Parking Availability System

Preliminary Report

Erik Meurrens, Benjamin Simonson, Ryan Jalloul, Evan Tobon, Samer Khatib

**Abstract**

The Parking Availability System is a low cost, portable edge device designed to operate within parking facilities to monitor occupancy. Within parking facilities, it tracks traffic entering and exiting the facility using a YOLOv11n lightweight computer vision object detection model developed by Ultralytics, running on a Raspberry Pi 5 (RPi5). This real-time data, along with the facility’s maximum capacity, is stored in an Amazon RDS SQL database via the AWS cloud services. A mobile application can access this data and update information for users worldwide on the current occupancy of any parking facility on the University of Florida (UF) campus.

**Introduction**

The limitations of parking at the University of Florida (UF) have a negative effect on campus life for students, faculty, and staff, acting as a source of stress, wasted time, additional traffic congestion, and increased vehicle emissions. Live parking data for the campus is only available at a select few locations, making it difficult for those who must drive to campus to find parking during busy periods, as it is impossible to check real-time availability prior to arriving. That lack of live parking data is what this project aims to address. The goal of the Parking Availability System (PAS) is to provide real-time, accessible parking capacity tracking information for individuals who wish to park at the parking facilities on campus. This information would be beneficial for off-campus commuters and visitors to campus, reducing the stress, frustration, and confusion drivers can feel trying to find parking, as well as the congestion that can occur at parking facilities, during busy traffic periods on campus. Our project, the PAS, was inspired to help the UF community, alleviating the symptoms of the lack of easily available parking resources for the campus and promoting quicker and more reliable transportation to and from campus, by offering a way to access parking availability information in real time via PAS.

**Background**

Currently, there are existing implementations that provide onsite observability for garage capacities. The solution currently employed by the University of Florida is the Genetec AutoVu system that uses an automatic license plate recognition service to identify and read license plates and count vehicles that are currently in the facility. While effective in license plate identification, this solution lacks availability across campus facilities, only being available in Parking Garage 14. This omission inconveniences drivers searching for available parking when none exists. Additionally, [1] argues in favor of a smart parking system with real-time availability accessible via a mobile app, from the perspective of two civil engineering researchers at UF. However, the proposed system in [1] as well as the current system comes with a significant cost burden and is facilitated via subscription services that may go underutilized. Specifically, this system maintains a “$6000 servicing cost, $11,000-$8,000 per lane, and $2,495 per camera” [1]. In contrast, our project will provide a much more affordable solution to the AutoVu as it will only utilize one-time equipment cost and installation fees. Our project will also provide virtual observability into parking facility current availability to streamline an individual's search into finding a parking spot.

In terms of an existing need and demand for a smart parking system on campus, a survey conducted by our group at UF demonstrated the desire to alleviate parking stresses. Approximately 70 UF undergraduate students were surveyed, and as demonstrated in Table I, we found that 89% of respondents expressed negative attitudes towards the existing campus parking situation. Anecdotes from respondents included having to spend upwards of 10 minutes just to find parking, being unable to find parking near a desired location, and giving up after circling a parking facility without finding an available spot. These results conclude that parking difficulties cause stress, frustration, congestion, and are a waste of time. There is a need to enhance the observability of parking availability for UF students and PAS is a cheap yet effective solution.

TABLE I. RESULTS FOR ATTITUDES RELATING TO CURRENT PARKING SITUATION

|  |  |
| --- | --- |
| **Question** | **Negative Response Rate Towards Current System** |
| “Overall, I am satisfied with the current parking situation on-campus.” | 89% |
| “It is easy to find a parking spot within my designated parking permit color.” | 80% |
| “Parking availability impacts my campus experience (e.g., attending classes, participating in events, etc.)” | 91% |
| “I avoid driving on campus due to the current parking situation.” | 90% |

For the mobile application component, our project will be building off of the prior art of UF Computer Science alumnus Drew Gill and his EZ Park UF codebase, with his permission. This app, when it was released three years ago but eventually removed from app stores due to lack of support, provided a map for users to view details about parking locations across campus, but did not offer a feature for viewing real-time availability at garages. To speed up development of PAS and apply our system to a user-facing component, we built upon his codebase to implement the mobile application component for our system.

**Design**

To achieve the goal of the Parking Availability System, providing real-time parking information easily accessible to users, our team developed four critical components: a mobile application, an edge AI device to be deployed in parking facilities, the car-detection pipeline, and a database.

*External Interface*

[TODO: Raspberry Pi & Mobile App]

*Persistent State*

[TODO: Database]

*Internal Systems*

[TODO: Car-detection pipeline]

Interface

*Pi Configuration Script*

The Raspberry Pi configuration script’s target user is not the primary target user of the system, that is commuters and people looking for parking on campus, but system administrators and those looking to deploy the system in a parking garage. Additionally, it acts as a quality-of-life feature for us as developers of the system. The purpose of the script was to ensure that a fresh Raspberry Pi can be set up in a more automated, seamless way so that scaling the system to other garages could be feasible, configuration options and settings for the Pi can be documented, and that test devices could be reformatted easily if needed.

The Raspberry Pi configuration script is a bash script run via the command line. Before using the script, it must be downloaded from the public project repository using the command git clone <https://github.com/emeurrens/parking-availability-system.git> -b rpi\_files. The user can then run the script by navigating to the clone repository directory parking-availability-system/Desktop/setup and running the command bash pi\_setup.sh. Afterward, the user interacts with the script by following the script’s instructions in the command line, providing keyboard input when requested by the script to control the execution and configuration settings.

*Mobile App (Alpha Build)*

Normal users will be able to interact with the system and retrieve real-time parking facility data using a mobile app developed as part of the system. As seen in the figure below, users will be able to navigate a map of the campus and view the locations of parking facilities plotted on the map, colored according to their corresponding decals. Users can either select a location on the map page, or search for a location in a list view page. Additionally, users can filter locations based on decal, name, and restriction times, which affects the locations available on the map and list view. Once a user has selected a parking location, they will be able to view the facility’s parking information in the parking details page on the bottom ribbon. This page’s details are updated periodically by polling the LOT table tracking parking facilities in the backend, cloud PostgreSQL database.

The mobile app implements all the features necessary to be able to view real-time parking facility updates by reading entries from the LOT table in the backend database. The UI is intuitive, and users can quickly navigate to the parking information screen for a specific parking facility in no more than three taps. Navigation of the UI takes inspiration from other navigation apps, so controls should feel intuitive, especially for frequent users of such apps. Additionally, it implements Google Maps API functionality to provide users with an accurate and familiar top-down view of their location. Finally, with the resources provided by a previous endeavor into improving parking observability systems, the mobile application can also filter facilities on campus by the parking pass clearance you have, for example: red, green, orange, etc. This will also serve in effectively guiding users to valid facilities and informing them of the facility’s occupancy count.

A screenshot of a phone

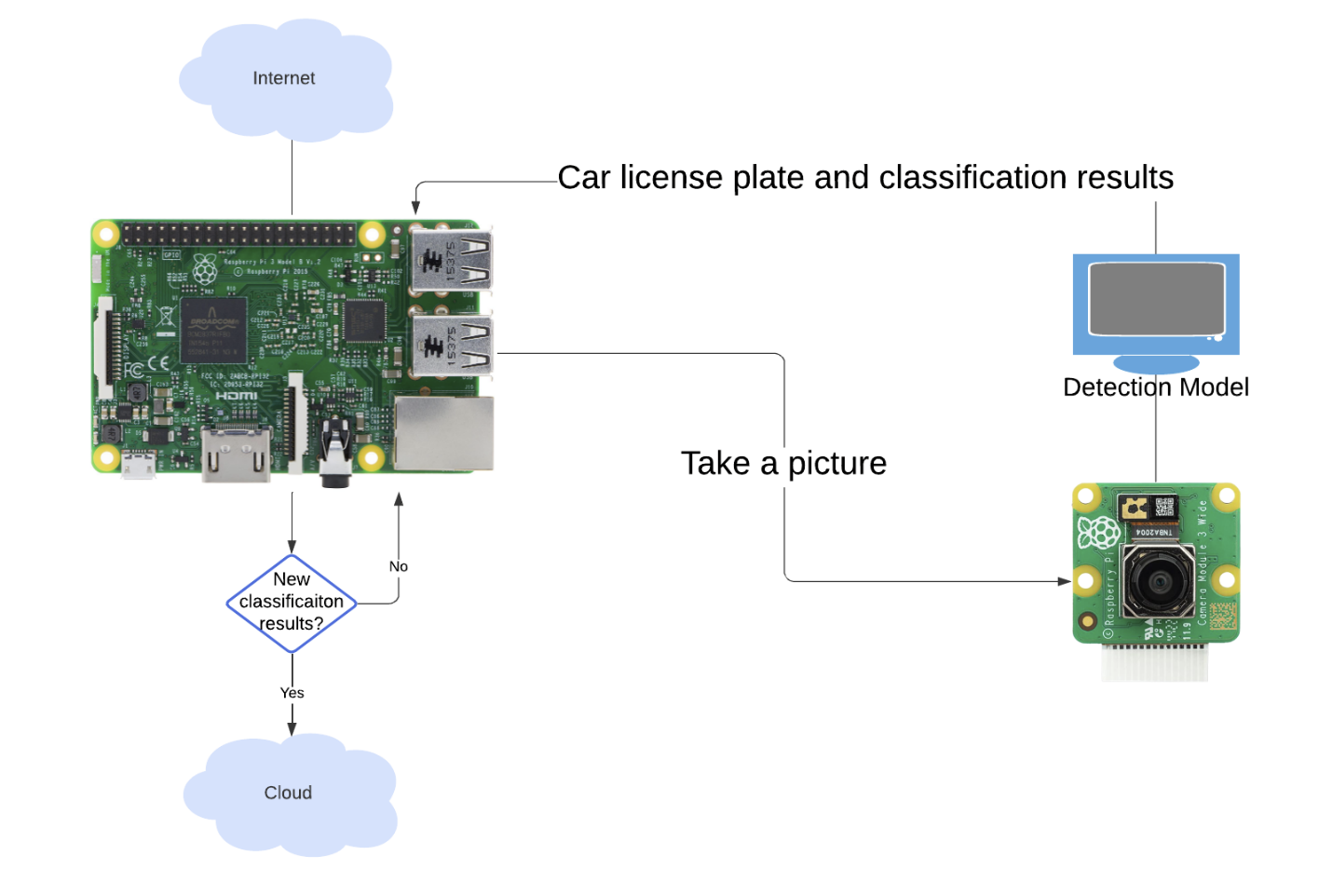
Description automatically generated

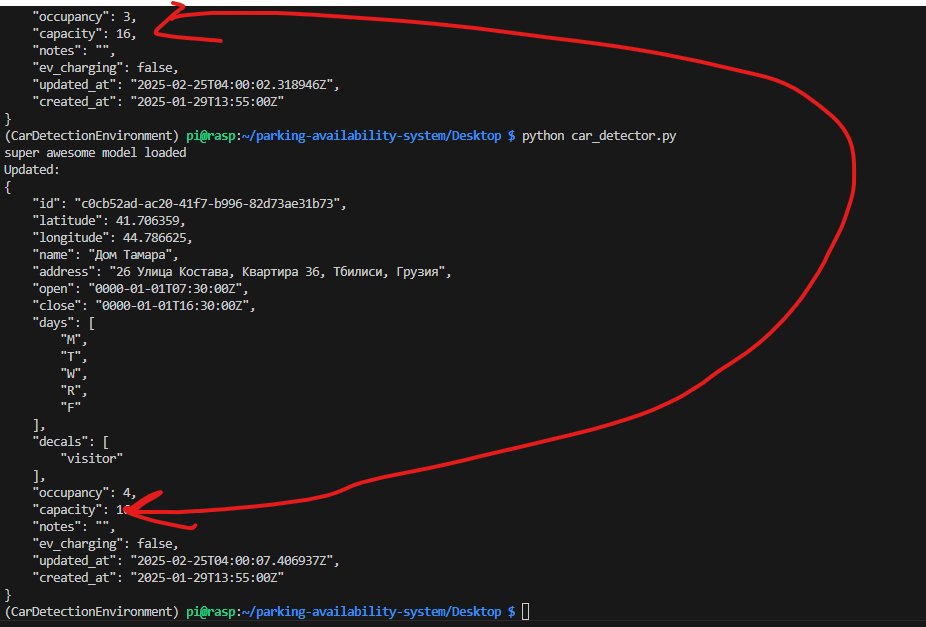
**Figure 1.** Parking App Frames

External Interface

*Raspberry Pi*

The RPi is configured to a camera to continuously capture images of its environment. These pictures are then fed into the object detection model to determine whether a license plate is present in one of the images. If it is, the RPi queries a PostgreSQL database to retrieve the parking facility occupancy count. Upon retrieval, the count is incremented or decremented depending on if the RPi is facing the exit or entrance of the parking facility. The license plate values are then recorded and added to or subtracted from the database. The update method functionality can be seen in Figure 8, where the RPi is able to query a parking facility, like UF’s Parking Garage 14. From this, there is an occupancy integer attribute that can be adjusted accordingly. Once this is all done, the cycle repeats all over again. See the video for further details and code review.

**Figure 2.** RPi Flow Diagram



**Figure 3.** RPi querying and updating the occupancy values of a lot upon detecting a car

The external interface relies on the camera taking pictures fast enough to constantly provide the model with an accurate and timely representation of the environment. When a vehicle enters the photo frame, the model is fed the images and observes the frames provided by the photo feed from the camera. Our group makes the assumption that all motor vehicles will have a license plate, if an entity does not have a license plate, then it is not a motor vehicle.

*Mobile App (Alpha Build)*

The app communicates with the PostgreSQL database API endpoints via HTTP requests to perform CRUD operations on the data tables. Despite wrapper functions being developed for all supported operations on the LOT table, realistically the app only calls two of these functions —`getAllLots` and `getLot` — in order to retrieve the most recent lot data.

**Tools, Design Constraints, and Engineering Soundness**

The mobile application was developed within the Android Studio IDE, providing IntelliSense error checking and linting capabilities as well as an emulator allowing for building the project on an emulated device to test and verify functionality from a user’s perspective. The Flutter framework was used to build a functional, cross-platform application using the Dart programming language, implementing in-depth debug tools to view perform breakpoint traversal, memory viewing, and application performance metrics to further verify functionality. The Google Maps Platform SDK was used to offer a map API to implement navigation and location functions familiar to users so that they can seamlessly and intuitively orient themselves on campus and navigate parking locations. The application employed HTTP requests to communicate with the backend database through a RESTful API that employed CRUD operations.

The edge device was developed through various software and physical means. To develop the physical housing, SOLIDWORKS CAD software was used to develop physical models to be 3D printed. The physical board our team programmed to implement the device was a Raspberry Pi 5, offering a 4-core ARM A76 processor, operating at a maximum frequency of 2.4 GHz, and 8 GB of RAM. Software on the device was developed within a 64-bit Linux environment, and frequently employed the SSH protocol to remotely access and program the device. The APT and PIP package managers were used to download dependencies needed to implement the car-detection pipeline. Most additional software on the board was programmed using bash and Python.

The ESP32 was developed using the Arduino IDE. The ESP32 contains a WiFi library, and this was used in the configuration of the NAT router. The Arduino IDE allows you to compile and upload the code to the board. Further, the Arduino IDE allows you to download an esp32 library, which lets you pick from a range of esp32 boards. T

[TODO: ESP32 Network repeater]

[TODO: Car detection pipeline]

[TODO: Remote database]

**Needs and Impact Analysis**

**Results**

*Alpha Testing Results*

*Raspberry Pi License Plate Detection Pipeline*

A goal of the project is to get as close as possible to an inference speed of 10 frames per second (100 milliseconds) without significantly sacrificing model performance scores. Utilizing processed image inputs into the model at a size of 640x384 pixels, we were able to achieve the resulting statistics in Table 1 on the Raspberry Pi 5, without any deliberate tuning or changes of the model to improve inference speed.

TABLE 1. License Plate Detection Inference Speed

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Input Size** | **MinTime** | **MeanTime** | **MaxTime** | **MaxFPS** | **MeanFPS** | **MinFPS** |
| 640x384 | 203.4 | 271.14 | 373.4 | 4.916 | 3.688 | 2.678 |

Even under ideal conditions, the YOLOv11n model achieves just under half of the stated goal of 10 frames per second. To get closer to our stated goal in developing our Release Candidate, we aim to improve our performance by investigating model optimization methods, such as dimensionality reduction and pruning, hardware/software optimization methods, such as ensuring maximum core utilization and overclocking the processors, or alternative models to get closer to our 10 frame per second inference goal.

*Raspberry Pi Persistent State*

A goal of the RPi’s persistent functionality was that it could perform its task of taking a picture, inferencing data from the picture, and manipulating the results accordingly without trouble. While the RPi demonstrated consistent success in recognizing license plates with high confidence, see Images 1 and 2 below, it failed to consistently operate from available mobile power sources such as being plugged into a laptop or battery supply. At the time of testing, there was no available power source that could drive the RPi without needing to be stationary. While this point is not a requirement in the project, it severely hinders the capacity of testing the RPi as it interferes with the mobility of the user testing it. Since we are bound to the location of a power outlet and the length of a substantial power cord. Driving the RPi from the batteries available and from various laptops did not produce the required voltage and amperage to drive the RPi without failure as seen in Image 3 it fails to reach the goal of 5V 3A. All this means though is that further testing will be required where a wall outlet is available, and the RPi can still maintain appropriate field of view on the garage setting.

*Raspberry Pi Configuration Script Testing*

Despite not being listed in the Alpha Test plan, it was necessary to develop and test a configuration script to ensure that it can set up a fresh Raspberry Pi (yummy) so that the system can be scaled with additional devices to ensure accurate inflow and outflow counts from a parking garage, configuration settings are documented, and that test devices that would need to be reformatted can be reconfigured quickly. The configuration script was tested by reinstalling Raspberry Pi OS and running the script repeatedly to search for bugs or errors in the user control flow, testing all user flow branches for expected output. The maximum execution time for the configuration script on a fresh Raspberry Pi 5 is within the range of 15-20 minutes.

*Raspberry Pi WiFi Antenna Testing*

To ensure a reliable and stable connection to the University of Florida network, many possible solutions were investigated during the Alpha Build sprint to remedy a WiFi dead zone within our pilot garage for testing the Release Candidate. In addition to developing an ESP32 repeater during the Beta Build, it was deemed useful to investigate utilizing a USB WiFi modem with an antenna on the Raspberry Pi devices, as opposed to the on-board Wi-Fi modem, to maximize signal strength and stability. Wireless connection performance metrics were measured for both the on-board WiFi modem of the Raspberry Pi, as well as a USB WiFi modem with an antenna, at various locations using the following Linux command sequence,

iwconfig wlan0; iw dev wlan0 link; iw dev wlan0 info

and the resulting statistics were summarized in Table 2.

TABLE 2. Modem Performance Statistics

|  |  |  |  |
| --- | --- | --- | --- |
| **Modem Type** | **AvgBitRate (Mb/s)** | **AvgLinkQuality** | **AvgReceivedSignalStrength (dBm)** |
| **On-board (5 GHz)** | 54.53333 | 79% | -55 |
| **Antenna (5 GHz)** | 115.175 | 91% | -46.3333 |

Overall, the results demonstrate a better signal by demonstrating a higher average link quality and higher received signal strength using the USB WiFi modem, which is what we were hoping to achieve. However, a more robust test plan with more data is needed to support these results, which will likely be employed in tests with the ESP32 repeater. Regardless, the USB WiFi modem allows our team more flexibility in designing the housing, as the on-board WiFi chip may encounter signal attenuation due to the enclosure around the Pi or due to noise by other hardware components, whereas an antenna can be positioned outside of the housing.

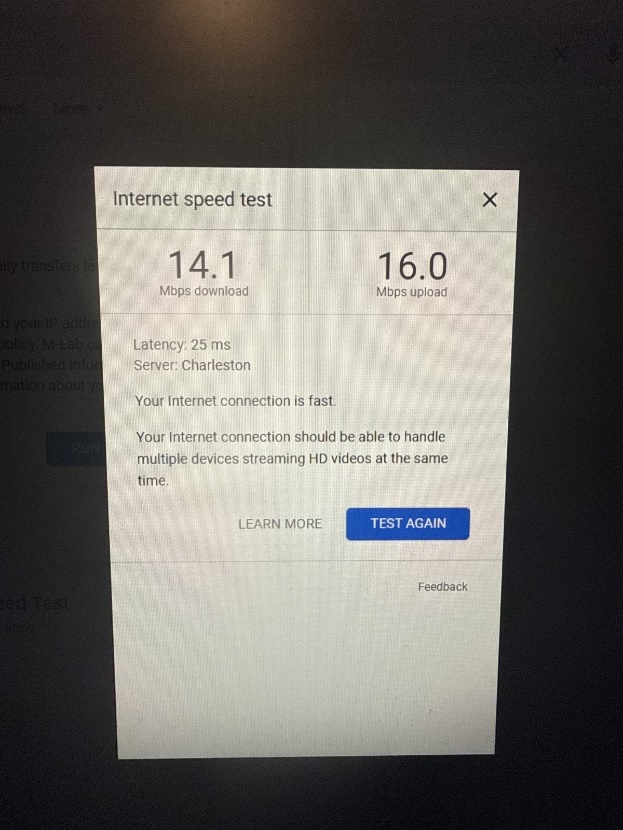
*ESP32*

The connection to the ESP32 access point is secure, and the download/upload speeds of the access point have proven to be sufficient. The security of the ESP32 access point was tested, and this was successful, as the ESP32 limited the number of clients on the network. During this test, the ESP32 remained stable although there were multiple requests being made to join the network. Further, we have now connected the Raspberry Pi to the network, and we have confirmed that the Raspberry Pi can utilize the ESP32’s network to perform the post-processing.

*Beta Testing Results*

*ESP32 and Connection Metrics*

Initially, there was a lot of trouble in ensuring the RPi had a stable connection between the RPi5 and the UF Wi-Fi near the Reitz union garage. Our plan was to extend the range of this network by setting up an ESP32 as a hotspot at the midpoint between these locations and connecting the RPi5 to it. While the RPi5 has not been tested, the test was conducted between the ESP32 and our computer, effectively emulating a very similar scenario and environment that the RPi5 will be in. The test was conducted to simulate the environment the RPi5 and ESP32 would be in, enclosed within a building and approximately 50-100 feet apart from one another. As seen in the figure below, the connection established between the laptop and the ESP32 proves more than sufficient in providing the desired stable source.

  
**Figure 4.** Wi-Fi speed performance from ESP32 host

*Raspberry Pi Database Communication*

The RPi was expected to transmit data, such as updating the occupancy value for a specific lot, in real-time without any flaw. Tests ran involved configuring two RPi’s, one as the exit and one as the entrance RPi. We then simulated cars entering and exiting the facility all the while monitoring the database data to ensure that the occupancy value reflected whether or not a car was exiting or leaving the facility. As you can see in the figures below, the database, viewed using the Postman API, accurately reflects the data the RPi is sending. This proves that not only is the RPi communicating with the database in real-time, but it is accurately transmitting the data captured in its environment, also proving that the model onboard is function sufficiently.

**Add images from postman here (cant find any on my device maybe test in garage 03/27/2025)**

*Mobile App (Alpha Plan, Technical Debt)*

Maybe talk about the mobile app receiving the data appropriately now

*Raspberry Pi License Plate Recognition (Alpha Plan Continuation)*

*Hardware/ 3D Prints*

**Status**

*Completed Tasks*

*In Progress*

Raspberry Pi License Plate Recognition

The RPi5 still struggles to capture images at the desired frame rate falling just short of 5 FPS whereas our goal is 10 FPS. Speculation results in a few possible causes for this. One of them being that the images being captured for the model are too large, thus requiring the model to downsize the image which might not be efficient. A solution being tested is to have the camera configured to capture images appropriate to the native size the model is expecting. Another possible cause is that the model is too large, i.e. it is performing more tasks than it should resulting in a “beefy” model taking more time than is realistically necessary. A solution to this would be to downsize the model to make it less comprehensive but still perfectly capable of achieving the desired performance. Further testing is required to ensure there is not a large performance drop-off when the model is downgraded.

Raspberry Pi Image Transfer and Image Fidelity

The RPi5 is still struggling to consistently capture images of vehicle traffic passing through in high image fidelity. If traffic is moving to rapidly, or the RPi5 is not fixed at the appropriate angle, resulting images captured prove blurry producing images that not even the human eye can make out the license plate characters. Currently, the RPi5 is being tested with a higher shutter speed to attempt to reduce the probability of heavily blurred images.

Because of this, progress into image transfer to the model hosted on AWS has been limited as we cannot ensure that RPi5 is producing an image worth sending to that model. Regardless, testing has shown that it is difficult to transfer and rebuild the images received by the RPi and RPi performance drops significantly during image transfer effectively quadrupling the size of the RPi script just to transfer the image.

*Planned Tasks*

**Conclusion**

**References**

[1]L. Du and S. Washburn, “SMART PARKING SYSTEM ON UF CAMPUS,” Apr. 2019. Available: <https://fora.aa.ufl.edu/docs/38/2018-2019/SmartParkingProposalLiliDuScottWashburn.pdf>