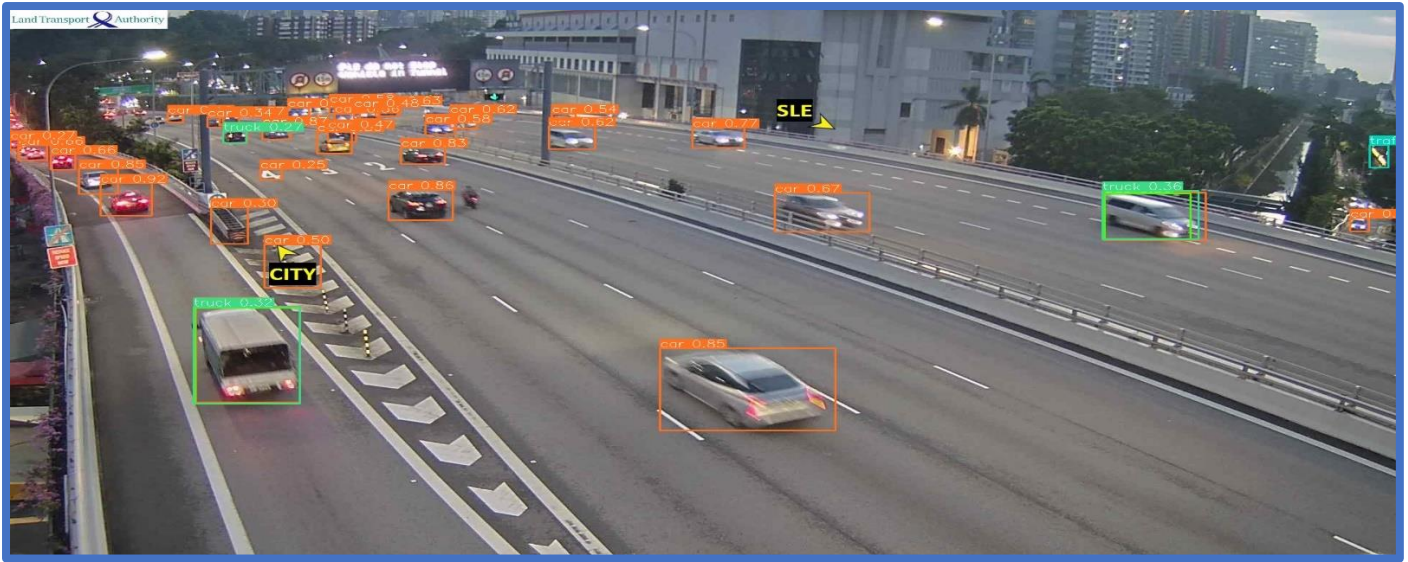


Figure 1: Model detection of vehicles

The results are then displayed on the interface hosted on local server (127.0.0.1) at port 8050. The interface consists of the following 2 main features:

### Feature 1: Map Page (127.0.0.1:8050/map)

This page displays an interactive map where each of the 87 cameras' position are marked using coloured markers based on their latitude and longitude (refer to figure 2 on the right). The markers are colour-coded accordingly to their probability of traffic jam, where high, medium, and low probability of traffic jam are red, orange, and green respectively. The colour code of the markers is assigned based on the latest images from LTA's API. They are updated when the images from DataMall are refreshed at every 5 minutes intervals. The objective of this page is to provide LTA engineers with an overarching view of the current road conditions of Singapore at each 5 minutes intervals; enabling them to identify locations where traffic jams are possibly going to be forming up soon (locations with cluster/line of yellow or red markers).

If a specific camera on the map forms a point of interest and requires further investigation, the user can click on the marker of the camera. Clicking on the camera will show a detailed description of that camera, which includes the image from the camera predicted on by the model, the directions of the road, as well as the normalized density and probability of traffic jam for each direction in the image (refer to figure 3 on the right).

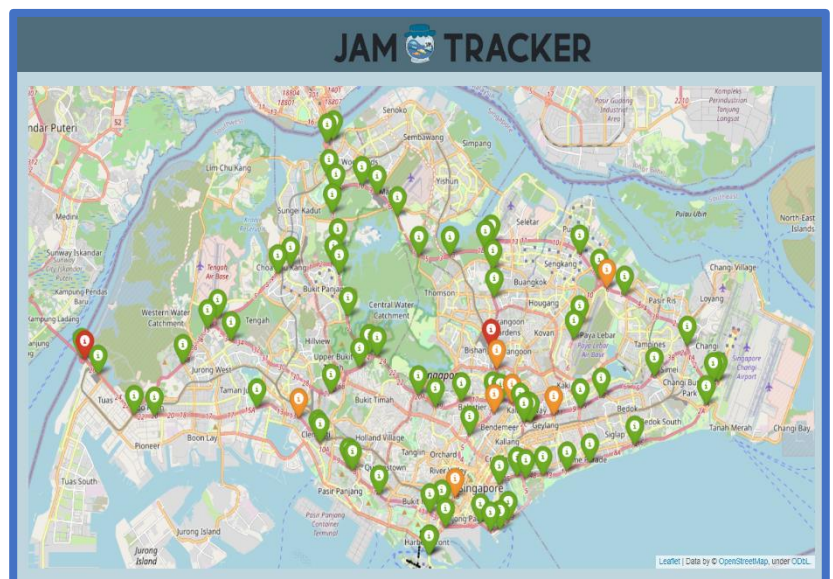


Figure 2: Map Page

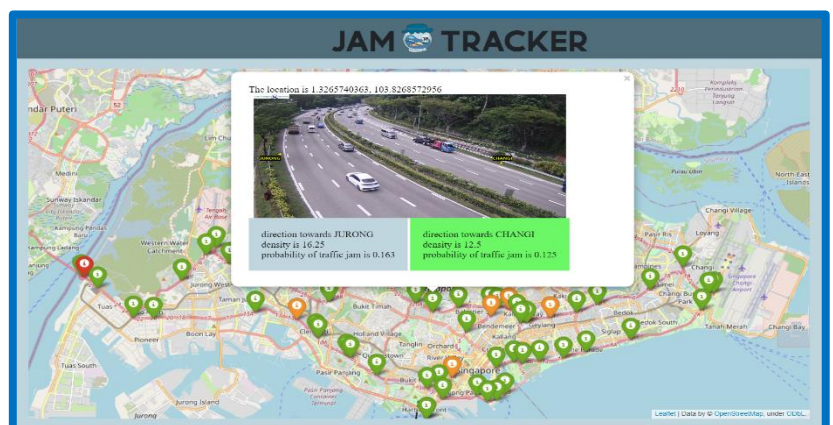


Figure 3: Detailed camera view on map



## Feature 2: Grid Page (127.0.0.1:8050/grid)

This page showcases the camera images from each of the 87 cameras, with each page currently displaying a maximum of 12 pages. Each of these cameras are colour-coded just like the markers on the map page, where red, yellow, and green represent high, medium, and low probability of traffic jam. The images are also sorted according to their probability of traffic jam, where the images with higher probability of traffic jam are displayed earlier. This allows users to easily identify and focus on the more important cameras first, saving the hassle of having to source through the pages. Each of these images are also tagged with their camera ID at the top left-hand corner of the image to allow users to identify which camera they are currently looking at; and in the event they want to focus on a single camera, they have a means of identification and do not have to identify based on the image surroundings. There are further planned improvements to be made to this page to greatly improve the user's experience and utility of this page (refer to the [Customizable Grid](#) in the Future Development Directions section).

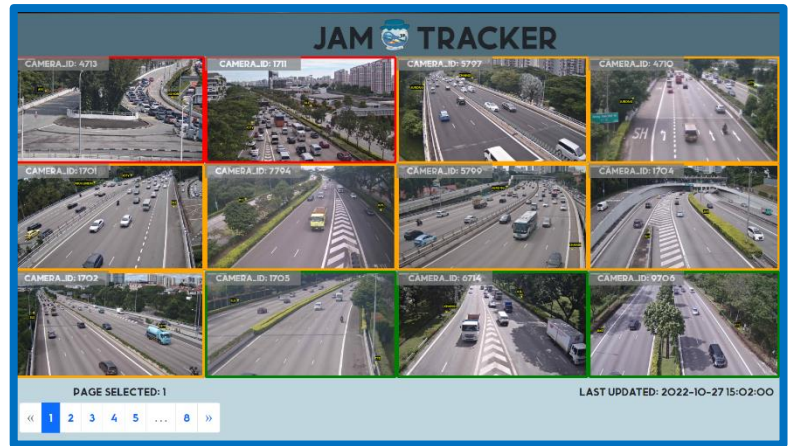


Figure 4: Grid page

Additionally, similar to the clickable markers on the map page, if the users have taken an interest in any of the cameras, they could click on that particular camera image, which will redirect them to a page that displays the detailed description of that camera (refer to Figure 5 on the right). The next top 4 images with the highest probability of traffic jam (excluding the camera with current point of interest) will also be displayed at the bottom of the page.

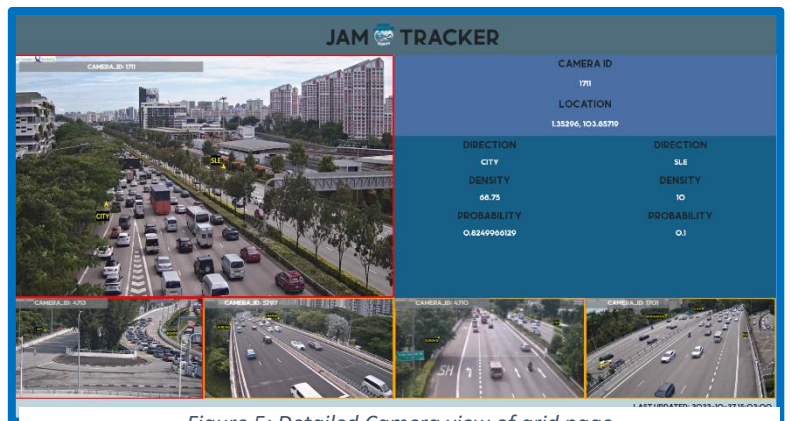


Figure 5: Detailed Camera view of grid page

## Home Page (127.0.0.1/)

To assist the users with understanding the utilities of the webpages and visualize the model's capabilities in detecting images, we added a home page (which is also the landing page) that provides a brief explanation of the model as well as an introduction to the different webpages on the site (refer to figure 6 below).

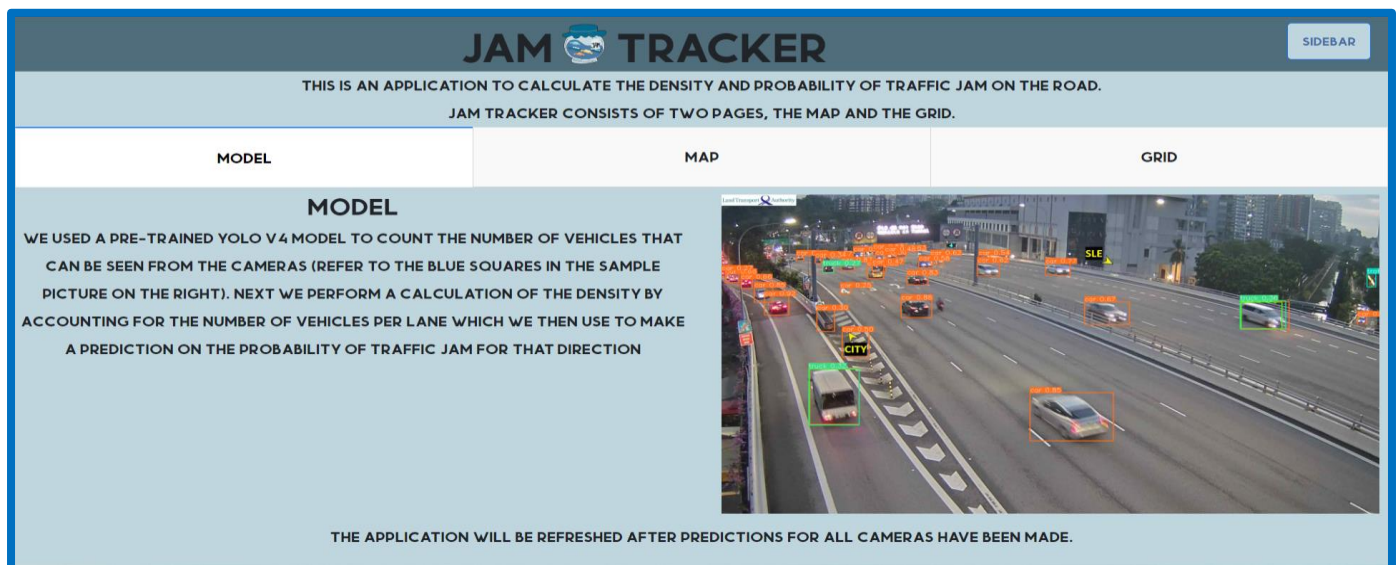


Figure 6: Landing Page (Home Page)

# Repository Breakdown

## Application Architecture

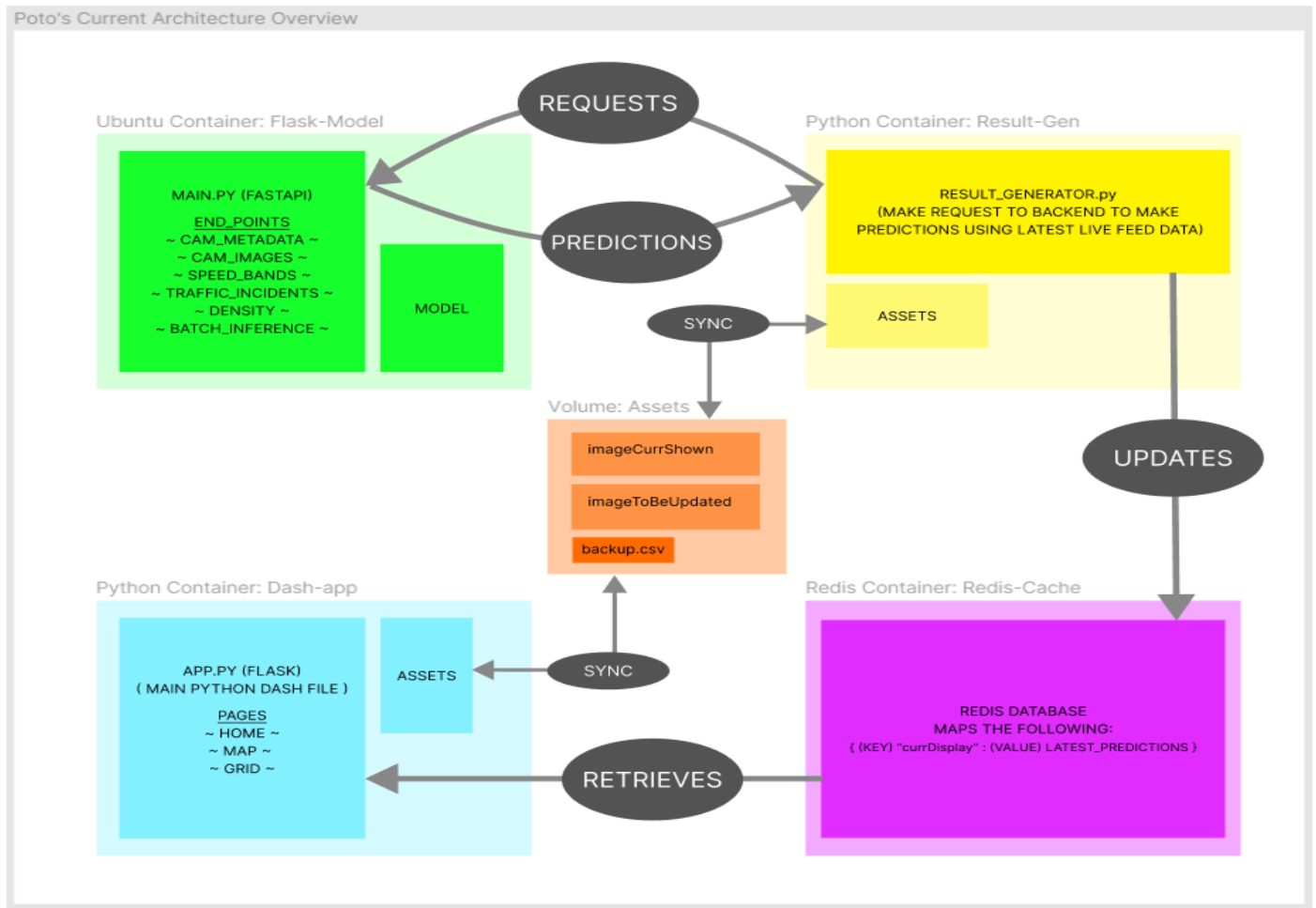


Figure 7: Current Application Architecture

## Repository Structure Breakdown

At the root directory, the folder is broken down into frontend and backend (refer to figure 8 on the right). It also contains the docker-compose.yml file which is used to set-up the whole application in docker with the current architecture shown in [figure 7](#) above. I will further elaborate on the “docker-compose.yml” file in the later parts of this [section](#). The backend folder contains the dockerfile required to create the flask-model<sup>1</sup> container and all the necessary source code to create the model, train the model, make predictions, and finally serve the model using FASTAPI on localhost port 5000 to receive request from other services. I will not be going into the details of the files, please refer to backend project group member technical report for more information.

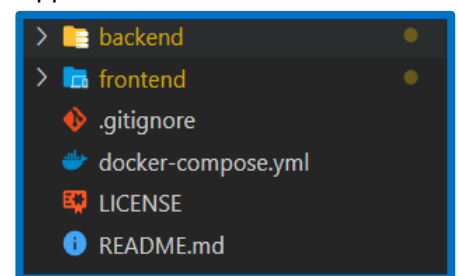


Figure 8: Root Directory

<sup>1</sup>(NOTE: Now uses **FASTAPI** instead of **Flask**, name not changed to ensure consistency between members’ report)

## Frontend detailed breakdown

The frontend is broken down into 2 separate important components. Firstly, it is the portion of the code containing the dash application, which serves the interface that the user will interact with for the application. Secondly, it is the portion of the code under the result\_generator folder, which makes requests to the backend server on port 5000 to make new predictions every minute and saves the result in the redis-cache database for retrieval by the dash application (refer to [figure 7](#) for a visualization of the described relationship). Each of these 2 components will form a container by themselves in the dockerize application, hence there is a dockerfile for each of them. This would be further elaborated in the [sub-section on “docker-compose.yml”](#).

## Dash-Application

The dash application is split into 4 main components, namely:

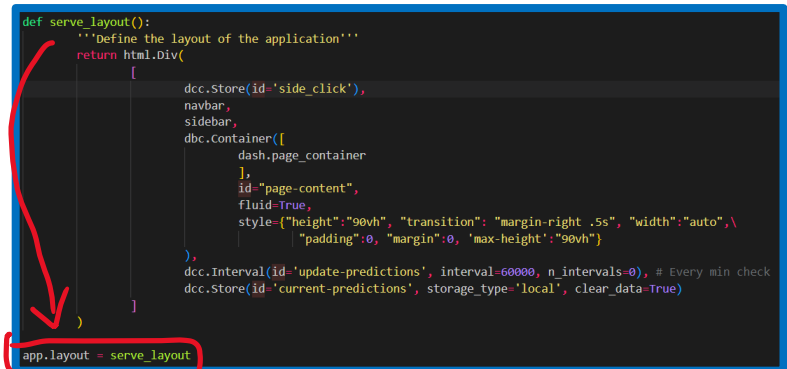
1. app.py (Contains the code of the main dash application)
2. components (Contains the code files of the common components used in the application)
3. pages (Contains the code files of the individual pages displayed in the application)
4. assets (Connected to the shared volume with result\_generator, contains the necessary resources)

### app.py

Inside app.py, both the flask server and dash application are initialized.

#### app.py :: Dynamic serve\_layout

Since the application is meant for serving live updates, the layout of the application must be dynamic and change everytime the page is loaded. As dash defaults to storing its app.layout in memory to save loading time, we have to set app.layout to a function that returns the actual layout as shown in figure 9 to achieve our intended goal. By setting app.layout to a function, the function will be called everytime the page is refresh and hence returns a fresh layout with the updated data.



```
def serve_layout():
    '''Define the layout of the application'''
    return html.Div(
        [
            dcc.Store(id='side_click'),
            navbar,
            sidebar,
            dbc.Container([
                dash.page_container
            ],
            id="page-content",
            fluid=True,
            style={"height": "90vh", "transition": "margin-right .5s", "width": "auto", \
                "padding": 0, "margin": 0, "max-height": "90vh"}
            ),
            dcc.Interval(id='update-predictions', interval=60000, n_intervals=0), # Every min check
            dcc.Store(id='current-predictions', storage_type='local', clear_data=True)
        ]
    )

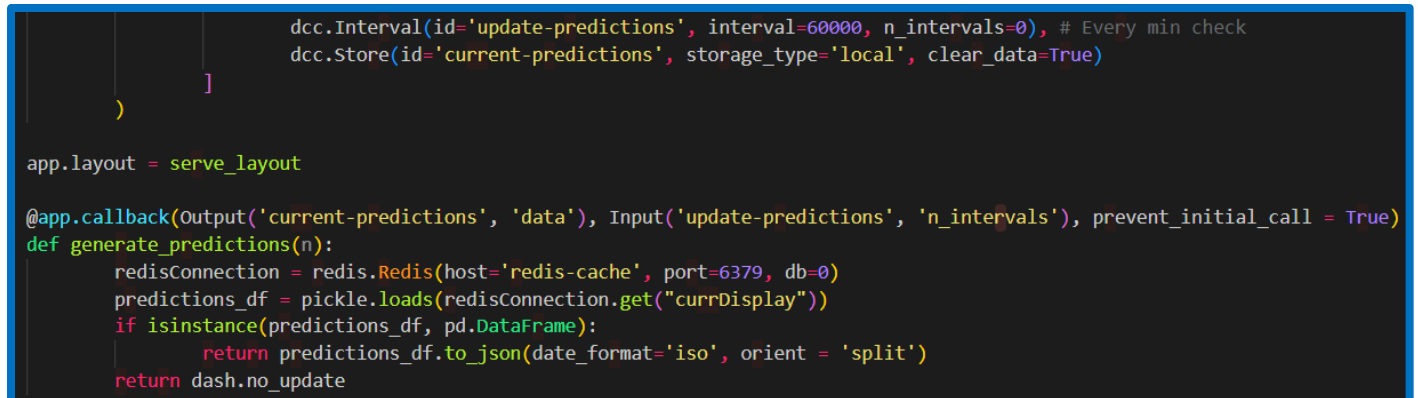
app.layout = serve_layout
```

Figure 9: Dynamic Serve Layout

#### app.py :: Multi-pages application

The dash application is initialized with an additional variable, namely “use\_pages” set to True such that the multiple page feature of the dash application is enabled. This couples with the dash.page\_container that is inside the serve\_layout function and the register\_page that we will see in the pages python file (will cover that in the later [sub-section on pages](#)). In summary, the page container keep tracks of the registered pages and display those pages when they are called upon, letting dash handles all the callbacks and URL routing.

#### app.py :: Updating predictions



```
    dcc.Interval(id='update-predictions', interval=60000, n_intervals=0), # Every min check
    dcc.Store(id='current-predictions', storage_type='local', clear_data=True)
]

app.layout = serve_layout

@app.callback(Output('current-predictions', 'data'), Input('update-predictions', 'n_intervals'), prevent_initial_call = True)
def generate_predictions(n):
    redisConnection = redis.Redis(host='redis-cache', port=6379, db=0)
    predictions_df = pickle.loads(redisConnection.get("currDisplay"))
    if isinstance(predictions_df, pd.DataFrame):
        return predictions_df.to_json(date_format='iso', orient = 'split')
    return dash.no_update
```

Figure 10: Updating Prediction Callback

As mentioned, since the application displays live data, we would have to update the data when the new predictions are up. Using dcc.Interval, setting interval = “60000” milliseconds, which is equal to a minute, we trigger the generate predictions callback to connect to the redis database to retrieve the dataframe of the new set of data. The dataframe contains key information like the link to the images, as well as the predictions required for displaying in the other pages interface. This data is then stored in dash storage and will be retrieved by other pages for display. Storage type is set to “local” to ensure that the data is persistent until a new set of predictions data replaces it.

## components

Components contain 2 files, `__init__.py` and `sidebar.py`. By importing each component into the init file, it allows for cleaner import by the main application where the components are used.

### components :: Sidebar (integrated navbar)

This contains the implementation of the sidebar with the integrated navbar. The main purpose of the sidebar is to allow the user to easily navigate between the pages. We added an intuitive button to toggle off the sidebar if the user wishes to have a full-size screen view of the page. The main implementation method here revolves around creating 2 styling for the sidebar and page component, one for when the sidebar is toggled OFF and other for toggled ON (refer to figure 11 on the right). The property of style in question here revolves around having a negative right margin for sidebar when it's toggled off and having an appropriate right margin the size of the sidebar for content when the sidebar is toggled on (right margin as sidebar is on the right).

```
@callback(  
    [output("sidebar", "style"),  
     output("page-content", "style"),  
     output("side_click", "data")],  
    [input("btn_sidebar", "n_clicks")],  
    State("side_click", "data"),  
)  
def toggle_sidebar(n, nclick):  
    if n:  
        if nclick == "SHOW":  
            sidebar_s = SIDEBAR_HIDDEN  
            content_s = CONTENT_HIDDEN  
            cur_nclick = "HIDDEN"  
        else:  
            sidebar_s = SIDEBAR_SHOWN  
            content_s = CONTENT_SHOWN  
            cur_nclick = "SHOW"  
    else:  
        sidebar_s = SIDEBAR_SHOWN  
        content_s = CONTENT_SHOWN  
        cur_nclick = "SHOW"  
    return sidebar_s, content_s, cur_nclick
```

Figure 11: Sidebar Toggle

## pages

This folder contains the pages that are to be displayed on the interface, where each file contains the code for a page. The init file here was not necessary since the pages are directly registered into the dash application, however we included it anyway if the purpose arise that it is required. Each page will contain a `dash.register_page` at the start of the code to register the page inside `dash.page_container` such that it can be call upon to display when accessed. The relative endpoint is parsed in to `register_page` and the URL routing will be settled by dash.

### pages :: home (path: /)

Since the home page is also the landing page of the application, there was no need to include any endpoint when registering the page. This page contains the introduction to the interface as well as some insights into the model used. We utilised tabs and keep the information concise such that it can be easily understood by the user, and such that they will not find it a hassle to read through. There are not much implementations details required to be explained here, hence I would not be elaborating on the details of the code.

### pages :: map (path: /map)

This page is registered with the endpoint "map". Main implementation of this page revolves around creating an additional html called `folium_map.html`, which is displayed when the page is called. Utilising the library `folium` and our established parameters required to see the whole of Singapore, we will retrieve the prediction data stored in the dash storage to create a new `folium_map.html` for display everytime there is an [update](#) in the storage using the `refresh_map` callback (refer to figure 12 on the right) . The cameras are then added as coloured markers onto the map, where their colours are determined by the predicted probability of traffic jam (the higher value if there are 2 directions). Then for each of these markers, they are encoded with their detailed data as a pop-up, which will appear when the markers are clicked upon. As observed in figure 12, we added a check requirement for data from the storage. If there are no data in the storage, to ensure some data would be displayed, we will read from our backup data instead.

```
@callback(Output('map', 'srcDoc'),  
          Input(component_id='current-predictions', component_property='data'))  
def refresh_map(json_data):  
    if json_data is not None:  
        df = pd.read_json(json_data, orient = "split")  
    else:  
        df = pd.read_csv(r"src/assets/backup.csv")  
    make_map(df)  
    return open('folium_map.html', 'r').read()
```

Figure 12: Creating the Map on Update

### pages :: grid (path: /grid)

This page is registered with endpoint "grid". The main implementation details on this page lies in the `create_card` function shown in figure 13 below, which is used to create a box for each image.

```
def create_card(img_src, cameraID, severity):  
    image = rf'assets/imageCurrShown/{img_src}'  
    return dbc.Card(  
        [dbc.CardImg(src=image, className = 'align-self-center', style={"max-height": "25vh", "height": "auto"}),  
         dbc.CardImgOverlay(  
             [html.H5(f"CAMERA_ID: {cameraID}", className="card-title", style={"color": "white",  
                 "background-color": "gray", "width": "50%", "opacity": 0.7}),  
              dbc.Button(href=f"http://localhost:8050/grid/detailed?main_picture={img_src}",  
                 style={"opacity": 0, "height": "100%", "width": "100%", "margin": 0, "padding": 0, "border": 0})  
             ], style = {"padding": 0, "margin": 0})  
         ], style = severity
```

Figure 13: Create Card Function



Similarly to the map page, when the dash storage is updated, it will trigger the callback to update\_images as seen in figure 14 by the code portion marked in yellow. If the dash storage is empty, it will load from the backup data instead. The dataframe is first sorted according to the probability of traffic jam from highest to lowest (marked in red on figure 14). Then based on the current page number selected, a specific window of size 12 of the dataframe will be selected for input to the create\_card function (marked in green on figure 14). A total of 12 cards will be created (filled using empty cards if insufficient images left) and updated into the grid with the output id listed above. Each card will be colour-coded with a border similarly to the markers on the map page and displayed accordingly. From figure 13, it can be observed that there is a button (marked by the code in purple). The styling of the button is set such that its enveloping the entire card, covering the image. However, its' opacity is then set to 0 such that its the image is visible, allowing us to stimulate the effect of the user clicking on the image that they will want to select. The link of the button is then set to be the gridDetailed page with the image reference link parsed in as a query string (marked in figure 13 by the red circle). This will allow us to keep track of which image is clicked upon by the user as we move to the gridDetailed page such that we can display what the user is interested in.

```
@callback(
    [Output("img_a1", "children"), #a1
     Output("img_a2", "children"), #a2
     Output("img_a3", "children"), #a3
     Output("img_a4", "children"), #a4
     Output("img_b1", "children"), #b1
     Output("img_b2", "children"), #b2
     Output("img_b3", "children"), #b3
     Output("img_b4", "children"), #b4
     Output("img_c1", "children"), #c1
     Output("img_c2", "children"), #c2
     Output("img_c3", "children"), #c3
     Output("img_c4", "children"), #c4
     Input("pagination", "active_page"),
     Input(component_id='current-predictions', component_property='data')]
)
def update_images(page, json_data):
    if json_data is not None:
        df = pd.read_json(json_data, orient="split")
    else:
        df = pd.read_csv("src/assets/backup.csv")
    imgSrcList = [[w, x, max(y, z)] for w, x, y, z in zip(df['imagefile'], df['CameraID'], df['prob1'], df['prob2'])]
    imgSrcList.sort(key=lambda xx:xx[2], reverse=True)
    if not page:
        page = 1
    startIndex = (page-1)*12
    endIndex = startIndex + 12
    if endIndex > len(imgSrcList):
        endIndex = len(imgSrcList) - 1
    imgToDisplay = [create_card(imgSrcList[imgNum][0], imgSrcList[imgNum][1], seriousness(imgSrcList[imgNum][2]))
                    for imgNum in range(startIndex, endIndex)]
    imgToDisplay += [[]] * (12 - len(imgToDisplay))
    return imgToDisplay
```

Figure 14: Updating the Grid

pages :: grid (path: /gridDetailed)

The implementation of gridDetailed is the same as the implementation of the grid page, with the only difference being the way we select the images for display. Firstly, we have to implement layout as a function such that we are able to retrieve the selected image reference link from the query string (refer to figure 15 below). After retrieving the image link, we would use that link to isolate that specific image row in the dataframe for card creation of a different styling, as well as its data for display (refer to figure 16). Then, the next top 4 highest probability traffic jams cameras for display (refer to figure 17) will be selected as per how it was done in grid page.

```
def layout(main_picture=None, **other_unknown_query_strings):
    return html.Div(
        [
```

Figure 15: Retrieving Query String value in Layout

```
# Isolating row of interest
main_value = 0
for i in range(len(imgSrcList)):
    if imgSrcList[i][0] == selected_img:
        main_value = i
        row = df[df['CameraID'] == imgSrcList[i][1]]
        break
```

Figure 16: Isolating main data from dataframe

```
# Main
imgToDisplay = [create_card(imgSrcList[main_value][0], imgSrcList[main_value][1], seriousness(imgSrcList[main_value][2]), True)]
detailed_data = [row['CameraID'].values[0], f'{row["Latitude"].values[0]}, {row["Longitude"].values[0]}', \
                 row['dir1'].values[0], round(row['density1'].values[0], 3), row['prob1'].values[0], \
                 row['dir2'].values[0], round(row['density2'].values[0], 3), row['prob2'].values[0]]

# Top 4 aside from main
for imgNum in range(len(imgSrcList)):
    if imgNum != main_value:
        imgToDisplay += [create_card(imgSrcList[imgNum][0], imgSrcList[imgNum][1], seriousness(imgSrcList[imgNum][2]))]
        if len(imgToDisplay) == 5:
            break
imgToDisplay += detailed_data
return imgToDisplay
```

Figure 17: Creating the different Grid Style

## assets

Assets contain the resource files required for styling as well as the images to be displayed. It is linked to docker volume of the same name and shared with the result-gen container. This folder contains the backup data, as well as the hard coded direction labels for the images and the newly created folium\_map from the map page. The 2 most important folders under assets are the imageCurrShown folder and the imageToBeUpdated folder. imageCurrShown folder contains the images of the 87 cameras that are on display in the interface at the current time. imageToBeUpdated acts as a temporary folder to store the newly downloaded images during the prediction process. This is done as the image

link provided by the LTA DataMall only lasts 5 minutes, thus, to ensure that the image is kept, it is downloaded into this folder. The images from imageToBeUpdated will then replace the images in imageCurrShown once the predictions are done and data in the redis database is updated so that the latest images and data shown on the interface will be consistent.

### Result-Generator

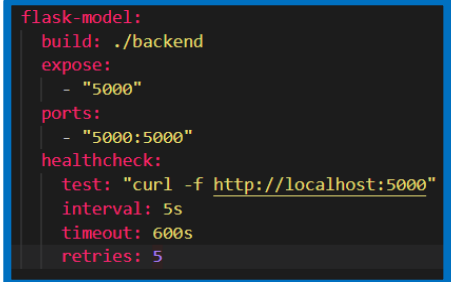
The result generator serves as “stimulator” that keeps the predictions and images fresh by constantly making requests to the backend server to make predictions on new images from LTA DataMall. To ensure resources are not wasted, the code will first check if the new images that we are requesting on is the same as the current images by comparing their image link since image links are unique. If the images are the same, the function will put itself to sleep for 60 seconds before once again making a new batch of prediction request to the backend server. If the images are not the same, it will call upon the function generate\_new\_set() to send request to backend to start new predictions. Firstly, when generate\_new\_set() is called, it will remove all images in the imageToBeUpdated folder. Afterwards, for each cameraID, the function will make a request to make prediction for that camera. At the same time, the image will be downloaded for that camera. The predicted data will be saved to backup.csv. Next, the function will empty the images in the imageCurrDisplay folder and replace them with the images from the imageToBeUpdated folder. Refer to the code in result\_generator.py as explanation is more clearly documented by chunk.

### Docker-compose.yml

As observed from [figure 7](#), there are 4 containers for this application.

Firstly, is the flask-model, which will run on port 5000. There is an additional health-check parameter for flask-model as we need to ensure that the flask-model’s server is always healthy such that result\_generator will be able to make request to the flask-model to generate new predictions.

Secondly, we have the dash-app, which will run on port 8050. The dash application will have to run with volume assets attached to the assets folder in src as it requires shared access to the images downloaded by result generator. Sharing a volume is more efficient then sending data across the network and hence this is more ideal.



```
flask-model:
  build: ./backend
  expose:
    - "5000"
  ports:
    - "5000:5000"
  healthcheck:
    test: "curl -f http://localhost:5000"
    interval: 5s
    timeout: 600s
    retries: 5
```

Figure 18: Docker-compose (Flask Model)

Thirdly, we have the redis-cache container, which will run on the port 6379. The purpose of the redis container is to store the dataframe of the new predictions created after result generator complete its request to the flask-model container. The stored dataframe will then be used to update the dash application data stored in the dash storage for propagation to all pages.

Lastly, we have the result-generator container. This container is initialized to run with volume assets attached to its assets folder as well such that the images it downloads, as well as the back-up file could be shared by the dash-app. This container will only start running after ensuring that the flask-model container is healthy. Once the container starts running, it is just set to keep running result\_generator.py to keep creating new predictions until stopped.

## Design Conceptualization

### Pre-Development

#### Interviews Outcome [User Research]

We formulated some general questions for the interview, separated mainly into drivers and engineers. Some of the questions are as follow:

#### Engineer

1. What do you think is the hardest part about being an engineer?

[Goal: Understanding their frustrations]

2. What do you think is the goal of an engineer?

[Goal: Understanding their responsibilities]



## Driver

1. What is their driving experience? (Frequency, feels about the traffic condition)

[Goal: Understanding of the current situation]

2. How often do you encounter traffic jams in Singapore? Where do you encounter traffic jam the most? How do they know about the traffic conditions?

[Goal: Understanding of the traffic conditions and efforts made by the engineers]

Initially before we carried out the interview, we created an imaginary persona from researching into LTA engineers via job portals and LTA website to help with brainstorming of features. From what we gathered from the interviews, I would say our personification was rather accurate and there were little changes to the persona. Insightful common response from the engineer's side regarding their frustrations were mainly on clearly identifying the problem due to their complexity linking to having many factors to consider. Drivers experience with traffic jams were not consistent, citing many different locations, which we thought was a good sign that our map interface is a step in the right direction as it can provide an overview on all areas.

## First phase user experience

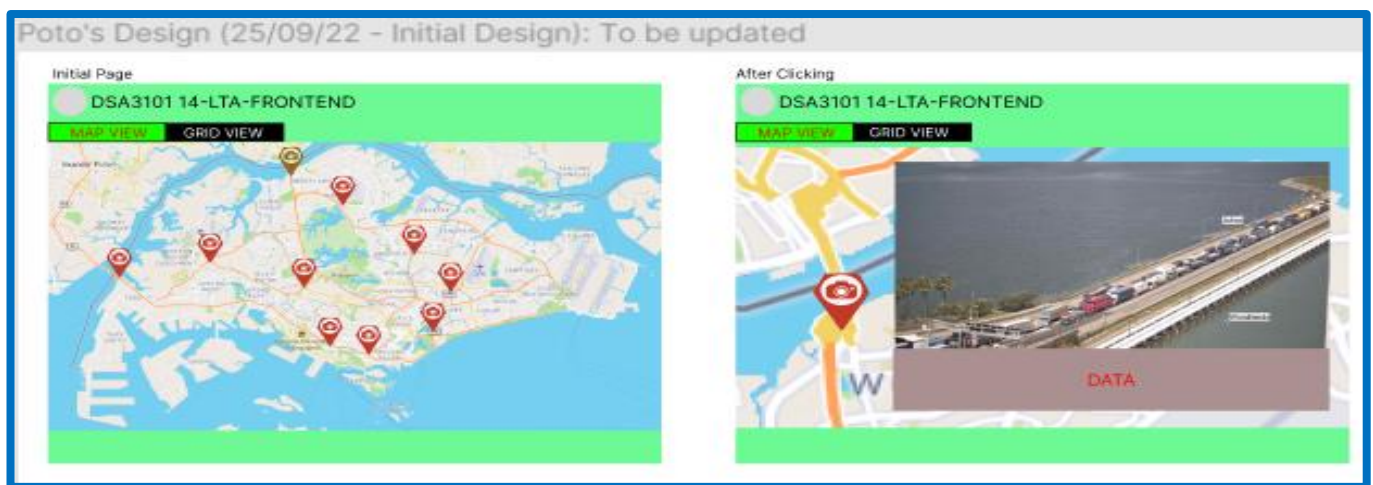


Figure 19: First Mid-Fidelity Wireframes

As a follow-up to the previous interviews, we checkback with our engineering friends what they feel about current features we are implementing and their feedback on the current design. Some insightful concerns raised were regarding the number of cameras that are going to be shown on the map, if too many would result in a very cluttered map, and possibly overwhelming to look at. Others talk about too bright and contrasting colours, as well as the hassle to have a visualization of the different cameras as multiple clicks must be made on the markers.

Developed after the interview:



Figure 20: Grid Mid-fidelity Wireframes

## Second phase user experience



Figure 21: Product Development done and functional

With the 2 main features (map and grid) ready and functional, we once again let the users try and click through to get their feedback on the current application, as well as whether they feel anything was missing from the application and what they think could be good complements to the application. While the general comment was compliment towards the application's design, we realised that the application wasn't that intuitive as the users needed to be guided by us to know what the application can do, other were curious about how the predictions were made. One of my friends who tested the application did question the implementation of the grid system. He asked what if the engineer wants to monitor specific cameras, how would he be able to do that if the pictures on the grid are not labelled and the cameras' position keep moving around according to probability? While I had intentions to make the grid customisable according to the engineer's preference, it was not implemented yet. However, to make immediate improvements to the user's experience of the feature, I added the cameraID label to each image.

Changes made after interview (Added in cameraID):



Figure 22: Grid Page after adding in camera ID



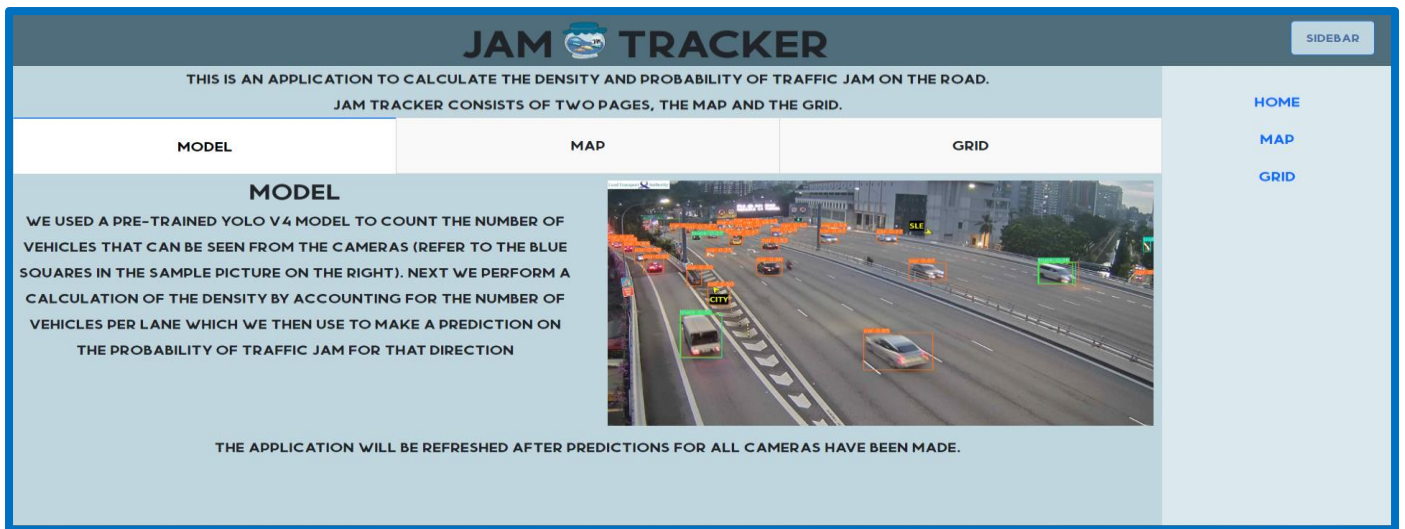


Figure 23: Home Page Developed

## Future Development Directions

### Architecture Changes

In week 12, the backend team implemented their own storage to store the latest images as well as the latest predicted result, which we can now simply retrieve via request from 2 new endpoints they created. However due to time constraints, the frontend team decided not to refactor the code. If refactor is done, we would be able to remove both the redis container and docker assets volume. Result generator would also be simplified to just making requests to FASTAPI-server to make predictions, no longer having to download the images, or storing the result in Redis db.

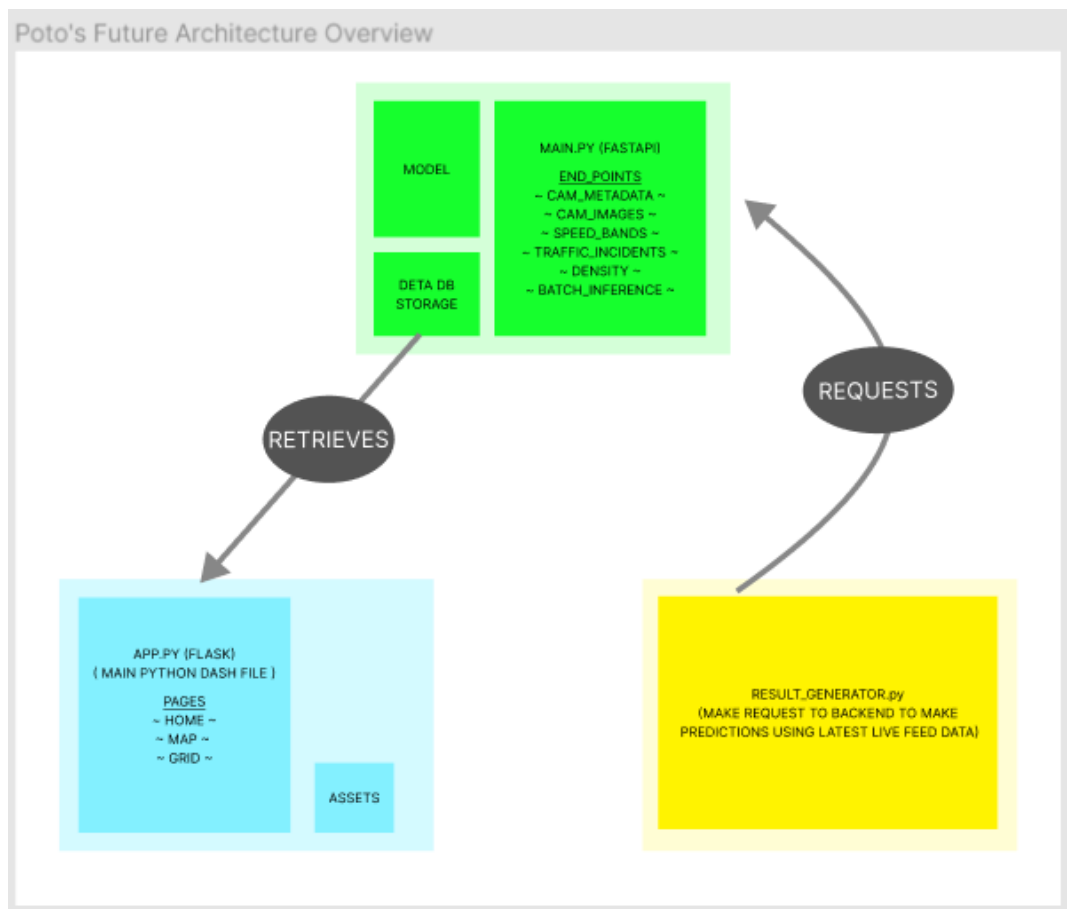


Figure 24: Future System Architecture



## Frontend

### Customizable Grid

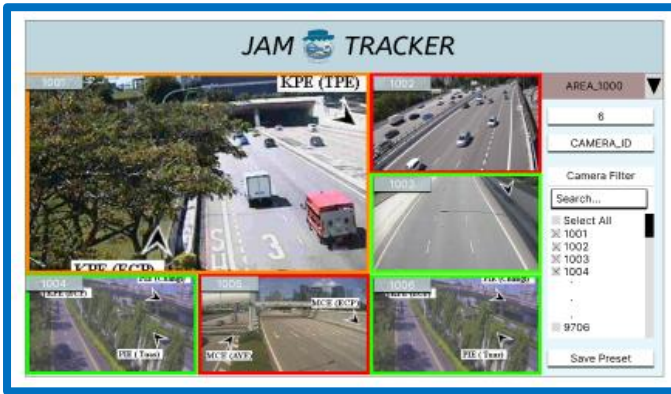


Figure 25: Customized by CameraID

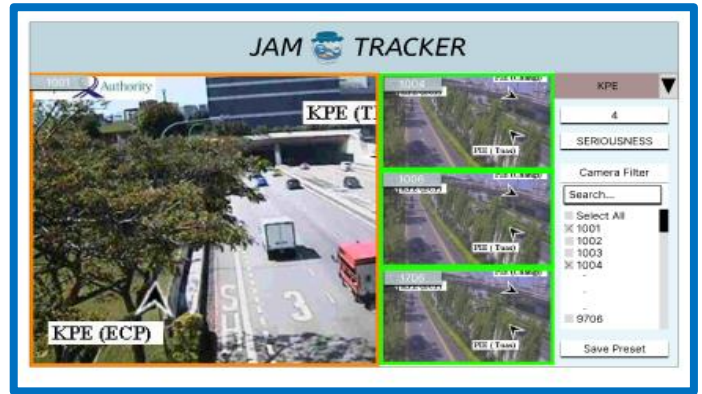


Figure 26: Customized by Area

As mentioned, to further improved the user's experience and utility of the grid page, we could introduce a customizable grid feature. As we know, road conditions can be unpredictable and change drastically in a short span of time. At those critical moments, having the clearest and most efficient way to monitor specific areas will be such a huge enabler in making strategic decisions. The user could do some preparations for such scenarios by creating several presets beforehand. For example, they could group cameras according to cameras id (refer to figure 25), then save a preset watchlist for it, and future usage will just be a single click away. Alternatively, they could even create a layout where cameras are group by certain key areas (refer to figure 26), save a preset with those cameras filtered out, and that too will just be a single click away as well. Classifying these pictures together might even help them identify trends or establish some relationships between them. Number of images per page, sorting key, selection of cameras are just some of the features that can utilised for customization.