San Francisco Real-Time Traffic Prediction & Optimization Project

Project Mission

To build a cloud-based system that uses real traffic data from San Francisco to predict congestion and dynamically regulate traffic light timings at intersections, reducing delays and improving urban mobility through AI-driven decision-making.

Phase 1: Data Acquisition & Road Network Setup

Goal: Prepare a clean, queryable dataset and road graph for San Francisco using real-world data.

Steps:

1. **GCP Environment Setup**
    - Create GCP project
    - Enable APIs: BigQuery, Vertex AI, Cloud Run, Maps API, Cloud Storage
    - Create BigQuery dataset (e.g., `sf_traffic_data`)
    - Create service account with appropriate roles
2. **Download & Inspect Data**
    - Traffic Flow Segments (SF Open Data)
    - Signalized Intersections (SF Open Data)
    - Uber Movement SF (optional, for historical patterns)
3. **Upload to BigQuery**
    - `traffic_segments`
    - `signal_intersections`
    - `uber_travel_times` (optional)
4. **Build Road Network Graph**
    - Use NetworkX in Python
    - Nodes = intersections

    - Edges = road segments with attributes (speed, geometry, time windows)

5. **Notebook & Exploration**
    - Create Jupyter notebook for data exploration and graph creation

Deliverables: BigQuery tables, Python road graph object, EDA notebook

Phase 2: Train Congestion Prediction Model

Goal: Predict traffic congestion for segments using real historical data.

Steps:

1. **Preprocessing**
    - Aggregate speeds by segment and time window
    - Generate features: time, speed, length, previous readings
2. **Model Selection**
    - Choose between XGBoost, Random Forest (tabular) or LSTM (time series)
3. **Training & Evaluation**
    - Train and validate with train-test split (avoid time leakage)
    - Measure MAE, RMSE, etc.
4. **Model Export**
    - Save with `joblib` or `SavedModel`
    - Upload to GCS

Deliverables: Trained model + evaluation report

Phase 3: Define Traffic Light Timing Logic

Goal: Adjust signal timings based on predicted congestion.

Steps:

1. **Baseline Setup**

    - Fixed-cycle signal timings for control
2. **Adaptive Logic Design**
    - Extend green/red based on predicted congestion
    - Optionally implement queue-length or throughput-based rules
3. **RL Option (Advanced)**
    - Reinforcement learning agent (e.g., DQN) to control signals
4. **Simulation**
    - Simulate using prediction + logic loop
    - Track metrics: delay, queue length, throughput

Deliverables: Signal logic engine + performance metrics

Phase 4: Build & Deploy Backend (Cloud Run + Vertex AI)

Goal: Host prediction model and signal logic as APIs.

Steps:

1. **Model Deployment**
    - Deploy to Vertex AI or serve via Flask on Cloud Run
2. **API Development**
    - Endpoints: `/predict_congestion`, `/signal_state`, `/update_traffic_state`
3. **Cloud Setup**
    - Configure service account, logging, billing
    - Add Stackdriver monitoring

Deliverables: Deployed backend with public REST API

Phase 5: Build and Connect the Dashboard

Goal: Create a web interface to visualize live and predicted traffic data.

Steps:

1. **Choose Framework**
    - Streamlit (fast prototyping) or React + Mapbox
2. **Implement Features**
    - Map of SF
    - Predicted congestion (color-coded)
    - Signal light states at intersections
3. **Backend Integration**
    - Fetch predictions and logic outputs via REST API
4. **Live Controls (Optional)**
    - Allow users to simulate scenarios or switch modes

Deliverables: Web dashboard connected to backend

Phase 6: Documentation & Project Packaging

Goal: Make the project clear, reproducible, and shareable.

Steps:

1. **Technical Docs**
    - Describe architecture, data flow, ML model, cloud setup
    - Use diagrams, flowcharts, and inline comments
2. **Code Docs & README**
    - Add `README.md` with setup, usage, structure, and credits
    - Add docstrings and inline comments to code
3. **Experiment Logs**
    - Save performance comparisons of signal strategies
    - Include visualizations: delay charts, speed heatmaps

4. **Reproducibility Setup**
    - Add `requirements.txt`, `setup.sh`, Dockerfile, or notebooks
    - Add quickstart and teardown scripts for GCP
5. **Final Assets**
    - Upload UI screenshots, architecture diagrams, optional demo video
    - Include LICENSE and acknowledgment section

Deliverables: Full project documentation + packaging for deployment or handoff