Indiana University Southeast

Feasibility Report #4

TrafficLouisville Software Design Description

Capstone Team: **So Much For Subtlety**

John Brown

Mason Napper

Aaron O’Brien

11/12/2024

Table of Contents:

1.1 System Overview

Traffic Louisville is a multi-component, combined hardware/software system that delivers near real-time information about the density of vehicle traffic on Louisville highways through an online map interface. It achieves this through accessing CCTV images publicly available through TRIMARC, a project of KYTC. It then analyzing the images with machine learning models that perform object detection, and after calculating the relative density of traffic at a geographic location, sends this information to be displayed on a custom heat-map of Louisville hosted on a webpage. TrafficLouisville is not intended to be a commercial system or to be used in critical applications. It is designed as part of an academic project that illustrates the potential of utilizing machine learning for near-real time applications. ***The purpose of this document is to describe the software design of TrafficLouisville.***

1.2 Individual Contributions

The members of the development team So Much For Subtlety include John Brown, Mason Napper, and Aaron O'Brien. We are grateful for help received from our capstone professor, Dr. Ronald Finkbine and our sponsor, Chris Sexton. We would also like to thank the Kentucky Transportation Cabinet Office of Information Technology, both for providing a valuable public information resource and for their guidance in our project.

Team Member Responsibilities:

John Brown: Team Leader, responsible for developing the backend services required for the project. **For this particular report, responsible for documenting the overview, data and architecture design, part of the interface design, and the local processing section of the procedural design.**

Mason Napper: Responsible for developing the public facing interface of Traffic Louisville, also contributes hardware to the project. **For this particular report, responsible for documenting part of the interface design and the remote frontend piece of the procedural design.**

Aaron O'Brien – Responsible for developing the machine learning component of our project through training models and governing their output, also contributes hardware for the project. **For this particular report, responsible for documenting the inferencing, training, and dataset creation piece of the procedural design.**

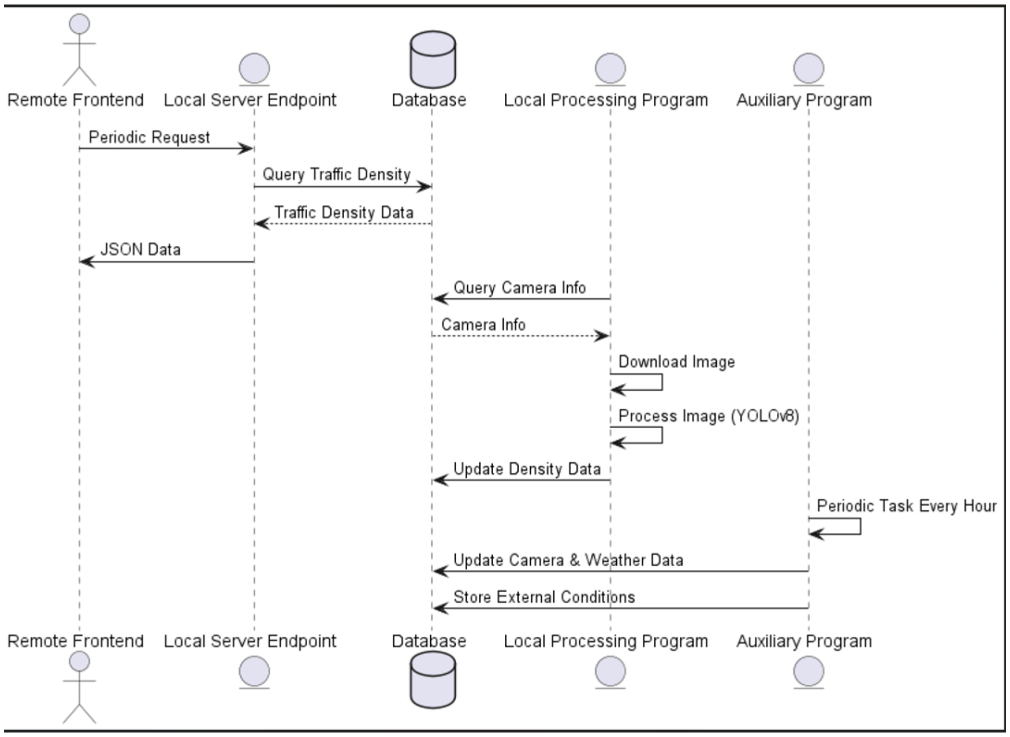
1.3 Data Design and Architecture Design

The data design for TrafficLouisville is structured around the SQLite database and a temporary image directory. The architecture design of TrafficLouisville is modular, with each module accomplishing a function that connects the data source to the database to the eventual output in the heatmap.

Images, camera metadata, and weather conditions are acquired externally via URLs. The auxiliary program updates camera metadata and weather conditions in the database, while the main program retrieves this information but can also update camera metadata for particular scenarios (error handling of expected errors).

To determine context for a particular camera, camera metadata is retrieved from the database. To acquire images for processing from external links (camera metadata), camera metadata retrieved from the database is required. In order to apply object detection on the retrieved images, weather conditions are retrieved from the database for a particular camera.

The results of image processing are written to the database by the main program, which may be retrieved by the server endpoint when a request is received.



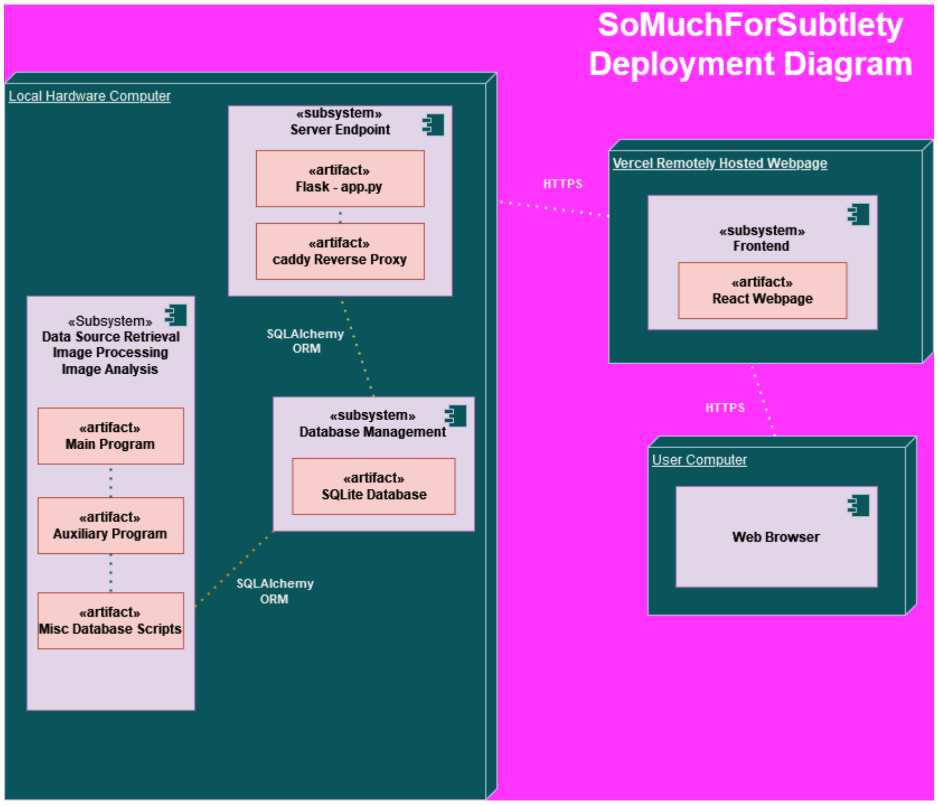
1.4 Interface Design

Within the local hardware used for Traffic Louisville, the main connection between the subcomponents is the database. Each has a connection to the SQLite database that allows for CRUD operations.

Between the local hardware and the remote frontend, the main connection is the local server endpoint. It allows the remote frontend to access the information it needs to populate the heatmap.

External interfaces for Traffic Louisville are the external data sources accessed by local hardware and the remote frontend hosted on Vercel. The human interface that users access will appear as:

- Large heatmap right in the middle of the webpage, only a single page on the whole site
- TrafficLouisville as a large header above the map
- KYTC message directly below the map
- Credits about the software team fairly small at the bottom of the page


SoMuchForSubtlety Deployment Diagram
Local Hardware Computer
«subsystem» Server Endpoint
«artifact» Flask - app.py
«artifact» caddy Reverse Proxy
«Subsystem» Data Source Retrieval Image Processing Image Analysis
«artifact» Main Program
«artifact» Auxiliary Program
«artifact» Misc Database Scripts
SQLAlchemy ORM
«subsystem» Database Management
«artifact» SQLite Database
SQLAlchemy ORM
HTTPS
Vercel Remotely Hosted Webpage
«subsystem» Frontend
«artifact» React Webpage
HTTPS
User Computer
Web Browser

1.5 Procedural Design

<u>Local Processing:</u>

<u>Main Program:</u> main_program.py contains an indefinitely running loop that handles image retrieval, initiates image processing and analysis, and updates the database. The duration of each iteration of the loop is ten seconds.

Input: Pre-loop, the program establishes a path to the database that can be connected to repeatedly. In the loop, the program retrieves the oldest camera entry from the database, acquires camera images from the URL in the camera entry.

Output: In the loop, updates the CurrentCamera table with timestamps (always) and camera statuses (conditionally upon error). Inserts entries in the TrafficCount table with outputs from the model.

Process:

1. Records current time as "start time".
2. Establishes fresh session with the database and queries for the oldest camera (the last updated camera).
3. Retrieves image via oldest camera URL (metadata), handles errors that may arise
4. Calls the appropriate Yolov8 model dependent on the metadata, receives traffic count
5. Retrieves last updated traffic count for that particular camera. Inserts traffic count into the TrafficCount table, updating the maximum traffic values if the current traffic count exceeds the historical maximum.
6. Updates CurrentCamera table with a timestamp, commits session transactions with database, closes the session
7. Waits until ten seconds have elapsed

Constraints:

1. No iteration of the loop can happen in less than ten seconds due to external request limit requested by KYTC. This is mitigated by the timer logic in the loop.
2. The main loop cannot constantly keep a session that is writing to the database open. This is mitigated by each loop iteration generating a new session and closing it within the iteration to ensure database writes are completed.
3. Image retrieval may not occur due to a lack of availability of external sources. This is mitigated by error handling within each iteration.

Auxiliary Program: aux_program.py contains an indefinitely running loop that handles weather condition and camera metadata retrieval and updates the database with this information, while also periodically performing database backup. The duration of each is loop is half an hour.

Input: Pre-loop, the program establishes a path to the database that can be connected to repeatedly. In the loop, the program retrieves rain probability from an external URL, retrieves camera status from an external URL, and queries the TrafficCount table for its length.

Output: In the loop, updates the CurrentCamera table with camera status and weather conditions, updates OfficialCameraList with camera metadata, and may initiate a backup of the TrafficCount table depending on the length retrieved.

Process:

1. Establishes fresh session with the database.
2. Retrieves data from ArcGIS server
3. Retrieves data from weather API
4. Retrieves length of TrafficCount table in the database
5. Updates the OfficialCameraList table with camera metadata.
6. Updates the CurrentCamera table with weather conditions.
7. Conditionally backs up the TrafficCount table.
8. Commits session transactions with database, closes the session
9. Waits until half an hour has elapsed.

Constraints:

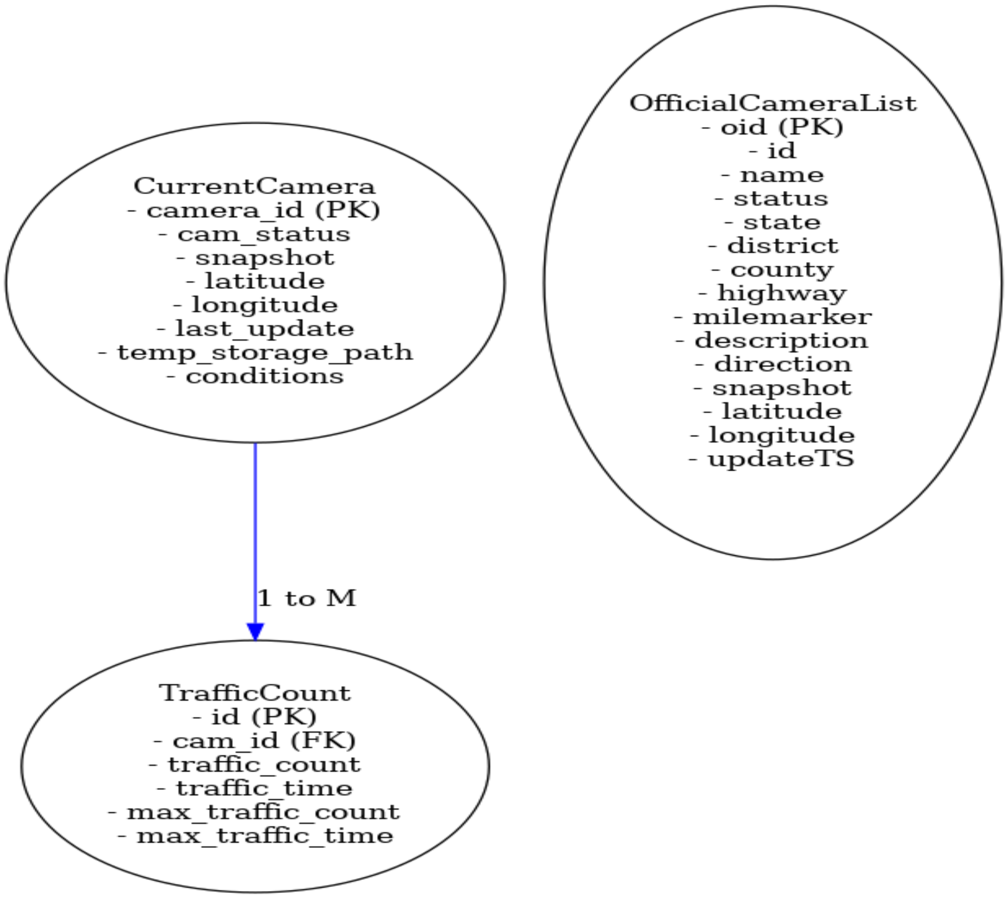
1. No loop should be completed in less than half an hour. This is not a hard limit, but is intended to reduce overlapping writes to the database and to not request too often from the ArcGIS/weather API resource. This is mitigated with timer logic.
2. External resources may fail to supply a resource. This is mitigated through error handling, and through an alert if the failure persists.

External Libraries:

- Python-dotenv : load directory as environment variable
- Requests (Python)- allows for crafting get requests to acquire external resources
- Urllib (Python) – allows for downloading image from external resources
- Time (python) – allows for keeping track of time using system time
- Sqlalchemy (Python) – allows for Python-Sqlite ORM access to database

Local Database:

TrafficLouisville uses an SQLite database. The first table describes the current rotation of cameras, with a camera id, status, coordinates, time of last update, a temporary storage path for a downloaded picture, and current conditions. The second table describes an hourly update of the official camera data from KYTC ArcGIS server. The third table consists of records of traffic counts for a particular camera id, as well as the max traffic count for that camera so density can be calculated. The temporary images directory contains a single image for each camera, the last one that was accessed. Each time a new picture for a camera is downloaded, it replaces the old one. There is no long-term storage of pictures.



External Libraries: sqlalchemy – used to create sqlite database in directory

<u>Local Server Endpoint:</u> app.py provides an endpoint for retrieving the most recent traffic data and camera coordinates from the database.

Input: Outside the endpoint, the program establishes a path to the database that can be connected to repeatedly. Inside the endpoint, accepts a GET request at /latest-traffic.

Output: Returns id, density, latitude, and longitude for the most recently updated traffic data.

Process: (when a request is made)

1. Starts a new database session
2. Queries the TrafficCount table for the most recent record based on traffic_time in descending order
3. Queries the OfficialCameraList table using cam_id from TrafficCount
4. Calculates traffic density as traffic count / max_traffic_count, sets density to 0 to avoid dividing by zero
5. Constructs a JSON response and sends it
6. Commits and closes the session

Constraints:

1. The traffic data might not exist. This is mitigated by error handling, sending a 404 if there is an error.
2. The max_traffic_count could be zero. This is mitigated by checking if max_traffic_count is zero and skipping the division.

External Libraries:

- Flask/flask_restful: allows for using a flask server as an API endpoint
- Sqlalchemy (Python): allows for Python-Sqlite ORM access to database

Local Image Processing:

Main Components

Inferencing: Applies preprocessing to an image, passes it to a trained model, and returns the number of detections.

Training: A pretrained model is trained on an input dataset, weights are returned for future inferences.

Dataset Creation: Input images are manually annotated and returned as a properly formatted and preprocessed dataset.

External Libraries

Ultralytics/YOLO: Main library for training and deploying models.

Roboflow: Library used for creating and importing datasets to train on.

Torch: Allows for the use of GPU instead of CPU when training models.

Inferencing

Purpose of this component: To use a trained model to analyze images and detect vehicles, returning the count of vehicles detected.

Input: Raw images.

Output: Number of detections.

Process to convert input to output:

a. Preprocess the image (resize, orient, grayscale, blur).
b. Pass the image through the model to get detections.
c. Interpret detections and return the number of vehicles.

Performance requirements: Not demanding of powerful hardware as detections are not real-time. CPU can be utilized for inferences, GPU not required.

Training

Purpose of this component: To fine-tune or train a model using a specific dataset, adjusting model weights for future inferences.

Input: Pretrained model, annotated and preprocessed dataset, epochs.

Output: Trained model weights and metrics.

Process to convert input to output:

a. Load the pretrained model and dataset.
b. Set up training loop with specified parameters.

c. Weights are adjusted based on dataset annotations.
d. Final and best model weights are saved for future inferences.

Performance requirements: Powerful hardware is necessary as a CPU will not be sufficient. A stronger GPU will allow for the use of more capable pretrained models with more available parameters. This allows for higher accuracy during inferences.

Dataset Creation

Purpose of this component: To create and format a dataset for model training by annotating images, ensuring they are properly preprocessed.

Input: Raw images

Output: Formatted and preprocessed dataset with annotated images.

Process to convert input to output:

a. Load raw images into Roboflow workspace.
b. Manually annotate images.
c. Preprocess images and format the dataset properly.

Performance requirements: Time intensive but not computationally demanding as Roboflow runs in browser.

Remote Frontend:

Main Components

App: Root component of the application

Map: Handles map rendering and live data display as a heatmap

DataFetcher: Manages the retrieval of data periodically from the backend API

UI: Manages user interface elements like messages, text, credits, etc.

External Libraries

React: Main framework for the component structure

Leaflet: Library for rendering the interactive map and heatmap overlay

Axios: Library for HTTP requests to the backend API

Major Methods/Routines

App Component – App.js

a. **Render() –** Initialize the main components, such as Map and UI, and serves as the application's root component

Map Component – Map.js

b. **initializeMap() –** Sets up the Leaflet map centered on Louisville, initializing map settings and zoom level

c. **addHeatmapLayer() –** Creates and adds a heatmap layer o the map to visualize traffic density

d. **updateHeatmap(data) –** Updates the heatmap layer with new traffic density data received from the backend API

e. **convertDatatoHeatmapFormat(data) –** Formats data from the API to a format compatible with the heatmap overlay

DataFetcher Component – DataFetcher.js

f. **fetchTrafficData() –** Uses Axios to send a GET request to the backend API for traffic density information

g. **handleFetchSuccess(response) –** Processes the API response, extracting and preparing traffic data for the map update

h. **handleFetchError(error) –** Handles any errors during data fetching and logs them for debugging

i. **startPolling(interval) –** Initiates a periodic fetch of traffic data at a specified interve

j. **stopPolling() –** Stops the periodic data fetching to reduce unnecessary load when not needed

UI Component – UI.js

k. **renderHeader() –** Displays the project title "TrafficLouisville" and other such information

l. **renderCredits() –** Shows information about the team like names and role descriptions

m. **renderMessage() –** Displays a message as requested by the KYTC with a link to their website as well

Utility Functions – Utils.js

n. **formatTimestamp(timestamp) –** Converts the timestamp from the backend API into a human-readable format for display

o. **calculateHeatIntensity(count) –** Maps traffic count data to heatmap intensity levels to standardize the visual representation