Vehicle Detection, Tracking, and Status Classification in Parking Lots

Final Report

2N1

Jackie Mioduski

Fabianna Barbarino

Spencer Cho

Aniruddha Srinivasan

Coleman Todd

Tyler Roosth

Department of Computer Science

Texas A&M University

04/24/23

Table of Contents

1 Executive summary

This project was assigned to us by a large physical retail organization with stores all across the United States. Many vehicles park there daily, and the parking lots are monitored by security cameras. Any incidents are caught using manual reviews, security guards. Parking layout improvements, analysis and design are all based on manual reviews subject to human bias and perception. This process can be time and money consuming, and it can be difficult for both security guards and business analysts to get an understanding of incidents and patterns.

However, our customer believes that new technologies can make it radically easier for both security guards and business analysts to get a good understanding of their parking space and how exactly it's utilized. Instead of human review, they wanted us to train machine learning models to detect parked cars, stopped cars, and moving cars. The footage and associated data will then be sent to security guards, giving them the tools to quickly act. Business analysts will have a separate view where they can analyze historical data. For scalability and cost purposes, our solution uses their existing setup of cameras with no additional sensors. Since infrastructure capabilities could vary by store, our application must be containerized and thus be flexibly able to run either in various clouds, or an on prem server.

Given time and experience constraints, we established concrete requirements while keeping in mind the vision outlined in the previous paragraph.

In order to design a 'full stack' solution, we trained an ML model based on YOLOv8 and ByteTrack capable of quickly and accurately identifying cars and car status. We established a flexible storage and database schema that logs relevant information about the vehicle and parking space occupancy based on MongoDB Atlas. We built a Web UI using React and Fastify with two separate views, one for the security guard that is optimized for immediately actionable items, and another view for the business analyst that allows them to view historical data. Finally, we have run this application inside two separate Docker containers. This solution works on pre recorded footage.

Our expected result at the end of this project was to be able to run this application with a single feed analyzing 5 cars at a time at 10fps as a minimum viable product. Our machine learning model exceeded this benchmark, with us being able to analyze 37 cars at a time with 11.77 fps. Our GUI is intuitive, with a table of information and associated video feed and allows security guards to click on a timestamp in a table to jump to it, along with options to filter and sort the data. We used MongoDB Atlas, which provides cloud storage and persistence along with an easy to use UI. All our components successfully work together, and our application is containerized

Overall, we successfully implemented good baselines for each component for our solution. This leaves room for future teams to focus on a single aspect of the application. For example, a future team could focus on just the User Interface, making it more accessible and relevant to the needs of security guards while still having our Machine Learning and Database solution. Another team could work on adding more states, or taking the application live. Therefore, we believe we've built a good platform for future development.

We expect that if further developed by our customer, they will see a huge increase in actionable intelligence and capability that will allow security guards and parking officials to have

faster response times and less manual review, potentially clearing up obstacles that allow stores to run smoother. Business analysts will have a wealth of data - potentially across multiple stores that will allow them to find patterns in parking space utilization and come up with ways to optimize the stores. Although there may be a cost for the infrastructure in order to run the application, the data that the application will provide will be a huge improvement to what the customer had previously.

Despite time and experience constraints with regards to machine learning, in the end our team was able to achieve our objectives. 6 people were divided into teams of 2, with a UI, Backend, and Machine Learning team. Team collaboration was good, with members cordial and willing to help one another out. Our program director provided employees to consult and give feedback and suggestions on architecture. Lessons learned include working on containerizing the application earlier, and having more cross functional collaboration.

In conclusion, our team was happy to come up with a successful solution we hope will eventually allow our customers to be more efficient and improve their services.

2 Project background

Parking lots are a source of frustration for many people. They tend to be the site of many crashes and illegal parking. As businesses grow their customer base, the parking lots around their locations will be subject to more traffic (both vehicle and pedestrian), this results in both security and analyst personnel having to process more data with the same amount of time. Security personnel must be able to identify the status of a large number of cars while Analysts must track and map all customer behavior. This becomes increasingly difficult without some sort of tool to monitor these occurrences. We can use machine learning to detect and track vehicles across a parking lot and continually log the driving patterns. Through the use of existing neural network models like YOLOv8 and modern web application frameworks like REACT we can create a monitoring app that is designed to solve this problem.

2.1 Needs statement

According to the National Safety Council (NSC), “tens of thousands of crashes occur in parking lots and garage structures annually, resulting in hundreds of deaths and thousands of injuries” [1]. While not all of these can be prevented, the ability to monitor and recognize traffic patterns in parking lots can greatly help avoid these kinds of accidents from occurring. It can also help our client ensure the safety of their parking lots and decrease any liability issues that could arise without the monitoring. Moreover, the security personnel watching over the parking lot could have a quicker reaction time if they are monitoring when something occurs. On top of this, traffic has been an

ongoing aggravation for many drivers and stores. It keeps people from getting to the store smoothly and having a pleasant shopping experience. Monitoring the traffic patterns in a parking lot can allow us to find ways to improve the flow of traffic and ensure the customers are satisfied.

2.2 Goal and objectives

Our goal is to build a containerized app that detects and tracks vehicles across a parking lot and continually logs the driving patterns. We want to have all of the parts: front end, back end, and database as separate Docker images that can run simultaneously. This can all be monitored using our user interface which allows two different types of users to view it: security personnel, and data analysts. The security personnel use the user interface as a way to monitor parking lot activity and ensure the safety of all the customers in real time. The data scientist uses the user interface to mine the database and analyze patterns like where the traffic jams are occurring and why. Our first objective is to ensure our vehicle detector can track at least 5 cars. The cameras will be looking out at around 12 feet off the ground, and we must ensure they can successfully track at least 5 cars at this height. Our second objective is to have bounding boxes around the vehicles whose coordinates will be sent to the database. Our third objective is to tie the backend with the front end successfully for our user interface. We have actually been able to complete all of these objectives successfully. We originally wanted our user interface to work in real time with a live video feed, but we have realized that with the time constraints it is best if we focus on using pre recorded mp4 videos. This will allow us to simulate a “live” feed while focusing more on the key parts of the project like labeling the states of multiple cars, table filtering, and jumping to specific parts of the videos. Our fourth objective is to ensure our implementation works as a desktop app on a GPU workstation or laptop.

2.3 Design constraints and feasibility

One of our technical constraints is the minimal testing videos that we were provided. We have been using these to train our model and it works great, but it will most likely have to be further trained for parking lots which are structured very differently. Our economical constraint is that we have a limited budget per person on our team for all of the expenses we will have for the project. We have not spent any money so far and are not expecting to, so we do not see this becoming a problem for our completion. Last but not least, our temporal constraint is that we have until April to work on the project. This has been our most impactful constraint so far and we have had to adjust accordingly.

First, we decided to use pre-trained software instead of training our own because we realized it would take up too much of our time. Secondly we also decided to use the pre recorded videos instead of trying to use a live feed for the security view. This will allow us to focus on the more important parts of the project with the remaining time we have.

2.4 Literature and technical survey

There are several commercial products and research projects already in place to solve the issue of parking lot monitoring and optimization for businesses. A few are detailed below.

- **M-Gage Node Pucks**:

  M-Gage Node pucks are sensors used to detect a vehicle in the vicinity via fluctuations in Earth's magnetic fields caused by any "large ferrous objects." They can be placed under the ground in parking spaces to provide a system to direct vehicles to available parking. This is a costly solution to reducing parking congestion and customer frustration which requires drilling a small hole in the concrete of each parking spot. These sensors would run upwards of $436 per parking space, not to mention costs associated with drilling into concrete: 3-inch core equipment, sealing material, etc. [2].

  This solution also provides limited data about traffic flow/safety/tracking within the parking lot unless you install multiple sensors in the lanes as well; and does not provide data on the individual status of vehicles so much as the presence of vehicles in specific locations. Therefore this is an overall poor solution as it is not cost effective, easily implemented, or scalable and it has limited applications to our problem needs.

- **Parksol**: Parksol is a fully managed product that offers integrated analytics that give more information as to parking habits, APIs, and sells both the sensor hardware and the software. It's designed for our customer, but perhaps with not as much scale. As a fully managed product, our customer will not have to spend development time to maintain and improve the product. Parksol also runs on-prem which means data is kept secure [3].

  Integrated analytics will help satisfy the business analyst's needs. However, we may have features in this product that we don't need and the product is mostly for smaller scale businesses and has not proven its ability to meet our customer's needs. Since our business has many stores country-wide, a custom solution will achieve meaningful cost savings.

- **ArcVision, A Deep Learning based Illegal Parking Detection Platform**:
  A 2019 Esri Intern Hackathon Project defines a camera-based solution to track illegal parking offenders while recording license plate numbers for vehicle identification. This solution uses OpenCV for video image extraction, Darknet YOLOv2 for vehicle extraction, Keras and Tensorflow for license plate detection, and ALPR (Automated License Plate Readers) for detecting license plate numbers. The user interface includes a Map View (for spatial data visualization and user-defined illegal zones for parking) and a Card View for video feed and parking classification statistics [4].

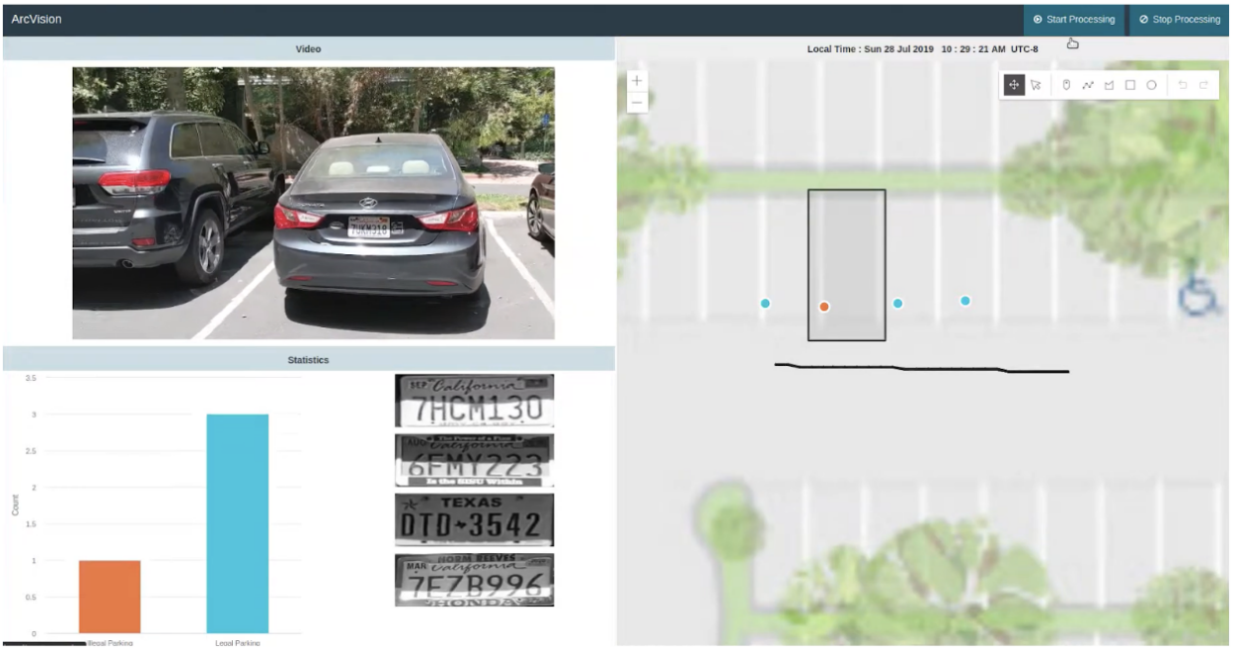


Figure 1: ArcVision UI
Source: Adapted from [5]

- **Smartpark System - Smart Parking**:
  Smart Parking provides many parking monitoring services and devices across the UK from vehicle sensors to mobile apps to Cloud interfaces for accessing and processing data in real time. The Smartpark system is fully integrated with parking guidance, payment, and analytics with the option for hiring human patrols to manage parking [6].

- **An Ultrasonic Sensor System for Vehicle Detection Application**:
  A team of researchers from the Department of Physics in the Bandung Institute of Technology developed an ultrasonic sensor system to count and classify passing vehicles by type (Sedan, SUV, or Truck) and speed. Their system used two ultrasonic sensors, an Arduino microcontroller with data collection capabilities, and a Java program. Their solution does not rely on video quality as a camera would and it is smaller and cheaper as well. Although unlike a camera, it only functions to monitor vehicles passing a certain point [7].

These designs relate to ours in that they attempt to solve similar issues with vehicle detection and parking lot management, albeit using different technology. Our proposed design is most similar to the ArcVision camera-based solution above and it should be cheaper than the sensor-based models, and more applicable to the needs of security as well as. Our solution will be better for tracking and classifying the status of multiple vehicles at once over a wide area and gathering data about traffic flow/congestion over time as well as logging illegal activities.

2.5 Evaluation of alternative solutions

Any solution that we come up with will have many components such as a detection system, a tracking system, a backend to store information, and a Graphical User Interface for both security guards and business analysts. The following alternative solutions will focus on a few different specific detection and tracking algorithms that we may choose from and their pros and cons.

Detection Algorithms:

YOLO (specifically YOLOv8) -

Pros: YOLO is the best open-source neural network model for object detection that is capable of drawing bounding boxes over vehicles in real time. It is designed for speed and low infrastructure use so it is perfect for optimizing cost. We will also easily be able to send data to any backend we decide to choose.

Cons: The algorithm is more sensitive to noise and not as accurate when estimating object area. It also struggles when detecting small things in images. The YOLO detection model is frequently updated. Retraining the model requires at least some level of software engineering expertise. The company for which we are building our solution has to manage their own infrastructure, maintenance, and updates.

Mask R-CNN -

Pros: Masking allows for a better approximation of area when compared to YOLO.

Cons: It is not as fast or accurate for detecting multiple objects in real time, so it is not the best option for our purposes.

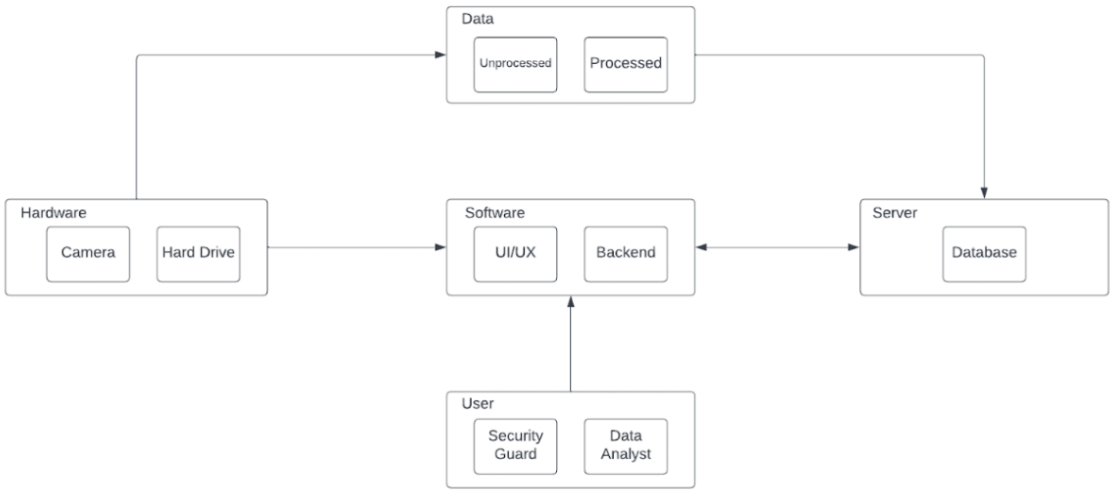
Conclusion -

Weighing the pros and cons of both of these options, we are leaning more toward choosing YOLO over Mask R-CNN because YOLO will better suit our needs for efficient real-time vehicle detection.

3 Final design

3.1 System description

The system description will consist of a high-level block diagram, and a functional description of the different parts and interfaces.



Hardware Module:

For the Hardware portion of our Block Diagram, this would consist of our physical media utilized for the project. For example, any cameras, hard drives, or computers, these would fall under this module. The parts of this module currently would include a camera and hard drive, and possibly more if we see the need for anything else throughout the development of the project. The camera would be used as our eyes, in order to gain information over a parking lot. How cars are parked, if

they are parked illegally, if they are moving, if there is a traffic jam, so on and so forth. This will be a major implementation we will do if given the time. Next, is our hard drive, this is fairly simple and will be a method of storing our data if needed, as well as using our application on other devices in order to test the functionality of how our application may run given different devices. We would also store any pre-made videos here in order to use our application. The information garnered from the camera or Hard Drive would be sent as Data to our Server and would be used as we see fit.

Data Module:

The Data module is very simple, but without it we would not be able to continue throughout our project whatsoever. To start off, the data gathered from the cameras of our Hardware portion of our Block Diagram is taken, purely as unprocessed, unfiltered data, and would then be sent to our Server in order to house this information until we decide to use it and determine what is occurring in the data itself, which would take place in our application via machine learning techniques. With this, while it may seem like not much, would be a major source in how we may be able to create our own model to use for our machine learning as we progress through the development process of our project.

User Module:

The User module is how a User will interact with our application. And how our entire project is made useful, is all dependent on the User. The User will mainly consist of two positions or viewpoints as of the beginning of our development, a Bodyguard view and an Analyst view. The idea is that the Bodyguard's role as a User is to use the information gathered from the Hardware and Software portion of the project in order to take note of any mishaps that may occur while they are on the job, which would most likely occur from the viewpoint of live video feeds. For example, if our application determines that a vehicle is illegally parked, the Bodyguard would in some way be notified so that they may be able to take the appropriate action required to remedy the situation. The same can be said if there was an accident, and so on. This notification while not currently set in stone can be a sound alert, a text message, an on screen display notification, etc… On the other hand for an Analyst the use of this application completely changes. Their goal will be to analyze the data from the Data and Server module, and take note of any unusual occurrences that our application may find, therefore their viewpoint would revolve more around the Data and less on video feeds.. For example, there may be an influx of cars causing a traffic jam at a certain spot, the Analyst would use our Data and determine why that may be occurring and try to solve it, or there may be a location prone to

numerous accidents, and again the Analyst would be in charge of trying to fix this issues in some way or form. The Data the Analyst uses would be gathered and organized in some way to make this process easier via algorithms we end up implementing.

Server Module:

The Server module consists of a database that would house not only the untrained data from the video feed, but the processed data used to train our application. The software would pull the untrained data from the server side, process it using machine learning algorithms, and push the processed data back into the database, mongoDB. Live video feed would be taken and transformed into data that would allow for further training of the model which is then stored in the server. The server would provide a consistent benchmark for the software by allowing us to view the training unfold over time.

Software Module:

The Software module consists of both a frontend and a backend. The backend will do much of the processing of data utilizing the programming language Python. Within Python, various machine learning modules will be used such as pytorch, sklearn, and pandas in order to train our application to properly identify vehicle positioning. This processed data will in turn be sent to the server for storage and there is hope that following training using recorded video, we will be able to transition to a live video feed for real time data processing. The software will primarily be focused on that of machine learning, a subset of artificial intelligence, that allows for the training of an application to classify and predict scenarios based on predetermined models. This involves using features and labels in order to fit data to models we deem appropriate and should the application be incorrect in its determination, we shall alter the program and the model to fit our specifications. We will also use Docker in order to containerize our application, making the testing, building, and deployment easier all the while allowing for version control. In addition to a backend, a frontend consisting of custom components created within React.js will also be created in order to make navigation and use of the application easier on users.

3.2 Complete module-wise specifications

Machine Learning Specifications:

The machine learning portion is made up of two components currently. We are running our current code in a Jupyter Notebook on Google Colab since the live requirement has been removed. The model is currently running on a 1GB VRAM GPU.

YOLOv8:

YOLOv8 is the latest version of the "You Only Look Once" detection model. Using this model, we can identify vehicles in the camera frame and assign them labels including car, motorcycle, and truck. We have set this model to only identify vehicles that it is at least 75% confident about. Anecdotally, we have found strong evidence that this is a suitable model for our needs

Bytetrack:

Bytetrack is the object tracking module that we attach to YOLOv8. It assigns persistent id throughout the video frame and allows us to track movement and thus assign states. This is how we can identify and record data in the format we specified in the Database Specification section. There is some 'connectivity' code between YOLO and Bytetrack since they don't work seamlessly together out of the box. We have written code to track the speed in pixels/frame in order to determine whether a vehicle is moving or not. We set the threshold value to 1.5 and found that this does a reasonably accurate job. This allows us to determine 'Stopped' and 'Moving'. There are some issues with Bytetrack, namely that the bounding boxes can get confused when an object is moving very close to another object. We added some logic in order to minimize this effect.

Performance:

Currently this model runs at around 17fps with more than 10 vehicles in the frame, well above our specifications. If we were to run it live, we would have it running on a server with a 'pubsub' method to send data to the database. In the future, we would like to increase accuracy for edge cases.

Database Specifications:

The Database is made from using MongoDB. Currently it is set up to where the following information is logged: Unique ID, X, Y, State, Timestamp, as well as Bounding Box Width, and Bounding Box Height. This information would be taken from the Machine Learning portion and entered into the Database. These would then be utilized by the Front End in order to give the Security

or Analyst page the information that is required. The information from the Database is hosted on MongoDB Atlas which is MongoDB's own cloud platform where information can be stored on the cloud. It's a nice alternative to some other choices with its ease of use and collaboration with MongoDB since they are developed hand-in-hand. Now the information can be queried fairly easily from any point as long as the appropriate access is given.

Back-End Specifications:

The back end is currently made up of Node JS as well as Python Node JS is used and connects with the Database and interacts with the frontend by allowing calls to be made to the database in order to retrieve the information needed to fill the tables that correspond to the submitted video. This is done via a Cloud connection to MongoDB Atlas, which would be where the information from the database would be stored as well as taken. On the other hand, we use Python when it comes to connecting the Machine Learning portion of the project with the Database. Since this is where the submitted video will be sent to in order to get that information. We then make calls on the backend in order to submit that information regarding the submitted video into the MongoDB database which will then in turn be used when showcasing the information related to the attached video in regards to the frontend within the tables.

Front-End Specifications:

The front end has been implemented using REACT, HTML5, and CSS. It is set up to interact with the node.js backend for api calls. These are used to get the data for the tables as well as the queries needed for the analyst view. There are two main pages/views for the front end: security guard view, and analyst view. The security guard sees a "live" view of the camera and a table to the right hand side with the statuses of the vehicles as time goes on. The video is currently a prerecorded mp4 video since we have determined the "live" feed to be a stretch goal. On the table they are able to see the car id, time stamp of the data, and the status. They are currently also able to filter the data by a specific status. For example, if I want to see all the parked cars throughout the day I would only check the parked checkbox. The analyst page is very similar with both a video and table with the same information. The analyst however, can inspect the details more closely. They have to choose from multiple videos first. From there, they are able to see the video and its associated data loaded on the table. They can also click on a timestamp and be redirected to that time in the video. We were going to implement some special queries for the analyst view as a stretch goal, but due to some time constraints we decided to focus on getting the most important parts done. Overall, both views are

very similar but the analyst has access to a few more tools. There is currently no login needed. Any user can access both views at the moment as well since it is not part of our scope to separate them.

3.3 Approach for design validation

Changelog from Project Proposal

- 'Parked' can be an extension of 'Stopped' state (eg: if car is stopped for more than 30 seconds, it is considered parked)
- GUI should be able to specify how long a car is 'Stopped' for it to be considered 'Parked'
- Annotated video does not have to be live - can be premade by ML script and uploaded to Web App

If we divide our project into Tracking, Database, GUI, Infrastructure - here are the tentative metrics for success.

Tracking:

Model should be able to track at least 5 vehicles at an FPS greater than 10. Should detect parked, stopped, not stopped cars. The accuracy should be reasonably good (85%). According to our project manager, this is a priority for the application. Currently the model achieves this, with a very high observed accuracy rate and around 60-70 fps using a GPU on Google Colab. For the 'Parked' state, instead of detecting whether a car is parked within the lines, we are cleared to use our 'Stopped' state instead to check if a car has been stopped for more than 30 seconds. If so, we will mark it as 'Parked'.

Database:

Log driving patterns with x,y,bbox width, bbox height, timestamp, and state. The database should also make sure that identifying information is not present. Temporary ID from Deepsort is ok and does not need to persist. Queries should be able to extract historical patterns for specific cameras.

GUI:

There should be two views: one for a security guard, and one for a business analyst. The security guard view should be tailored towards live, ondemand footage. Security guards would be able to view a live feed. They should also be able to easily see incidents sorted by time in a table to the side. Clicking on an event should bring up the video recording at the time. The business analyst view should focus more on historical data as well as patterns. At minimum they should see a table

where they can sort and filter by time/event. Both views should be able to customize the time it takes to determine

In terms of technology, a web app is preferred, but is a stretch goal. Currently we are using a React App

Infrastructure/Deployment:

Our app should be able to run in a Docker container. The intent of this is so that our project assigner can run our application on any infrastructure/cloud environment they deem necessary. Additionally, we should find minimum specs needed so that our application will run smoothly, and explore the relationship between increasing specs and the speed of our ML detection. It is likely that we will need infrastructure with a dedicated GPU for best performance in ML. Currently we are using a Google Colab GPU with 2 gigs of VRAM, and we can work further towards testing it on live infrastructure.

Testing:

Once we have established our application, we want to be able to test and see if it conforms to our project expectations. We will be using a single usb webcam to simulate a live feed, mounted at a similar location to how our project assigner will mount them, over a parking lot. Alternatively our project manager cleared us to use an mp4 file that is fed to our application as if it were live.

We will then use our application to view this live feed, and track the accuracy and speed (at least 10FPS for 5 cars, 85% accuracy of state tracking) . Security guards should be able to view any incidents by clicking on the timestamp of a particular incident and watching a replay. We will also use the Business Analyst view in order to track historical data, in this way testing the database. Our app should be running in a container.

Currently the ML model runs at 17 fps for 12 vehicles, well above our benchmarks. We also only track vehicles above a 75 percent confidence level.

After discussion with our project manager, the strictly necessary events to track are **stopped/parked/not stopped**. Speeding and illegally parked are out of MVP scope as they may require more advanced analysis.

Additionally our project manager has cleared us to use an annotated mp4 file that is premade our application. This would make the project testing less rigorous, as we can run our machine learning model to annotate the video separately from our web application.

If our application is able to successfully complete this test, we will consider that to be our MVP (minimum viable product). There are other potential considerations when looking at a customer that operates on the scale of ours. but we consider this a good benchmark for a proof of concept.

Demonstration:

Ideally the app will be demoed on Walmart's own infrastructure if we can get the container working with live video feeds. However due to the complexities, it's also acceptable if we pre-make the annotated video and upload it to our web app and run it on our personal computers.

Summary:

App must run in a container

App must detect stopped, not stopped, parked

App must have a database with x,y,timestamp,event,bbox width, bbox height

App must have two views

4 Implementation notes

Project Name

Vehicle Detection, Tracking, and Status Classification in Parking Lots

Type of Implementation

Type of System	Desktop GUI Application
Programming Languages	Python, JavaScript, HTML5, CSS, MongoDB
Data Storage	MongoDB database
UI Technologies	REACT, MaterialUI, HTML5, CSS, JavaScript

Engineer's Notes for Our System

Implementation of Machine Learning

Google Colab	We primarily used Google Colab to run our Jupyter notebook because it allowed us to test and debug our work with multiple users and provided access to cloud storage for our videos. It also provided a free NVIDIA T4 Tensor Core GPU which we used to calculate our benchmarks. Overall, we found it to be a useful platform and recommend it for future use.
Docker	Alternatively, our application can be run in a docker container on your own hardware. For optimal performance, we recommend using a GPU with at least 1GB VRAM.
YOLOv8	Our project is based on the YOLOv8 (You Only Look Once) object detection software. YOLOv8 is capable of detecting hundreds of object classes, however, for our use case, we focus on detecting cars, trucks, motorcycles, and buses. In our anecdotal experience, we found that the default model was effective in tracking these items, obviating the need to train our own model, although YOLOv8 does support such a function. It is important to note that YOLOv8 is updated periodically, and newer versions may become available that offer superior performance.
Bytetrack	Bytetrack is an object tracking software that utilizes detections from YOLOv8. Bytetrack assigns unique identifiers to each object detected by YOLOv8 and retains these identifiers as we progress through each frame of the video. This enables us to effectively track objects over time.
Boilerplate connectivity code between YOLOv8 and Bytetrack	As YOLOv8 and Bytetrack are distinct software projects, it was necessary to implement considerable boilerplate code to enable matching of detections generated by YOLOv8 with corresponding objects tracked by Bytetrack. It may be worth noting that subsequent updates to YOLOv8 have

	eliminated the need for this code since we have started the project. We used a helpful instructional video from YouTube to guide us in the development of the required code: https://www.youtube.com/watch?v=OS5qI9YBkfk
Roboflow	Roboflow is an online platform that provides an open-source library designed to assist with various computer vision tasks. The platform also provides a selection of useful datasets, free of charge, as well as comprehensive instructions on how to combine YOLO and Bytetrack among other features. Using this library, we were able to efficiently organize our code and detections. Although usage of this library is not necessary, it can be helpful in order to circumvent the necessity to write custom code for all aspects of this implementation.
Implementation Steps	● Utilize the pre-trained YOLOv8 model for object detection in video frames. The YOLOv8 model processes the video frames and generates bounding boxes around detected vehicles. Use only the classes for vehicles. Corresponding class IDs are as follows: ○ car - 2 ○ motorcycle - 3 ○ bus - 5 ○ truck - 7 ● Implement ByteTrack for video annotation and tracking. ByteTrack is used to track the detected vehicles in consecutive video frames. ● Set a threshold distance value to determine whether the vehicle is stopped or moving. If the calculated distance is below the threshold, the vehicle is considered stopped.

	Otherwise, it is considered moving. ● Implement a time threshold for a stopped car to be considered parked and create a dictionary, **park**, to record the frame count for that threshold. If the vehicle state remains stopped for the required time, change its state to parked until it starts moving again. ● Update the vehicle state for each tracked object by iterating through the **detections_dict** dictionary. For each **tracker_id**, compare the new and old detections and calculate the distance between their center points. ● Calculate the distance between the center points of bounding boxes for each frame using the Euclidean distance formula. The center points are derived by averaging the coordinates of the top-left and bottom-right corners of the bounding boxes. ● Use the calculated distance to determine the vehicle's state: Moving, Stopped, or Parked. ● To handle spikes of movement in stopped vehicles caused by detection algorithm issues, create another dictionary (**movement_count**) and time threshold (0.3 seconds) to maintain the original state when movement doesn't persist for longer than the specified duration. ● Update the **park** and **movement_count** dictionaries accordingly based on the vehicle's state.

MongoDB Atlas	Collections vehicleids: ● _id ● sourceVideo ● annotatedVideo ● id changedpositions: ● _id ● videoId ● tracker_id ● class_id ● status ● timestamp ● time ● frame ● changed
Connection Method	FastifyAPI Server
Notes on Implementation	The connection to the database is fairly simple through the use of FastifyAPI. It can be done by initializing the FastifyAPI via npm init fastify. From there we will be following a few different steps to properly create our endpoints for the database. ● .env ● controller.js ● model.js ● route.js After doing all of this and creating our endpoints, this gives us the basis for the server that will be used later on in the Front End. A great source of information was this video.
.env	The .env file is where we provide the connection URL to our mongodb database. As well as change our port, in case the current port is competing with another.

controller.js	The controller file is where we decide what information we'll want to end up pulling from our database. We create functions that will end up retrieving whatever we want. In the case of this project all we need is a single function that will return all the information pertaining to the car's information. We all do this for the video names.
model.js	This model is where we create our schema file. This will end up being a check making sure the information pushed to the database is known and doesn't end up as a surprise. In the case of this project while it isn't needed, it is great to use in order to test the project.
route.js	Finally the route.js is where we create our endpoints. Which make use of the functions we made in the controller.js to get the information from the database. In the case of this project we have two routes, one for getting the information for all of the vehicles. And on for getting all of the video names.

Implementation of Front End

Notes on Implementation	The front end was implemented using REACT, HTML, CSS, and JavaScript. For this reason we have broken up our different views into various components that send information to one another. The components are: ● Navbar.js ● SecurityView.js ● AnalystView.js ● TableMaterial.js They all have style sheets associated with them as well which are

	the same name but '.css' instead of '.js'. These can be modified to change the flow of the app as wanted.
Types of Views	Our implementation has two different views: 1. <u>Security View</u> The security view simulates a "live" video feed where the security guards can monitor all of the cars in the parking lot in real time. They are able to see if a crash occurs or even any sort of illegal parking. This is a simple view where we can only filter the table by the different states (Parked, Moving, Stopped). 2. <u>Analyst / Data Scientist View</u> The analyst or data scientist view is very similar to the security view. However, the analyst first must choose a video to analyze from the drop down in the top left hand corner which will then show the video and the corresponding data will fill the table. The analyst view also has the ability to filter the data by the states (Parked, Moving, Stopped). On top of this, the analyst is able to click on any row/column of the table and the video will jump to the corresponding time stamp for that event. The analyst view is also where future engineers can expand on this project. Below the video there is a section that is labeled "Analyze Data". This is where future engineers can create queries or insert different visualization tools to further inspect the data.
Navbar Component	The navbar is implemented in a very simple manner using the Link import from react-router-dom. This allows us to send a user to a specific link when they click on a button. ● The Security Feed button links to the Security View ● The Analytics button links to the Analyst View
ReactPlayer	ReactPlayer is a React component for playing a variety of URLs,

	including file paths, YouTube, Facebook, Twitch, etc. This is an existing react component that we are using in order to display the videos on the application. ● The url prop is set equal to the file path where we are storing the video. (This path changes when a different video is selected) ● We have the playing and controls props set to true so that when the video loads it is automatically playing and the users have access to the pause/play buttons. There are many other props that can be used for this component which can be found in the documentation, but these are the ones which have been of most relevance for our project. Documentation for react-player here.
Table Material Component	The Table Material component is a simple table created using Material UI. ● The table is filled with the data received from the corresponding API calls from either the Security View component or the Analyst View component. ● There are also Material UI Checkboxes above the data table which can be used to filter the data shown. The three: filters are Parked, Stopped, and Moving. ● The checkboxes are set to be checked by default so that all the data is shown at first. When a checkbox is unchecked the data is filtered using the filter() method which creates a shallow copy of a portion of a given array, filtered down to just the elements from the given array that pass the test implemented by the provided function. ● This component is also where we calculate the time to jump to when a table cell is clicked. The cells contain strings, so we use a function (handleCellClick) to convert the string to ints and then calculate the seconds which that amounts to.

	Documentation for MUI React Table Component. Documentation for Array.prototype.filter().
Security View Component	The Security View component is composed of two other components: ● ReactPlayer component to show the video (on the left hand side of the screen) ● Table Material component to show the table with the data associated with the video (on the right hand side of the screen) Since the security view is supposed to be simulating a "live feed", there is simply an API call to fetch the data on load of the page. The functionality of this component is for the security guard to monitor the parking lot and filter the data by the different states as needed.
Analyst View Component	The Analyst View component is also composed of two other components: ● ReactPlayer component to show the video (on the left hand side of the screen) ● Table Material component to show the table with the data associated with the video (on the right hand side of the screen) The Analyst View component does everything the Security View component does, but it has more functionality. The first thing an Analyst must do is select a video from the drop down at the top left hand corner of the page. This will then load the video and the table (the necessary variables are updated using an on change event of the video selected). The analyst may also click on any row of the table and the video will jump to the corresponding time stamp. This is done through an OnClick function which changes the timeToStart variable which is used to set the time of the video by using the useEffect from REACT.

	In order to add more videos to the Analyst View, import the videos from the video folder, and map them. For example: mapVideos.set('video_in_database'', 'import from video folder'); This will enable the video to be available on the dropdown menu, and the appropriate information will be filled out in the table. Documentation on onChange. Documentation on useEffect.
Video Storage	The videos are currently being stored as part of our app in a folder named videos. Future engineers have the ability to change from local storage to the cloud, but due to some time constraints this was the best way to complete our project.
Data Retrieval	The data is being retrieved from our MongoDB database using Fastify API calls. There are API calls to fetch the data in the following files: ● SecurityView.js ● AnalystView.js These calls are very simple and can be found in the definition of our getData variable.

Engineer's Notes to Run our Application

Github	In order to run our app you must first download our code. This can be done by accessing our GitHub which is linked at the end of this section. There are multiple ways to clone the repository. We suggest clicking on the green Code<> button and copying the web URL to clone the repository in VS code. Once you have the repository cloned, you can begin to run it using the Docker Commands in

	the next section. Find our GitHub here.
Run with Docker	In order to run the frontend and server within Docker, first clone the repository onto your local machine. Once this has been done, cd into the 'CSCE482_23S_2N1' directory. Now, run the following Docker command: 'docker compose up -d'. When opening a new tab in Mozilla Firefox or Safari, enter 'localhost:3000' as the url. After a few moments, you should now see our application. To run the machine learning model, navigate to the 'machine_ml' directory within the 'UIstarter' directory. Here, you can input a video from another path to be processed by the machine learning model, which can then be output into a user defined path. First, run the following docker command to build the image: 'docker build -t your_image_name .' Replace the parameter 'your_image_name' with the name you would like to give the Docker image. Now, you can run the following command to process a video and output it to our local machine. 'docker run -v /path/to/video_directory:/app/videos -v /path/to/desktop:/app/output -e VIDEO_NAME="your_video.mp4" your_image_name python vehicle_class.py' For '/path/to/video_directory' replace this with the directory in which the video is located. Also, replace '/path/to/desktop' with where you would like the processed video to be output. Replace "your_video.mp4" with the name of the video to be processed and replace "your_image_name" with the name of the Docker image.

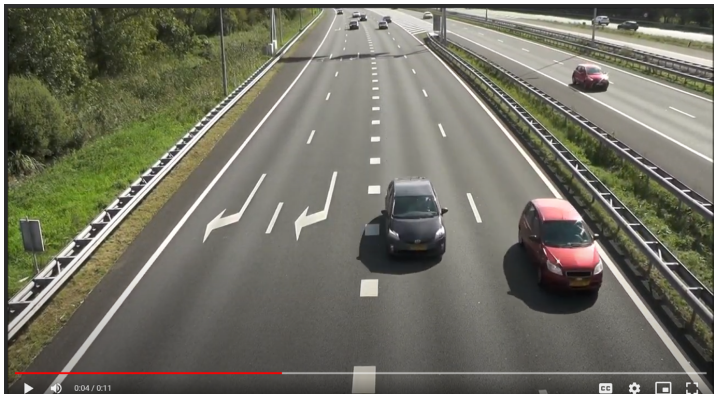
5 Experimental results

To validate our system, we tested it with multiple videos. We used the pre-trained YOLOv8 model and selected the specific classes to detect cars, motorcycles, buses, and trucks:

```
# dict maping class_id to class_name
CLASS_NAMES_DICT = model.model.names
# class_ids of interest - car, motorcycle, bus and truck
CLASS_ID = [2, 3, 5, 7]
```

We ran tests for our selected classes with short videos of edge cases, such as trucks and motorcycles, to ensure the system could handle alternative scenarios:

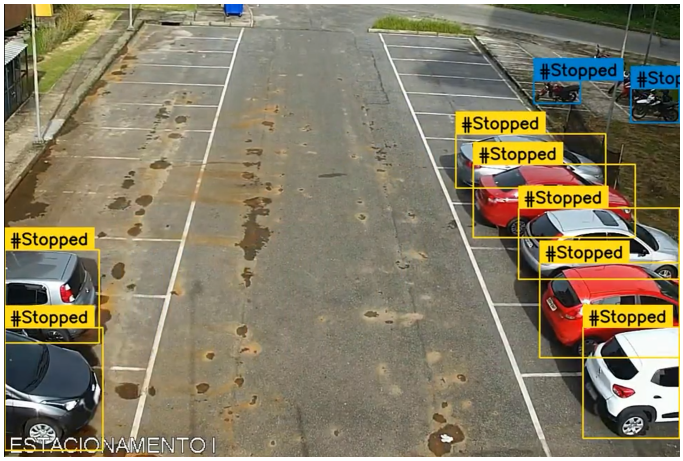
Using a traffic video, we determined the accuracy of detection based on car distance from the camera as cars approached:



In order to determine the difference between a moving and stopped vehicle, we used the Euclidean distance formula to calculate the pixel distance of the center of the bounding box from the last to that of the current frame and set a distance threshold. We determined a distance threshold of at least 1.5 pixels was sufficient for a car to be considered moving:

```
    # Calculate the distance between the two center points using the Euclidean distance formula
    distance = math.sqrt((box1_center[0]-box2_center[0])**2 + (box1_center[1]-box2_center[1])**2)
    # Labels for Stopped, Moving, Parked
    distance = round(distance, 2)
    if distance < 1.5:
      detections_dict.get(tracker_id).append(str(distance) + " Stopped")
    else:
      detections_dict.get(tracker_id).append(str(distance) + " Moving")
  else:
    detections_dict.get(tracker_id).append(-1)
print(detections_dict.get(tracker_id))
```

We used various methods to capture vehicle states, incrementally improving where there were issues with the detections. We tested our approach by processing shorter parking lot videos/videos with less cars. This is a typical parking lot as ascribed by our project manager:



Notice that the cars are horizontally parked and there are about 9 cars and 3 bikes.

We gradually worked our way up to a longer 5-minute video with more than 30 cars:

Initially the code was breaking due to vehicle detections being lost, leading to invalid tracker IDs. This would occur when vehicles went out of frame, were too far away/small, or were otherwise obstructed by other objects. To get around this issue, we had to reset tracker IDs that lost detection, giving them a new value.

Another challenge we encountered was an issue with the detection producing faulty/jittery bounding boxes especially when a moving vehicle was too close to a stopped or parked vehicle. This had the effect of destabilizing the actual distances calculated between the center of vehicles from one frame to the next, creating spikes of motion exceeding our 1.5 pixel distance threshold. In order to fix this issue we tried a few different methods:

1. If the distance measure between the current and previous frame exceeds 30%, disregard the current state and keep that of the previous:

```
    # Labels for Stopped, Moving, Parked

    #tracking
    if len(old_tracker_id) < 5:
      prev_state = ""
    else:
      prev_state = old_tracker_id[4]

    if distance > old_tracker_id[3]*0.3 and old_tracker_id[3] < 1.5:
      detections_dict.get(tracker_id).append(distance)
      detections_dict.get(tracker_id).append(prev_state)
    elif distance < 1.5:
      detections_dict.get(tracker_id).append(distance)
      detections_dict.get(tracker_id).append("Stopped")
    else:
      detections_dict.get(tracker_id).append(distance)
      detections_dict.get(tracker_id).append("Moving")

  else:
    detections_dict.get(tracker_id).append(-1)
    detections_dict.get(tracker_id).append("")
  print(tracker_id,detections_dict.get(tracker_id))
```

This method was largely unsuccessful in minimizing the distance spiking problem.

2. Adjusting ByteTrack arguments:

```
@dataclass(frozen=True)
class BYTETrackerArgs:

    track_thresh: float = 0.25 #was 0.25
    track_buffer: int = 30 #was 30
    match_thresh: float = 0.8 #was 0.8 The threshold for matching detections to tracks in the tracker.
    aspect_ratio_thresh: float = 3.0 #was 3.0 The maximum aspect ratio of a bounding box that will be considered valid.
    min_box_area: float = 1.0 #was 1.0 The minimum area of a bounding box that will be considered valid.
    mot20: bool = False
```

We researched what each ByteTrack variable does and attempted to improve tracking by playing with the values. However, the results were not much different than before.

3. After adding a parked state Using the same method to create a “parked” state using a timer threshold (countdown) and dictionary to record frame state and monitor vehicle state consistency over time (if the car is stopped for the entire length of time, it is considered parked) , we decided to try the same method to smooth the spiking phenomenon:

```
        # Labels for Moving, Stopped, Parked
        distance = round(distance, 2)
        if distance < 1.5:
            if tracker_id in park.keys():
                park[tracker_id] = park[tracker_id] + 1
            else:
                park[tracker_id] = 1
            if park[tracker_id] > countdown:
                detections_dict.get(tracker_id).append("Parked " + str(distance))
            else:
                detections_dict.get(tracker_id).append("Stopped " + str(distance))

        else:
            park[tracker_id] = 0
            detections_dict.get(tracker_id).append("Moving " + str(distance))

    else:
        detections_dict.get(tracker_id).append(-1)
firstTime = False
print(detections_dict.get(tracker_id))
```

We used another dictionary and time threshold in the same way as the “parked” state solution only for maintaining a persistent state through movement spikes (the car has to be constantly “moving” for at least 0.3 seconds for it to switch from “parked”/”stopped” to register as “moving”):

```
fps = video_info.fps
parkCountdown = 10 # how many seconds to wait before considering a stopped car as parked
moveCountdown = .3 # how many seconds to wait before switching states (buffer for spikes)
# adjusted for frames
countdownP = fps * parkCountdown
countdownM = fps * moveCountdown
movement_count = {} # Initialize movement_count dictionary

park = {}
```

```
distance = round(distance, 2)
if distance < 1.5:
    if tracker_id in park.keys():
        park[tracker_id] = park[tracker_id] + 1
    else:
        park[tracker_id] = 1
    if park[tracker_id] > countdownP:
        state = "Parked"
    else:
        state = "Stopped"

    # Reset movement_count for this tracker_id
    movement_count[tracker_id] = 0
else:
    if tracker_id in movement_count:
        movement_count[tracker_id] += 1
    else:
        movement_count[tracker_id] = 1
    # code to manage spikes
    # Check if the car has been moving for longer than 0.3 seconds
    if movement_count[tracker_id] > countdownM:
        park[tracker_id] = 0
        state = "Moving"
    else:
        # Maintain the original state when movement doesn't persist for longer than 0.3 seconds
        if park[tracker_id] > countdownP:
            state = "Parked"
        else:
            state = "Stopped"

detections_dict.get(tracker_id).append(f"{state} {distance}")
```

With regards to FPS, we initially were reaching 70 FPS for 10 cars. We calculated this by dividing 1 second/ the inference time in seconds. This was run in Google Colab with an NVIDIA k80 GPU.

```
0: 640x384 (no detections), 62.0ms
Speed: 3.2ms preprocess, 62.0ms inference, 0.6ms postprocess per image at shape (1, 3, 640, 640)

0: 640x384 (no detections), 62.0ms
Speed: 2.6ms preprocess, 62.0ms inference, 0.8ms postprocess per image at shape (1, 3, 640, 640)

0: 640x384 1 person, 39.1ms
Speed: 3.4ms preprocess, 39.1ms inference, 1.6ms postprocess per image at shape (1, 3, 640, 640)

0: 640x384 1 person, 38.4ms
Speed: 3.5ms preprocess, 38.4ms inference, 1.6ms postprocess per image at shape (1, 3, 640, 640)

0: 640x384 1 remote, 37.1ms
Speed: 2.9ms preprocess, 37.1ms inference, 1.5ms postprocess per image at shape (1, 3, 640, 640)
```

We were very happy with this value initially. However, after weeks of development and adding support for stopped, moving, and parked, we wanted to calculate our values again. We also determined that instead of using YOLO inference time, that we should calculate our FPS value

using frames over time for the whole process to run. This was done because it more accurately reflects what our client will experience in production.

We measured the system's speed, and it exceeded our goal of 10 FPS for 5 vehicles, achieving 11.77 FPS with 37 cars. We found that our value had dropped significantly from the previous value of 70 FPS with 10 cars. Although we are still above the benchmark, we realize that further optimizations and improvements can be made in this regard.

One way to increase the FPS is to retain the detections across multiple frames, and not evaluate every frame. Realistically, this makes much more sense. For example, 10 cars in a 5 second 30FPS video would generate 1500 separate data points, an absurd amount of data points that is not sustainable in production. Furthermore, we found that evaluating detections every other frame would increase the accuracy of our ML model, since there is less room for frame by frame jitter. Going every other frame increases the fps to 16.7 FPS for 37 cars. This represents a 40% improvement, and is well above the benchmark suggested by our client. We found that going every other frame results in both a smooth and accurate video for most situations.

6 User's Manuals

Hardware Installation

There is no hardware installation necessary for our project.

Software Installation

Download Code	In order to download our system you must access our Github by clicking here. This will take you to our repository which is where we store our code. There are a couple of ways you may download our project: 1. Please ensure that you have Visual Studio Code downloaded 2. On our Github repository click on the green button that says "Code<>" on it. 3. There it shows you the multiple options. 4. Click on the button next to the link shown under

	"Clone." This gives you the URL which can be pasted in Visual Studio Code to clone the repository. 5. Open Visual Studio Code 6. Once the app opens there is a section that says "Start" and the third option is "clone Git repository" 7. Click the third option which opens up a text box at the top. 8. Paste the URL you copied from our Github repository and press "Clone from URL" 9. This will take a couple seconds to download. 10. Once the download is done it asks if you would like to open the project. Click yes. 11. Once the project is open you can go through any of the files that you desire to modify. The code for the front end is under the UIstarter/vehicle_classification folder. 12. If you wish to run the code you can open a terminal by finding the "Terminal" button at the toolbar at the top. 13. Click "Open new terminal" 14. From here you may run our system using Docker as described in the next section.
Run Project with Docker	In order to run the application that includes both frontend and server: 1. Navigate to the 'CSCE482_23S_2N1' directory 2. Run the command: `docker compose up -d` 3. In a Mozilla Firefox or Safari window, input 'localhost:3000' into the URL In order to process a video with the machine learning model:

	1. Navigate to the ‘vehicle_ml’ directory within the ‘UIstarter’ directory 2. Build the docker image using the following command: ‘docker build -t your_image_name .’ where the parameter ‘your_image_name’ is the name you would want to give the Docker image 3. Once the image has been successfully created, run the following command to input a video name from a directory on a local machine and output the processed video on the local machine: ‘docker run -v /path/to/video_directory:/app/videos -v /path/to/desktop:/app/output -e VIDEO_NAME="your_video.mp4" your_image_name python vehicle_class.py’ a. Replace ‘/path/to/video_directory’ with the directory in which the video is located on the local machine b. Replace ‘/path/to/desktop’ with the directory in which you would like the processed video to be output c. Replace ‘your_video.mp4’ with the name of the video to be processed d. Replace ‘your_image_name’ with the name of the image you gave to the Docker image 4. Navigating to the directory given to output the processed video should show a video that has been run through the machine learning module.

Operation Instructions

There are two types of users for our system: security guards, and data analysts.

Characteristics of Security Guards:

Security guards will use the system to monitor the parking lots through a live feed. They are able to filter the data displayed by the different states of the cars. The security guards most likely do not have a technical background, so the app is designed to be very simple and straightforward to use. Security guards are meant to be very prepared and have quick reflexes in case an incident occurs in the parking lots. This is why our app is designed to have a live feed of the parking lot at all times. This will allow for a quicker reaction time since they are able to see events unfolding in real time. Security guards also have the responsibility of ensuring illegal parking is not occuring, and this is much easier to monitor through a video feed than by walking through the aisles of the parking lot.

Characteristics of Data Analysts:

Unlike security guards, data analysts do have a technical background, so their view is a bit more elaborate. Analysts have to look through a lot of data and find patterns and ways to improve the traffic flow throughout the day. They are very detail oriented individuals who need to use tools that make their jobs easier. The app is designed to allow for them to have a very organized experience. Analysts are able to choose from the videos that have been uploaded on the dropdown. Once they choose a video they are able to watch the video and filter the data by the different states. They are also able to click on a certain row of the table and the video will jump to that timestamp so they are able to see what was happening during that time.

Security Guard Operation	Security Guards have the live view of the 2 views. This means they may not change the video feed that is playing, they can simply filter the results that they would like to see. For example, if you only want to see the data for the cars that are moving you have to uncheck the boxes for parked and stopped. This view is used mostly to monitor the parking lot and ensure the safety of the customers, so there is not a lot of upkeep or operations that can be done.
Analyst Operation	The analyst has a slightly more in depth view. They

	are able to choose a video from the dropdown and filter the results as well as jump to specific times in the videos. Analysts will most likely need to add new videos they want to analyze to the app.
Filter By State	If you would like to filter the data on the table to only show a specific state, then you have to uncheck the other checkboxes which are above the table. For example, if you only want to see the results of the cars that are parked, then you must uncheck the moving and stopped boxes. If you would like to see all the data again simply check all the boxes again.
Jump to a Specific Time	In order to jump to a specific time in the analyst view you may click on any of the column values within that row. For example, if you would like to jump to the time in the video where the car with id 22 was moving you may choose to click on the id, the state, or the timestamp within that row. You should see the video immediately jump to that timestamp.

7 Course debriefing

In reflecting on our team management style, we valued the open and inclusive environment we fostered, allowing each team member to express their ideas and opinions. The absence of a de facto leader ensured that everyone's perspective was heard and considered before reaching a consensus. This approach helped the decision-making, as everyone felt valued and included in the project's development.

However, there were areas where we could have improved. In hindsight, we realize that scheduling more frequent group meetings would have been beneficial. Meeting only twice a week sometimes led to miscommunications and lack of clarity about individual tasks and progress. More frequent meetings, perhaps through daily stand-ups, would have allowed us to stay better aligned, improve our communication, and address any roadblocks or challenges more effectively. Also talking about group meetings, we also should have begun meeting with Andrew, Devin, and the Walmart team a bit more outside of class in order to understand their expectations a bit more as well as to get a

better input on how our project was heading. Though in the end it seems everything turned out alright, if we had introduced this even earlier rather than about half way through production, we may have been able to implement even more.

Another aspect we would have approached differently is the Docker component of the project. We underestimated its complexity and importance, and as a result, we relegated it to the back burner for too long. This delayed decision ultimately led to a slightly rushed implementation, which may have impacted the overall quality of the project. In the future, we would prioritize key technical components like Docker from the outset to avoid last-minute scrambling and ensure a more polished final product. A good time frame would have been to aim for creating a Dockerized container of the program using the code we had at the time of the Critical Design Review,

Despite these challenges, there were several aspects of our team management style that we would retain in future projects. For instance, the open communication and democratic decision-making processes fostered a sense of camaraderie and team spirit that contributed positively to the project's outcome. Moreover, our willingness to learn from our mistakes and adapt along the way helped us to grow as a team and improve our overall performance.

In conclusion, our team management style was largely successful in promoting an inclusive and collaborative atmosphere. If given the chance to do the project again, we would maintain this approach while incorporating more frequent meetings and better prioritizing key technical components. By refining our team management style based on these lessons learned, we believe that we would be better equipped to execute future projects with even greater success.

In terms of safety and ethical concerns, our project indeed presents several important issues that would need to be carefully addressed, particularly if it were to be commercialized. To begin with, we took great care to ensure that no extraneous information about the vehicles in the videos was collected or tracked. By using internal IDs to identify and monitor vehicles, we eliminated the need to track specific details like the make or model, thus mitigating potential privacy concerns and negative judgments from users.

However, there are additional steps that we would have taken if given more time or resources, or if we were to undertake the project again. One such step would be to implement a blurring algorithm for people's faces and license plates in the videos. This would help address privacy and safety concerns by further anonymizing individuals and their vehicles, ensuring that no personally identifiable information is inadvertently revealed or misused. Of course with those implementations it would have to work well with how our current processing of videos is done and ensure it doesn’t counter it in any way and cause errors.

The process of blurring faces and license plates would likely be incorporated into the machine learning stage of the project, where the video is initially processed. By adding this step to the workflow, we could ensure that the anonymized data is stored in the database and subsequently displayed in the final output. Implementing this feature would not only enhance the ethical considerations of our project but also demonstrate our commitment to user privacy and safety. Though that being said the blurring of faces and or license plates could have been its own separate project entirely, therefore I don’t think it would have been reasonable for us to implement it unless

we were given a strong basis to work off of, or maybe even another semester to work on the project. But in a commercial setting it would certainly be something we would have to take into consideration.

In summary, our project carries certain safety and ethical concerns that warrant careful consideration. While we have taken steps to address these concerns by using internal IDs and not collecting extraneous vehicle information, we recognize that further measures, such as blurring faces and license plates, would be necessary to fully address these issues. If we were to do the project again, we would certainly need to take these measures into account and hopefully if given the time, implement them as well, ensuring that our product not only meets its functional objectives but also upholds the highest standards of privacy, safety, and ethics.

We did test our product and confirmed that it works as proposed under the scenarios we examined. However, there are additional situations that we have not yet tested, which may affect the performance of our product. If we were to undertake this project again, we would expand our verification and testing procedures to cover these additional scenarios.

For instance, we did not thoroughly test our product under inclement weather conditions, such as heavy rain or snow. These environmental factors could have a significant impact on the data quality and the camera's ability to recognize vehicles. Factors like raindrops on the camera lens or large puddles causing reflections might lead to inaccuracies in the data since they may cause unknown errors to occur. Not only that but snow covering the ground or vehicles could have also rendered the vehicles unrecognizable on the machine learning side due to the fact that they may simply cover everything up, or change the vehicles silhouette so much that they look completely different from what the machine learning side may expect. Testing under these conditions would provide valuable insights into the robustness of our product and help us identify any necessary improvements.

Another area where further testing would be beneficial is in different lighting conditions. Although we tested our product on videos with varying lighting, we have not yet explored its performance in parking lots at night. This limitation raises concerns about the camera's ability to detect vehicles in low-light conditions, particularly darker-colored cars like black or navy vehicles. Conducting tests under these circumstances would allow us to evaluate the product's effectiveness and make any necessary adjustments to the machine learning algorithms. We could have also tested out to see if the machine learning system could work on cameras with night vision. With night vision cameras' unique view, we aren’t too sure how they would interact with our current product, and if they would cause any errors. It just goes to show that even more testing on different scenarios would have helped our final product immensely.

Also, creating and using our own test videos would have allowed us to design more targeted testing scenarios to evaluate the product's performance in specific situations. While this was not pursued due to time constraints, it would be a valuable addition to our testing procedures in the future. By simulating various real-world scenarios in our own videos, we can ensure that our product performs optimally under a wide range of conditions.

And finally, running more videos in general. This would have been a great help to see if there are any errors with the way the machine learning system handles different scenarios, and had we done this we could have also possibly answered some of the concerns above regarding different lighting or weather conditions for various parking lots. By running more videos, we would have gotten more information in various different ways which would have proven to be useful no matter what.

In summary, while our product was tested and found to work as proposed, there are several situations that have not yet been explored. If we were to do this project again, we would enhance our verification and testing procedures to include inclement weather, different lighting conditions, and the use of custom test videos. This would help us better understand the limitations of our product and make any necessary improvements to ensure its success in a broader range of real-world applications.

8 Appendices

We got the idea of how to format our Implementation Notes from:

> "Implementation Notes," *Implementation notes*. [Online]. Available: https://people.cs.pitt.edu/~chang/231/seminars/S06template/templates/implementation-notes.html. [Accessed: 26-Apr-2023].

References

(https://ieee-dataport.org/sites/default/files/analysis/27/IEEE%20Citation%20Guidelines.pdf)

[1] "Parking lots & distracted driving," *National Safety Council*. [Online]. Available: https://www.nsc.org/road/safety-topics/distracted-driving/parking-lot-safety. [Accessed: 02-Feb-2023].

[2] "Wireless M-Gage Node," *Banner Engineering*. [Online]. Available: https://www.bannerengineering.com/us/en/products/wireless-sensor-networks/wireless-sensors/wireless-m-gage-node.html?sort=1#all. [Accessed: 12-Feb-2023].

[3] "Monitoring software," *Parksol*, 26-Aug-2022. [Online]. Available: https://parksol.lt/solutions/monitoring-software//. [Accessed: 12-Feb-2023].

[4] Z. Yin, H. Xiong, X. Zhou, D. W. Goldberg, D. Bennett, and C. Zhang, "A deep learning based illegal parking detection platform," *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on AI for Geographic Knowledge Discovery*, 2019.

[5] *A Deep Learning Based Illegal Parking Detection Platform*. YouTube, 2019.

[6] S. parking, "Smartpark system," *Smart parking*, 07-Apr-2022. [Online]. Available: https://www.smartparking.com/smartpark-system. [Accessed: 12-Feb-2023].

[7] R. Stiawan, A. Kusumadjati, N. S. Aminah, M. Djamal, and S. Viridi, “An ultrasonic sensor system for vehicle detection application,” *Journal of Physics: Conference Series*, vol. 1204, p. 012017, 2019.

[8] “Papers with code - mot17 benchmark (multi-object tracking),” *The latest in Machine Learning*. [Online]. Available: https://paperswithcode.com/sota/multi-object-tracking-on-mot17. [Accessed: 13-Feb-2023].

[9] Emergen Research, “Security cameras market size, share: Industry forecast by 2030,” *Security Cameras Market Size, Share | Industry Forecast by 2030*. [Online]. Available: https://www.emergenresearch.com/industry-report/security-cameras-market#:~:text=Market%20Synopsis,18.7%25%20during%20the%20forecast%20period. [Accessed: 11-Feb-2023].

[10] D. Hardawar, “FCC Bans Telecom and video surveillance gear from Huawei, ZTE and other Chinese companies,” *Engadget*, 25-Nov-2022. [Online]. Available: https://www.engadget.com/fcc-officially-bans-telecom-and-video-surveillance-gear-from-several-chinese-companies-003040729.html. [Accessed: 11-Feb-2023].

[11] P. N. Service, “Turn off that camera during virtual meetings, environmental study says,” *Purdue University News*. [Online]. Available: https://www.purdue.edu/newsroom/releases/2021/Q1/turn-off-that-camera-during-virtual-meetings,-environmental-study-says.html. [Accessed: 02-Feb-2023].