Indiana University Southeast

Feasibility Report #2

TrafficLouisville Software Requirements Specification

Capstone Team: **So Much For Subtlety**

John Brown

Mason Napper

Aaron O'Brien

10/8/2024

Table of Contents:

1.1 Purpose

The purpose of this document is to define the requirements for our software system, TrafficLouisville.

A short description of team members and their roles are given, along with definitions and references used later in the document.

For context, a general overview of our system's components and potential use cases are explored, and then we give further examination of the functional, interfacing, and performance requirements for the components of TrafficLouisville.

Finally, we define our expectations of our software system in terms of reliability, availability, security, maintainability, and portability.

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.

Mason Napper: Responsible for developing the public facing interface of Traffic Louisville, also contributes hardware to the project.

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.

1.3 Definitions

YOLOv8: A real-time object detection model used for identifying and classifying objects in images. It is employed in this system for detecting vehicles in traffic camera images.

Flask: A Python-based web framework used to create the backend API, which handles data requests and interacts with the local database.

Vercel: A cloud platform optimized for frontend hosting, allowing quick deployment and high availability for React applications.

ArcGIS REST API: A web API provided by ArcGIS for accessing geospatial data and services, such as downloading images and retrieving metadata from traffic cameras.

SQLite: A lightweight, serverless database engine used for managing traffic data and camera status locally.

1.4 References

Our Github Repository: https://github.com/browjor/FA24_Capstone_SoMuchForSubtlety

Flask Server Documentation: https://flask.palletsprojects.com/en/3.0.x/

Python Documentation: https://docs.python.org/release/3.12.0/

SQLite Documentation: https://www.sqlite.org/docs.html

Vercel Documentation: https://vercel.com/docs

React Documentation: https://react.dev/reference/react

Leaflet Library Documentation: https://leafletjs.com/

Yolov8 Documentation: https://docs.ultralytics.com/

Pytorch Documentation: https://pytorch.org/docs/stable/notes/cuda.html

Roboflow Documentation: https://docs.roboflow.com/

2.1 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.

2.1.1 System Functions

The system is composed of the following functions:

- **Data Source Retrieval** - Downloading an image, retrieving camera information from ArcGIS REST API
- **Image Processing** - Applying object detection to the image, changing
- **Image Analysis** – Converting object detection results into traffic density data
- **Database** – Inserting and retrieving information about results and camera status, also maintains other information associated with conditions
- **Server Endpoint** – Respond to GET request by retrieving information from database
- **Frontend** – Sending request to endpoint, receiving it and displaying it at the correct position on the heat map

The system will accomplish the preceding system functions through the following hardware/software components:

Running on local, privately owned hardware:

- **Local Processing** – The data source retrieval, image processing, and image analysis functions will be accomplished by two programs run on local hardware: the main program and the auxiliary program. The main and auxiliary programs will be written and run as Python programs. The image processing will be accomplished with Yolov8, a machine learning model.
- **Local Database** – A database will be maintained on local hardware that fulfills the database functions of the system. The database will be an SQLite database.

- **Local Server Endpoint** – A backend API server run as a separate program on local hardware fulfills the server endpoint function. The backend API server will be a Python- Flask server.

Running remotely on a cloud hosting platform:

- **Remote Frontend** – An application running on a cloud platform that hosts a public web page will accomplish the frontend functions. The application will be a React application hosted on Vercel, a cloud-based hosting platform.

2.1.2 Use Cases

As a complete software system, TrafficLouisville can be utilized in various ways.

- **Near Real-Time Traffic Monitoring and Visualization:** Without any modification, TrafficLouisville can be used to visualize the density of traffic by visiting a web page. While it may have some delay, it could be used by the general public to estimate traffic.
- **Historical Traffic Data Analysis:** Given a program not within the scope of this project, the data collected by TrafficLouisville over time could be utilized by researchers, urban planners, or government agencies.

3.1.1 Functional Requirements

The specific functional requirements for TrafficLouisville are as follows:

Local Processing Requirements:

- The main program shall run continuously, iterating in a loop every ten seconds as described below:
    - The main program shall start an internal timer of ten seconds.
    - The main program shall query the first database table for the Camera ID that has the longest time since last being read.
    - The main program shall query the first database table for the status and URL location of that Camera ID.
    - The main program shall contain logic to determine the availability of that camera and if unavailable, should update the first database table with the update time and continue to the next Camera ID.
    - The main program shall query the second database and determine the external conditions associated with the camera ID and use logic to select an appropriate ML model to be used.
    - The main program shall download the image from the URL.
    - The main program shall store the image in temporary storage, and replacing the image associated with that Camera ID that is currently in storage.
    - The main program shall process the image with an appropriate ML model.
    - The main program shall update the third database table with the results received.
    - The main program shall wait until the internal timer of ten seconds has elapsed before continuing onto the next iteration the loop.
    - The main program shall implement error handling that corresponds to database connection errors, various errors downloading images from the URLs, and runtime errors that happen during image processing.
- The auxiliary program shall run continuously, looping every hour as described below:
    - The auxiliary program shall send an HTTPS request to the public KYTC ArcGIS REST API with a specific query string that returns JSON information for all cameras.
    - The auxiliary program shall parse the camera information and update the second database table.

    - The auxiliary program shall send an HTTPS request to a weather resource and reads in the information, updating the second table as necessary.
    - The auxiliary program shall implement error handling that corresponds to database connection errors and errors resulting from failure to gather information from external sources.

Local Database:

- The database shall maintain three tables:
    - The first table consisting of attributes: the Camera ID, the status of the camera, the snapshot URL, the latitude, the longitude (all preceding attributes gathered externally), the time of last update (generated internally), the string file path of temporary storage, and external conditions.
    - The second table consisting of all relevant information gathered from the KYTC ArcGIS REST API traffic camera layer and external conditions
    - The third table consisting of attributes: the Camera ID, traffic count results from the last update, and maximum historical traffic results, time of update.
- The database shall allow read and update queries from the main program, auxiliary program, and read queries from the server endpoint.
- The database shall periodically output historical results from the third table to a dedicated permanent storage.
- The database shall allow concurrent read operations and serialized write operations.

Local Server Endpoint:

- The server endpoint shall run continuously.
- The server endpoint shall contain a single “get” endpoint (from REST architecture).
- The server endpoint shall receive requests routed through a reverse proxy.
- The server endpoint shall query the first database table for the last updated camera.
- The server endpoint shall format responses (density, latitude, longitude) in JSON format.
- The server endpoint shall send responses through the reverse proxy.

Remote Frontend:

- The remote frontend shall display the project title, Traffic Louisville, and a short description of the project and it's source (the team members).
- The remote frontend shall display a static image of Louisville roads at a scale where the highways are prominently featured.
- The remote frontend shall send requests to the server endpoint (routed at the reverse proxy) at periodic intervals.
- The remote frontend shall receive responses from the server endpoint routed through the reverse proxy and extract the data in a JSON format.
- The remote frontend shall update the heatmap according to the data received.
- The remote frontend shall display the message requested by KCTC and a redirect to the site as directed.

3.1.1 Functional Constraints

Local Processing Constraints:

- The main program must not request an image from the TRIMARC Snapshot URLs at an interval less than ten seconds.
  Mitigation: The timer in the main program is a safeguard to prevent this.
  Mitigation: Models for various conditions will be trained and through training on image sets solely from traffic cameras, we can ensure better results than through training on other pictures.

Local Database Constraints:

- The database must not store images.
  Mitigation: This will be hard coded.
- The database must not be located remotely (not being implemented).
- SQLite is designed for small datasets.

Local Server Endpoint Constraints:

- The local server endpoint must not receive excessive requests from frontend.
  Mitigation: The limitations of the image source and processing mean that the frontend can send requests at ten second intervals, more than enough for the server endpoint to handle.

Remote Frontend Constraints:

- Vercel may impose API request limits and serverless function quotas.
  Mitigation: By handling most of the processing in the backend, the host has to do comparatively little. The React script will only need to handle sending requests and displaying data.

3.2 Interface Requirements

Local Processing:

- The main program shall have access to web resources and have its own database connection.
- The main program shall request resources from URLs that belong to safe, established resources.
- The main program shall communicate with the database through read and update queries.
- The main program shall have access to memory storage on local hardware.
- The auxiliary program shall have access to web resources and have its own database connection.
- The auxiliary program shall request resources from URLs and API endpoints that belong to safe, established resources.
- The auxiliary program shall communicate with the database through create, read, update, and delete queries.

Local Database:

- The database shall have established connections with the main program, the auxiliary program, and the server endpoint.
- The database shall have access to memory storage on local hardware.
- The database shall allow read and update queries from the main program, all CRUD queries from the auxiliary program, and read queries from the server endpoint.

Local Server Endpoint:

- The server endpoint shall operate on a network port of the local hardware, listening for requests routed through the reverse proxy.
- The server endpoint shall have an established connection with the database.
- The server endpoint shall perform read queries on the database.

Remote Frontend:

- The remote frontend shall send GET HTTPS requests to the proxy address, then routed to the local server endpoint.
- The remote frontend shall receive a response from the proxy in JSON format.
- The remote frontend shall provide a webpage that is available at a web address with no authentication.

3.2.1 Interface Constraints

Local Processing:

- The main program and auxiliary program may be unsuccessful in downloading an image, processing an image, or querying/requesting information from the database.
  Mitigation: Error handling will be implemented to handle these cases.
- The main program and auxiliary program must not write concurrently to the database unless the database is upgraded in a future update.
  Mitigation: The main program and auxiliary program are on different database connections, meaning that writes are performed serially, not simultaneously. This is something that SQLite confirms.

Local Database:

- SQLite is not designed for heavy writes and rewrites, or simultaneous concurrent writes.
  Mitigation: The main program and auxiliary program are on different database connections, meaning that writes are performed serially, not simultaneously. This is something that SQLite confirms.

Local Server Endpoint:

- The local server endpoint must not receive excessive requests from frontend.
  Mitigation: The limitations of the image source and processing mean that the frontend can send requests at ten second intervals, more than enough for the server endpoint to handle.

3.3 Performance Requirements

TrafficLouisville is not designed to perform commercial or critical functions, but the general performance requirements for components are that they are to remain continuously operational with minimal downtime. The bottleneck of our system is our image capture rate and processing rate, and external weather conditions that affect our image processing capabilities. This is a small-scale system that will not overtax the local database, local server endpoint, or remote frontend. Database queries are being made once every five minutes at the most frequently, which is well within our system capabilities.

3.3.1 Performance Constraints

Local Processing:

- Yolov8 object detection is intensive, and processing may take more than the ten second interval.
  Mitigation: This will be handled with logic in the main program loop, but should not be a problem, the loop will simply continue with the last updated camera.
  Mitigation: Yolov8 will be using GPU-acceleration to speed up processing on images.
- There are over two hundred cameras available on Louisville Highways. At our maximum download rate of one picture from a camera per ten seconds, and assuming the entire local processing tasks take ten seconds, this means that the

longest time for an update for a camera is 33.3 minutes. (200 cameras * 10 seconds / 60 seconds in a minute).
Mitigation: We will be narrowing the list of cameras to essential spots, choosing either at least fifty or at most one hundred cameras. The cameras on Louisville highways are not dispersed evenly, many are located in Spaghetti Junction where many highways intersect. This lowers the time for update for a camera to 16.6 minutes.

- The CCTV cameras used on Louisville highways often contain images of both lanes of traffic. If one has heavier traffic, this may bias the object detection capabilities of our machine learning.
  Mitigation: Due to the nature of this project and the CCTV cameras, solving this issue remains out of our scope. The CCTV cameras are movable by their operators, zooming in the frame of view and sometimes switching direction. There is no way for our project to take this into account.
- Environmental conditions on Louisville highways may change the outcome of the image processing capabilities.
  Mitigation: By utilizing different models of Yolov8 for different environmental conditions such as day, night, rain, and snow, we can attempt to overcome this.

3.4 System Attributes

3.4.1 Reliability

As our system is not intended to perform commercial or critical functions, it is designed to operate continuously. With error handling, we can keep the local main and local auxiliary program running and updating the database.

3.4.2 Availability

Our system is expected to be operational 95% of the time, and display when it is not available or being updated on the remote front-end.

3.4.3 Security

With a reverse proxy and HTTPS, the local sever endpoint is protected. Vercel handles security for the frontend and no authentication from users is needed, as the webpage is simply a display. The SQLite database is only accessed locally.

3.4.4 Maintainability

Our system is highly modular, with separate components that handle the various functions. Each component can be independently updated, so it is flexible.

3.4.5 Portability

As the remote frontend is hosted on a cloud-platform, it can be operated continuously on any other appropriate cloud platform. The local components can be run on any hardware compatible with Python, provided that the hardware has access to a GPU.