**Maintenance Guide**

[**System Overview**](#_6mjl01tty5ge) **2**

[**Web Overview**](#_5o6zpvh4qtue) **3**

[Project Structure and Folder Layout:](#_jtroannfc15i) 3

[Key Code Snippets and Explanations](#_ctbsowu8p55) 4

[Run Instructions](#_kmj1l2u41cb3) 14

[**Machine Learning Model (Google Colab)**](#_dtew47sbawvy) **14**

[Environment Setup](#_xcnka0132mb8) 14

[Data Preparation](#_w35znrp0wasw) 16

[1.1 Input Sources](#_u2v9ulblhopk) 16

[1.2 Feature Extraction](#_i0xdgycf04o2) 17

[1.3 Data Processing Pipeline](#_wxmlayxazhib) 17

[Model Training](#_eva97k5qwddf) 19

[2.2 Training the Traffic Scheduler](#_mduxokpe6mro) 19

[2.3 Training the Forecast Models](#_40z6s5glv48d) 20

[3. Forecasting & Inference](#_dgl8gccraqy5) 21

[3.1 Real-Time Traffic Light Decision](#_7s8a7phr4ckl) 21

[3.2 Forecasting for the Next Few Hours](#_yj4ovwkh6yux) 22

[4. Weather Data Integration](#_kaa63ubg8hhj) 24

[4.1 Getting the Weather Data](#_z6gepqdz8v1m) 24

[4.2 How We Detect “Severe” Weather](#_99yh08kbufyu) 24

[5. Main Prediction Loop](#_t0cvxvh350il) 25

[5.1 Helper Functions](#_uuqibbb930j) 25

[5.2 The Live Loop](#_38ol1kmb4j2o) 26

[Notes](#_wnthjzjw7c2q) 28

[**Firebase Structure Overview**](#_e323w2s22e67) **28**

## 

## 

## 

## 

## System Overview

The smart traffic light system combines hardware components, a machine learning model, cloud infrastructure, and a web-based dashboard to control and monitor an intersection in real time. Each part of the system works together to detect vehicles and pedestrians, make decisions, and update the traffic lights accordingly.

At the heart of the system is the M5Stack Core2, which acts as the main controller. It receives input from ultrasonic sensors (for detecting cars) and an AI camera (used to count or detect pedestrians). The Core2 processes this data and sends it to Firebase, which serves as the central database.

Based on the latest sensor data, a machine learning model hosted on Google Colab calculates which lights should turn green. This decision is written back to Firebase.

To control the physical traffic lights, the Core2 sends commands to Atom Lite microcontrollers using MQTT (a lightweight real-time messaging protocol). Each Atom Lite unit controls the LEDs for one of the traffic lights.

The web application ties everything together. It reads live data from Firebase to display:

* The current traffic light states
* Vehicle and pedestrian activity
* System decisions and logs

It also includes a 3D simulation of the intersection, along with graphs and analytics to help city operators understand traffic patterns and system performance.

All data flows through Firebase, making it the main point of coordination between the hardware, machine learning model, and web dashboard. Understanding how these parts interact is essential for maintaining and troubleshooting the system.

## Web Overview

The project uses the following technologies: HTML, CSS, and JavaScript.

The web application is divided into two functional modules:

* Real-time Dashboard – Displays current traffic-light states, sensor readings, pedestrian counts, weather conditions, and forecast charts, all pulled from Firebase Realtime Database.
* 3D Simulation – A Three.js scene that visualizes vehicles, pedestrians, roads, and traffic lights, updating continuously according to the same live data.

### Project Structure and Folder Layout:

***src/***

***│***

***├── assets/ # CSS styles***

***│ ├── dashboard-layout.css # Grid and layout for dashboard***

***│ ├── style-sidebar.css # Sidebar styling***

***│ └── style.css # Base global styles***

***│***

***├── components/ # Simulation engine (3D + controls)***

***│ ├── cars.js # Spawning and updating cars***

***│ ├── lights.js # Traffic light logic***

***│ ├── pedestrians.js # Pedestrian 3D logic***

***│ ├── responsive-sidebar.js # ☰ menu toggle logic***

***│ ├── road.js # Scene setup for roads***

***│ └── theme.js # Dark mode toggle***

***│***

***├── dashboard/ # UI cards and data-driven charts***

***│ ├── daytime.js # Displays current time/date***

***│ ├── forecast-pedestrian-chart.js***

***│ ├── forecast-traffic-chart.js***

***│ ├── greenlight.js # Green light state display***

***│ ├── pedestrians.js # Pedestrian count display***

***│ ├── sensors.js # Sensor data summary***

***│ ├── traffic.js # Traffic density display***

***│ └── weather.js # Weather data***

***│***

***├── public/***

***│ └── index.html # Main UI layout + canvas root***

***│***

***├── firebase.js # Firebase initialization***

***├── main.js # Simulation controller (entry point)***

***├── package.json***

***└── package-lock.json***

### Key Code Snippets and Explanations

This section provides focused and practical explanations for the key code snippets captured from the system.

The goal is to help future maintainers quickly understand how the essential logic works, such as vehicle movement, Firebase data retrieval, and forecast handling, so they can debug, modify, or expand the system with confidence.

Each snippet was carefully selected to represent a core functionality in either the dashboard or the simulation.

The web application is divided into two main views: the dashboard and the 3D simulation. The following sections explain each one in detail.

1. **Dashboard**

In this system, each information card displayed on the dashboard is managed by a dedicated JavaScript file located in the dashboard/ directory. Each file is responsible for retrieving the relevant data, processing it, and updating the corresponding card's content on the screen.

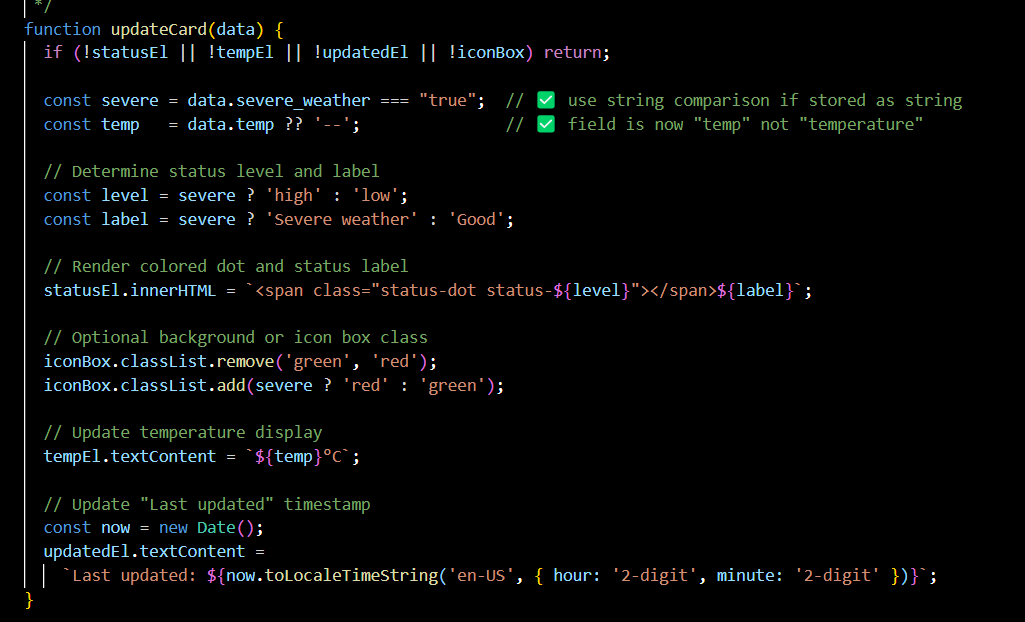
For example:

* dashboard/weather.js – updates the weather card
* dashboard/traffic.js – handles the traffic density card
* dashboard/pedestrians.js – displays the current number of pedestrians
* and so on...

To help the maintainer understand where the key logic resides in the code, I have included annotated screenshots of important code segments throughout this document.

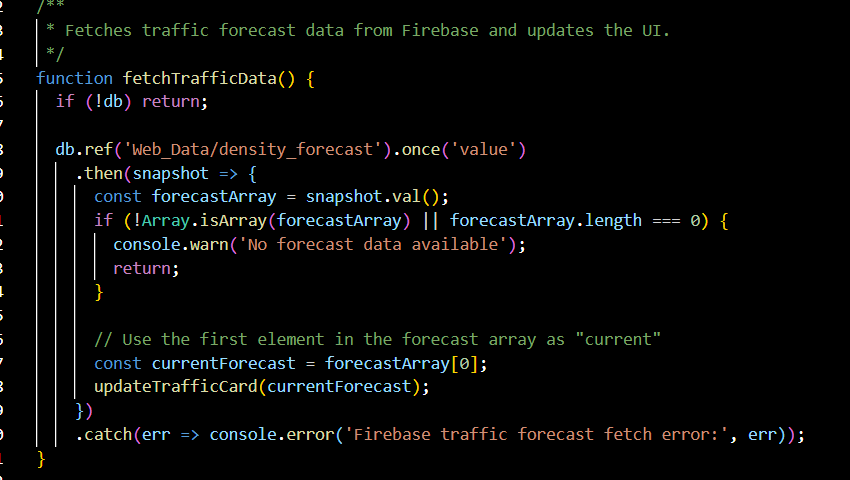
**dashboard/**[**weather.js**](http://weather.js)**:**

The updateCard(data) function updates the weather card on the dashboard based on data received from Firebase. It checks whether the weather is marked as severe (severe\_weather) and accordingly updates the status label, color (green/red), temperature value, and visual indicators. It also displays the last updated time. If any of the required DOM elements are missing, the function exits early without making changes.



**dashboard/**[**traffic.js**](http://traffic.js)**:**

This code snippet is responsible for retrieving the traffic density forecast from Firebase and displaying it on the dashboard. The fetchTrafficData function reads data from the Web\_Data/density\_forecast path, extracts the forecast information, and, if available, updates the user interface via the updateTrafficCard function. An initial fetch is performed when the page loads, and new forecasts are retrieved every 10 seconds thereafter.



**dashboard/**[**sensors.js**](http://sensors.js)**:**

This code snippet monitors the live status of all connected sensors in the system. The updateSensorCard function updates the dashboard UI to indicate whether all four ultrasonic sensors and the pedestrian camera are currently connected. It uses a simple validation check to determine if all relevant keys (ultra1 to ultra4 and Ped\_Num) exist in the Firebase Realtime Database. If so, it displays a green checkmark; otherwise, a red “Disconnected” indicator.

The fetchSensorStatus function retrieves this data every 10 seconds from the Web\_Data path in Firebase, ensuring that the dashboard remains synchronized with the real-world sensor status.



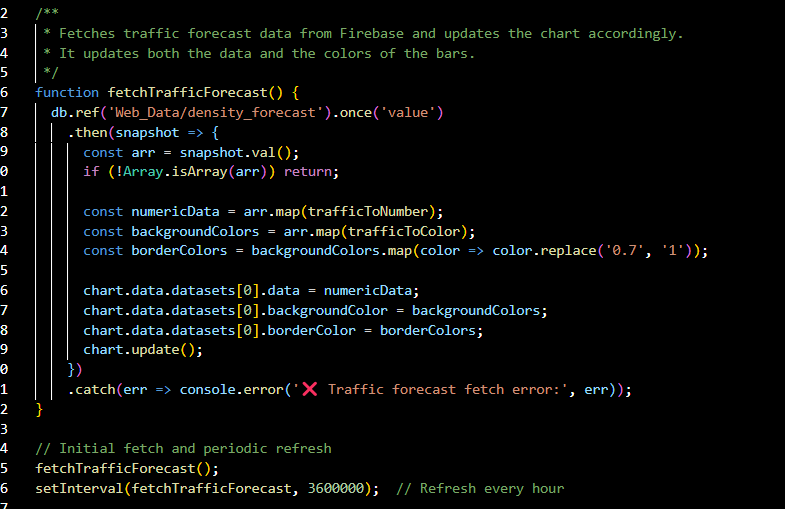
**dashboard/**[**pedestrians.js**](http://pedestrians.js)**:**

This file (pedestrians.js) handles the live display of the current pedestrian count on the dashboard. It fetches the value of face from the root of the Firebase Realtime Database and updates the relevant card in the UI. In addition to showing the current number, it also displays the time of the last update. The fetch is repeated every 10 seconds, so the number stays current. This script is essential for real-time monitoring of pedestrian presence at the intersection, it's not about forecasts, just what’s happening right now.



**dashboard/**[**forecast-traffic-chart.js**](http://forecast-traffic-chart.js)**:**

This function is responsible for fetching the forecasted traffic levels from the Firebase Realtime Database and updating the chart in the dashboard. The data is retrieved from the density\_forecast array, which contains string values like "low", "medium", and "high". The script maps these strings to numeric values for the Y-axis, and also assigns corresponding bar colors, green for low traffic, yellow for medium, and red for high. This function runs automatically every hour to ensure the chart always reflects the latest forecast.



**dashboard/**[**forecast-pedestrian-chart.js**](http://forecast-pedestrian-chart.js)**:**

This code displays a real-time forecast of pedestrian density using a line chart. The chart is generated with Chart.js and updated with data pulled from Firebase Realtime Database. The fetchPedestrianForecast function retrieves a forecast array from the database path Web\_Data/pedestrian\_forecast and populates the chart’s dataset. The data refreshes automatically every hour, ensuring that the dashboard presents up-to-date predictive analytics for pedestrian flow.

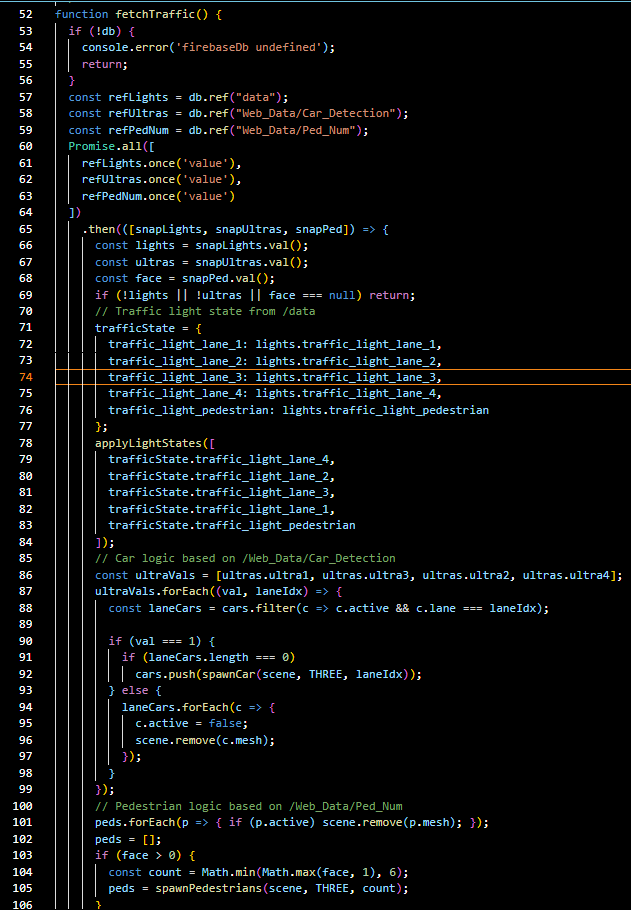


**2. Simulation**

The following code snippets implement the visual simulation of the intersection, as part of the smart traffic control system. The simulation is driven by real-time data from Firebase and includes a graphical representation of vehicles, pedestrians, and traffic lights. The implementation uses the Three.js library, with update mechanisms that simulate movement and timing according to the system’s live behavior.

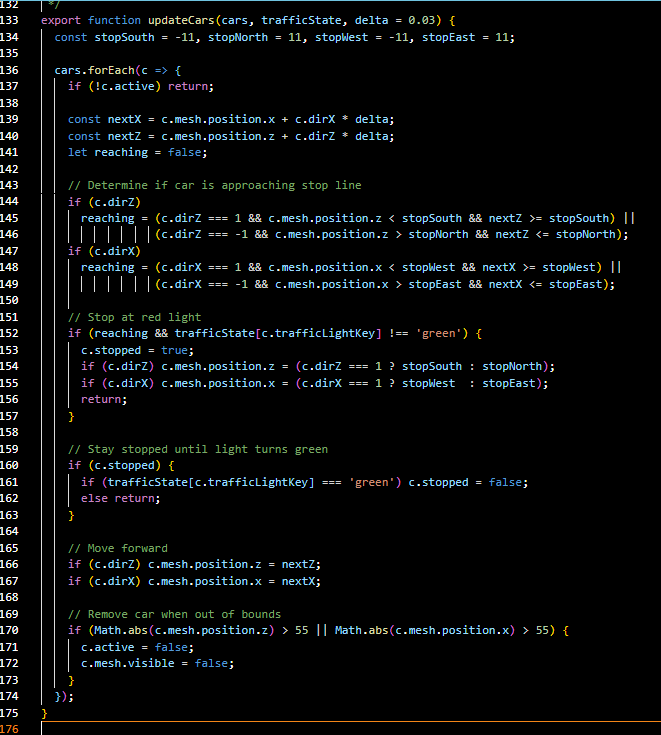
[**main.js**](http://main.js):

The fetchTraffic function synchronizes real-time sensor data with the visual simulation of the intersection. It retrieves the current traffic light states, ultrasonic vehicle detection values, and pedestrian count from the Firebase database, and updates the scene accordingly. The traffic light states are passed to the applyLightStates function to visually update the signals, vehicles are added or removed based on the sensor input, and pedestrians are rendered if detected by the camera. This function runs every ten seconds to ensure that the simulation accurately reflects the current state of the intersection.



**components/**[**cars.js**](http://cars.js)**:**

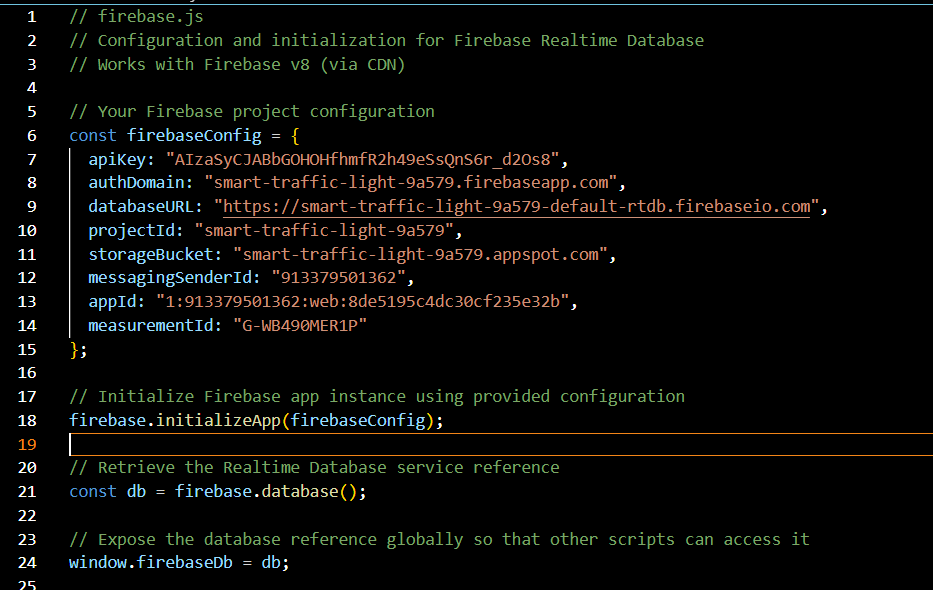
This function manages the movement and stopping behavior of each car in the simulation. Each car moves forward in its assigned lane unless it approaches a red traffic light, in which case it stops at a predefined position. The function continuously checks whether the car is near a stop line and evaluates the traffic light status using the trafficState object. Once the light turns green, the car resumes movement. If a car exits the visible area, it is marked as inactive and hidden from the scene.



**Firebase Initialization and Read**

[**firebase.js**](http://firebase.js)**:**

This code initializes the connection to Firebase Realtime Database using the project's configuration. It sets up the Firebase app instance and exposes the database reference globally (window.firebaseDb) so that other scripts in the project can access it without reinitializing Firebase. This file is loaded first to ensure that all modules depending on live data (such as the dashboard or simulation) can retrieve real-time updates consistently.



### Run Instructions

1. Open the project folde**r** in [Visual Studio Code](https://code.visualstudio.com/).
2. Install the required dependencies by running the following commands in the terminal:

* npm install three
* npm install firebase

1. Navigate to the public/ directory and open the index.html file.
2. Right-click on index.html and select “Open with Live Server” (requires the [Live Server extension](https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer) in VS Code).
3. A new browser window will open, displaying the dashboard interface and the 3D traffic intersection simulation.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## Machine Learning Model (Google Colab)

### Environment Setup

**Table Of contents (Our Colab Notebook):**

smart-traffic-light/

├── Run

├── Imports and Installations

├── DataBase Integration/

│ ├── Initialize Firebase

│ ├── Read data from Firebase

│ └── Write predictions or state updates

├── Weather API Integration/

│ ├── Fetch current weather and API

│ ├── Calculate if weather is severe

│ └── Check and return boolean result

├── Real-Time Prediction and Inference/

│ └── Run prediction

├── ML model - Random Forest/

│ ├── Data Processing Function

│ ├── Model Training + Evaluation Function

│ ├── Format and Display Results

│ └── Full Pipeline Wrapper

└── Pedestrians and Density Forecast/

├── Load and Preprocess Data

├── Train Models (People Count + Traffic Density)

└── Predict for Next N Hours

**Libraries :**

import pandas as pd

import numpy as np

import json

import time

import joblib

import matplotlib.pyplot as plt

import seaborn as sns

from datetime import datetime, timedelta

import pytz

import requests

import firebase\_admin

import os

from IPython.display import clear\_output

from firebase\_admin import credentials, db

from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor

from sklearn.multioutput import MultiOutputClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix, mean\_absolute\_error

### What Each Library Is Used For:

* pandas / numpy: Handle data loading, manipulation, and array operations.
* json / time / datetime / pytz: Deal with time formatting, timestamps, and JSON I/O.
* joblib: Save and load trained models as .pkl files.
* matplotlib / seaborn: Create plots and visualizations like confusion matrices.  
  os / clear\_output: Control system-level functions and clean output cells in Colab.
* requests: Make API calls (e.g., to weather API or Firebase REST endpoints).
* firebase\_admin: Interact with Firebase Realtime Database using admin credentials.
* sklearn (scikit-learn):
  + RandomForestClassifier, RandomForestRegressor: Main ML models used.
  + MultiOutputClassifier: For handling multi-label classification.
  + train\_test\_split: Splitting data for training/testing.
  + LabelEncoder: For encoding string labels.
  + classification\_report, confusion\_matrix, accuracy\_score: Evaluation metrics.

### Data Preparation

This section outlines how raw traffic data is processed into a structured format suitable for training and inference using the machine learning model.

#### **1.1 Input Sources**

* Live data is pulled from Firebase, including:  
  + ultra1–4: Binary features indicating the presence of vehicles using ultrasonic sensors.
  + face: Estimated pedestrian count from camera analysis.
  + severe weather: Boolean flag based on weather conditions.
* Offline data used for training is loaded from a local JSON file.

#### **1.2 Feature Extraction**

The following function extracts relevant inputs from Firebase:

def extract\_inputs\_for\_prediction(firebase\_data):

source = firebase\_data.get("data", firebase\_data)

try:

return {

"ultra1": source["ultra1"],

"ultra2": source["ultra2"],

"ultra3": source["ultra3"],

"ultra4": source["ultra4"],

"face": source["faces"],

}

except KeyError as e:

print(f"❌ Missing key: {e}")

return None

#### **1.3 Data Processing Pipeline**

Offline training data is cleaned and converted to numeric form. Key transformations include:

* Mapping "cars"/"pedestrians" to priority labels (0/1)
* Mapping "red"/"green" light states to 0/1
* Converting "good"/"bad" labels to binary values

def process\_data(input\_file):

"""Load and process data from JSON file, including 'severe weather' and 'label'."""

with open(input\_file, "r") as file:

data = json.load(file)

df = pd.DataFrame(data)

# Define mappings

priority\_map = {"cars": 0, "pedestrians": 1}

traffic\_light\_map = {"red": 0, "green": 1}

label\_map = {"bad": 0, "good": 1}

# Apply mappings

df["priority"] = df["priority"].map(priority\_map)

for col in [

"traffic\_light\_lane\_1", "traffic\_light\_lane\_2",

"traffic\_light\_lane\_3", "traffic\_light\_lane\_4",

"traffic\_light\_pedestrian"

]:

df[col] = df[col].map(traffic\_light\_map)

df["severe weather"] = df["severe weather"].astype(int)

df["label"] = df["label"].map(label\_map)

# Input features

X = df[["ultra1", "ultra2", "ultra3", "ultra4", "face", "severe weather"]]

# Output targets (including label)

y = df[[

"priority",

"traffic\_light\_lane\_1", "traffic\_light\_lane\_2",

"traffic\_light\_lane\_3", "traffic\_light\_lane\_4",

"traffic\_light\_pedestrian",

"label"

]]

return X, y

And this how we process the data for the forecasting :

def load\_and\_preprocess\_forecast\_data(file\_path="data\_r.json"):

with open(file\_path, "r") as file:

data = json.load(file)

df = pd.DataFrame(data)

# Encode categorical variables

label\_encoders = {}

for col in ["day", "traffic density"]:

le = LabelEncoder()

df[col] = le.fit\_transform(df[col])

label\_encoders[col] = le

# Convert time string to hour

df["hour"] = pd.to\_datetime(df["time"], format="%H:%M").dt.hour

return df, label\_encoders

### Model Training

This section describes how the Random Forest model was trained using the labeled traffic data and how the forecasting models were prepared using time-based features.

#### 2.2 Training the Traffic Scheduler

To train the model that decides which traffic light should turn green, we used labeled data (saved as data.json) with features like:

* Vehicle presence from 4 ultrasonic sensors
* Pedestrian count
* Weather condition

Training steps:

def train\_scheduler(X, y):

"""Train a Multi-Output Random Forest Classifier and return the model and test results."""

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42 , stratify=y)

# Train model

model = MultiOutputClassifier(RandomForestClassifier(n\_estimators=300, random\_state=42))

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

# Evaluate accuracy

print("📊 Prediction Accuracy:")

for i, col in enumerate(y.columns):

acc = accuracy\_score(y\_test.iloc[:, i], y\_pred[:, i])

print(f"{col}: {acc:.2f}")

print("\n📝 Sample Classification Report (Lane 1):")

print(classification\_report(y\_test.iloc[:, 2], y\_pred[:, 2])) # Lane 1 = index 2

# Save model

joblib.dump(model, "traffic\_model.pkl")

print("\n✅ Model saved as 'traffic\_model.pkl'")

return model, X\_test, y\_test, y\_pred

We train the model, evaluate how well it does on the test set, and save it for later use during live prediction.

#### 2.3 Training the Forecast Models

The system also includes short-term forecasting for both pedestrian flow and traffic density based on time and sensor data.

The train\_forecast\_models function handles this:

def train\_forecast\_models(df):

# Shared input features

features = ["ultra1", "ultra2", "ultra3", "ultra4", "hour", "day"]

# People Count – Regression

X\_p = df[features]

y\_p = df["face"]

X\_train\_p, X\_test\_p, y\_train\_p, y\_test\_p = train\_test\_split(X\_p, y\_p, test\_size=0.2, random\_state=42)

people\_model = RandomForestRegressor(n\_estimators=100, random\_state=42)

people\_model.fit(X\_train\_p, y\_train\_p)

print("📊 MAE (People Count):", mean\_absolute\_error(y\_test\_p, people\_model.predict(X\_test\_p)))

# Traffic Density – Classification

X\_d = df[features]

y\_d = df["traffic density"]

X\_train\_d, X\_test\_d, y\_train\_d, y\_test\_d = train\_test\_split(X\_d, y\_d, test\_size=0.2, random\_state=42)

density\_model = RandomForestClassifier(n\_estimators=100, random\_state=42)

density\_model.fit(X\_train\_d, y\_train\_d)

print("📊 Accuracy (Traffic Density):", accuracy\_score(y\_test\_d, density\_model.predict(X\_test\_d)))

return people\_model, density\_model

The forecasting models use:

* The same sensor data (ultra1–4)
* Day of the week
* Hour of the day

One model predicts how many pedestrians are likely to show up in the next few hours, and the other predicts how heavy traffic is expected to be.

### 3. Forecasting & Inference

Once the models are trained, the system uses them in two ways:

1. To predict traffic light actions in real-time based on sensor input
2. To forecast traffic density and pedestrian count for the next few hours

#### 3.1 Real-Time Traffic Light Decision

The model takes input from four ultrasonic sensors, pedestrian count (from the camera), and a weather flag. It processes this input and returns the predicted actions for each traffic light, along with the priority (vehicles or pedestrians) and an overall scenario label ("good"/"bad").

Here's how the prediction works:

def predict\_traffic\_decision\_with\_label(model, ultra1, ultra2, ultra3, ultra4, face, severe\_weather):

"""

Predicts traffic light actions and scenario label with preprocessing for ultra1–4.

Distances under 55 are treated as '1' (car), otherwise '0'.

"""

# Preprocess: convert raw distance to binary presence

def to\_binary(distance):

return 1 if distance < 55 else 0

sample\_input = pd.DataFrame([{

"ultra1": to\_binary(ultra1),

"ultra2": to\_binary(ultra2),

"ultra3": to\_binary(ultra3),

"ultra4": to\_binary(ultra4),

"face": face,

"severe weather": severe\_weather

}])

# Predict

predicted = model.predict(sample\_input)[0]

# Map outputs

light\_map = {0: "red", 1: "green"}

label\_map = {0: "bad", 1: "good"}

output\_labels = ["priority", "traffic\_light\_lane\_1", "traffic\_light\_lane\_2", "traffic\_light\_lane\_3", "traffic\_light\_lane\_4", "traffic\_light\_pedestrian", "label"]

result = {}

for label, value in zip(output\_labels, predicted):

if label == "priority":

result[label] = "pedestrians" if value == 1 else "cars"

elif label == "label":

result[label] = label\_map.get(value, f"unknown ({value})")

else:

result[label] = light\_map[value]

return result

This function is typically called during the real-time loop that runs every few seconds, making live decisions.

#### 3.2 Forecasting for the Next Few Hours

The system also includes a short-term forecast to help anticipate changes in pedestrian and vehicle traffic over the next few hours. It uses average sensor values and current time features to make predictions hour by hour.

def predict\_forecast\_separately(people\_model, density\_model, label\_encoders, df, hours\_ahead=5):

"""

Predicts people count and traffic density for the next N hours.

Returns two separate lists:

- people\_predictions: list of dicts with time and predicted people count

- traffic\_predictions: list of dicts with time and predicted traffic density

"""

# Average sensor values

avg\_ultras = {f"ultra{i}": df[f"ultra{i}"].mean() for i in range(1, 5)}

israel\_time = datetime.now(pytz.timezone("Asia/Jerusalem"))

current\_hour = israel\_time.hour

most\_common\_day = df["day"].mode()[0]

# Prepare future time features

future\_hours = [(current\_hour + i) % 24 for i in range(hours\_ahead)]

future\_data = pd.DataFrame({

"ultra1": [avg\_ultras["ultra1"]] \* hours\_ahead,

"ultra2": [avg\_ultras["ultra2"]] \* hours\_ahead,

"ultra3": [avg\_ultras["ultra3"]] \* hours\_ahead,

"ultra4": [avg\_ultras["ultra4"]] \* hours\_ahead,

"hour": future\_hours,

"day": [most\_common\_day] \* hours\_ahead

})

# Make predictions

people\_preds = people\_model.predict(future\_data)

density\_preds = density\_model.predict(future\_data)

decoded\_density = label\_encoders["traffic density"].inverse\_transform(density\_preds)

# Build output lists

people\_predictions = []

traffic\_predictions = []

for i in range(hours\_ahead):

future\_time = (israel\_time + timedelta(hours=i)).strftime("%H:%M")

people\_predictions.append({

"time": future\_time,

"people\_count": round(float(people\_preds[i]), 2)

})

traffic\_predictions.append({

"time": future\_time,

"traffic\_density": decoded\_density[i]

})

# Optional: print each prediction

print(f"\n📅 {future\_time} Prediction:")

print(f"👥 People Count: {people\_preds[i]}")

print(f"🚦 Traffic Density: {decoded\_density[i]}")

return people\_predictions, traffic\_predictions

These forecast results are pushed to Firebase and can be visualized in the dashboard to give a clear picture of how traffic is expected to evolve throughout the next few hours.

### 4. Weather Data Integration

This part of the system checks the current weather and helps the model decide if pedestrians should get priority — for example, when it’s raining heavily or there’s a heatwave. If the weather is bad enough, the system marks it as severe, which affects the traffic light logic.

#### 4.1 Getting the Weather Data

The weather information is fetched from an external API using this function:

def fetch\_weather\_data():

try:

weather\_url = get\_weather\_url()

response = requests.get(weather\_url)

response.raise\_for\_status()

return response.json()

except Exception as e:

print("⚠️ Failed to fetch weather data:", e)

return None

If something goes wrong — like a bad response or a network error — it logs the error and just returns None, so the rest of the program can keep running safely.

#### 4.2 How We Detect “Severe” Weather

Once we have the data, we run it through the function below to decide if the weather is considered *severe*. This affects the control system and can shift the priority toward pedestrian crossings.

def is\_weather\_severe(data):

if not data:

return False

try:

current = data['current']

temp = current['temp\_c']

wind = current['wind\_kph']

gust = current['gust\_kph']

precip = current['precip\_mm']

humidity = current['humidity']

condition\_text = current['condition']['text'].lower()

uv\_index = current.get('uv', 0)

except KeyError as e:

print("❌ Missing field in weather data:", e)

return False

STATIC\_TEMP = temp

if temp < 15 or temp > 33:

return True

if wind > 30 or gust > 45:

return True

if precip > 5:

return True

if "thunder" in condition\_text or "storm" in condition\_text or "snow" in condition\_text:

return True

if uv\_index > 8:

return True

return False

What counts as severe?

* Very hot or cold temperatures (under 15°C or over 33°C)
* Strong winds or gusts
* Heavy rain (more than 5 mm)
* Dangerous weather keywords like *storm*, *snow*, or *thunder*
* High UV index (above 8)

If any of these are true, we return True — which signals the rest of the system to favor pedestrians until conditions improve.

### 5. Main Prediction Loop

This is the heart of the system. It runs continuously, pulling sensor data from Firebase, predicting which traffic lights should turn green, checking the weather, and forecasting what the next few hours might look like. Everything gets updated in Firebase every 15 seconds.

#### 5.1 Helper Functions

Before making any predictions, we need to clean up the inputs. For example, the ultrasonic sensors give distance values, but the model expects binary values — either a car is present or not. We handle that here:

def to\_binary(distance):

return 1 if distance < 55 else 0

Then we wrap everything into a function that sends the preprocessed input into the trained model and decodes the output:

def predict\_traffic\_decision\_with\_label(model, ultra1, ultra2, ultra3, ultra4, face, severe\_weather):

df\_input = pd.DataFrame([{

"ultra1": to\_binary(ultra1),

"ultra2": to\_binary(ultra2),

"ultra3": to\_binary(ultra3),

"ultra4": to\_binary(ultra4),

"face": face,

"severe weather": int(severe\_weather)

}])

print("\n📥 Model input (after preprocessing):")

print(df\_input.to\_string(index=False))

prediction = model.predict(df\_input)[0]

light\_map = {0: "red", 1: "green"}

label\_map = {0: "bad", 1: "good"}

output\_labels = [

"priority", "traffic\_light\_lane\_1", "traffic\_light\_lane\_2",

"traffic\_light\_lane\_3", "traffic\_light\_lane\_4",

"traffic\_light\_pedestrian", "label"

]

result = {}

for label, value in zip(output\_labels, prediction):

if label == "priority":

result[label] = "pedestrians" if value == 1 else "cars"

elif label == "label":

result[label] = label\_map.get(value, f"unknown ({value})")

else:

result[label] = light\_map.get(value, f"unknown ({value})")

return result

#### 5.2 The Live Loop

This is the part that runs continuously. It reloads every few seconds, updates predictions, and sends them to Firebase. It also prints everything to the console so you can monitor what’s going on.

model = joblib.load("traffic\_model.pkl")

while True:

try:

clear\_output(wait=True) # cleans the output

firebase\_data = load\_firebase\_data()

inputs = {}

# ✅ Add severe weather flag

inputs["severe weather"] = check\_severe\_weather()

if firebase\_data:

# Extract features from Firebase

extracted = extract\_inputs\_for\_prediction(firebase\_data)

if extracted:

inputs.update(extracted)

# Step 1: Predict current traffic control

result = predict\_traffic\_decision\_with\_label(

model,

ultra1=inputs["ultra1"],

ultra2=inputs["ultra2"],

ultra3=inputs["ultra3"],

ultra4=inputs["ultra4"],

face=inputs["face"],

severe\_weather=inputs["severe weather"]

)

print("\n🚦 Predicted Traffic Control Output:")

for key, value in result.items():

print(f"{key}: {value}")

# Step 2: Forecast next 5 hours for pedestrians & density

people\_predictions, traffic\_predictions = predict\_forecast\_separately(

people\_model, density\_model, label\_encoders, df, hours\_ahead=5

)

# Step 3: Format as Firebase-ready dicts

padestrian\_forecast = {

pred["time"]: pred["people\_count"] for pred in people\_predictions

}

density\_forecast = {

pred["time"]: pred["traffic\_density"] for pred in traffic\_predictions

}

# Step 4: Push all data to Firebase

write\_prediction\_to\_firebase(result, inputs, density\_forecast, padestrian\_forecast)

# Wait before next update

time.sleep(15)

except Exception as e:

print("❌ Error during loop execution:", e)

time.sleep(15) # still wait before next attempt

#### Notes

* The model is only loaded once at the beginning (traffic\_model.pkl).
* If any part of the loop fails (bad data, network error, etc.), it catches the error and retries after 15 seconds.
* The forecasts help give visibility into expected traffic and pedestrian flow — this could be useful for future adaptive scheduling.

## **Firebase Structure Overview**

The traffic management system relies on a real-time database (Firebase Realtime Database) that stores sensor readings, system forecasts, and visual detections, alongside decision outputs generated by the control model. The data structure is logically divided into two main branches:

1. Web\_Data:Contains forecast information and environmental inputs collected for display, analysis, and real-time behavioral adaptation of the system.

* Car\_Detection: Binary status (0 or 1) for each of the four ultrasonic sensors (ultra1 to ultra4) indicating vehicle presence in each lane.
* Ped\_Num:Number of pedestrians detected by the camera (reflects crosswalk status).
* Date:Current day and time.
* density\_forecast:Hourly traffic density forecast, e.g., "12:00": "low".
* padestrian\_forecast:Hourly pedestrian forecast (integer values).
* weather: Weather condition data
* severe\_weather: Indicates whether the weather is extreme or dangerous (true / false)
* temp: Current temperature in Celsius.

1. data:Represents the current state of the intersection, including traffic light statuses, processed inputs, and the decisions selected for execution

* priority : Indicates whether the system prioritized pedestrians or vehicles ("pedestrians" / "vehicles").
* traffic\_light\_lane\_X : The status of each traffic light (green / red).
* faces :Number of pedestrians detected directly (usually the same as Ped\_Num).
* ultraX : Actual distance readings from each ultrasonic sensor (in millimeters), as opposed to the binary state in Car\_Detection.