Parking Availability System

Design Prototype Report

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

Evidence of Soundness

Theoretical Background

Currently, there is some trouble in running the YoloV11 LPR object detection model on the Raspberry Pi Model 3 B+ (RPi). This is crucial to the overall success of this project as running the model on the RPi will serve as our “eyes and ears” for the project. The model is currently working, as seen in figure 1, however, there is some trouble loading it and installing it on the RPi. Our group is confident that this can be done given that many others have gotten similar models to work on the same model as our RPi. You can reference the link here, to refer to the same tutorial that our group has initially followed to configure the RPi for usage with the YoloV11 model. Currently, the model is being attempted to run on a docker container that contains the necessary libraries to run the model, however, the group is currently experiencing crashing when trying to just import the library required to make inferences with the mode as seen in figure 2. For further proof, you can refer to this video, which proves that the mode can in fact be run on the RPi.

Figure 1. Car object detection

```
[*]: import cv2
     # from picamera2 import Picamera2

     from ultralytics import YOLO

     # # Initialize the Picamera2
     # picam2 = Picamera2()
     # picam2.preview_configuration.main.size = (1280, 720)
     # picam2.preview_configuration.main.format = "RGB888"
     # picam2.preview_configuration.align()
     # picam2.configure("preview")
     # picam2.start()

     Creating new Ultralytics Settings v0.0.6 file ✅
     View Ultralytics Settings with 'yolo settings' or at '/root/.config/Ultralytics/settings.json'
     Update Settings with 'yolo settings key=value', i.e. 'yolo settings runs_dir=path/to/dir'. For help see https://docs.ultralytics.com/quickstart/#ultralytic
     s-settings.
```

Figure 2. Failed/Freezing model load

Domain & Prior Art

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 an accessible view of parking facility capacities and lacks availability across the campus parking infrastructure. This omission inconveniences drivers searching for available parking when none exists. Additionally, 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 require equipment and installation fees. Our project will also provide virtual observability into the current availability of a parking facility to streamline an individual's search into finding a parking spot.

Empirical Evidence

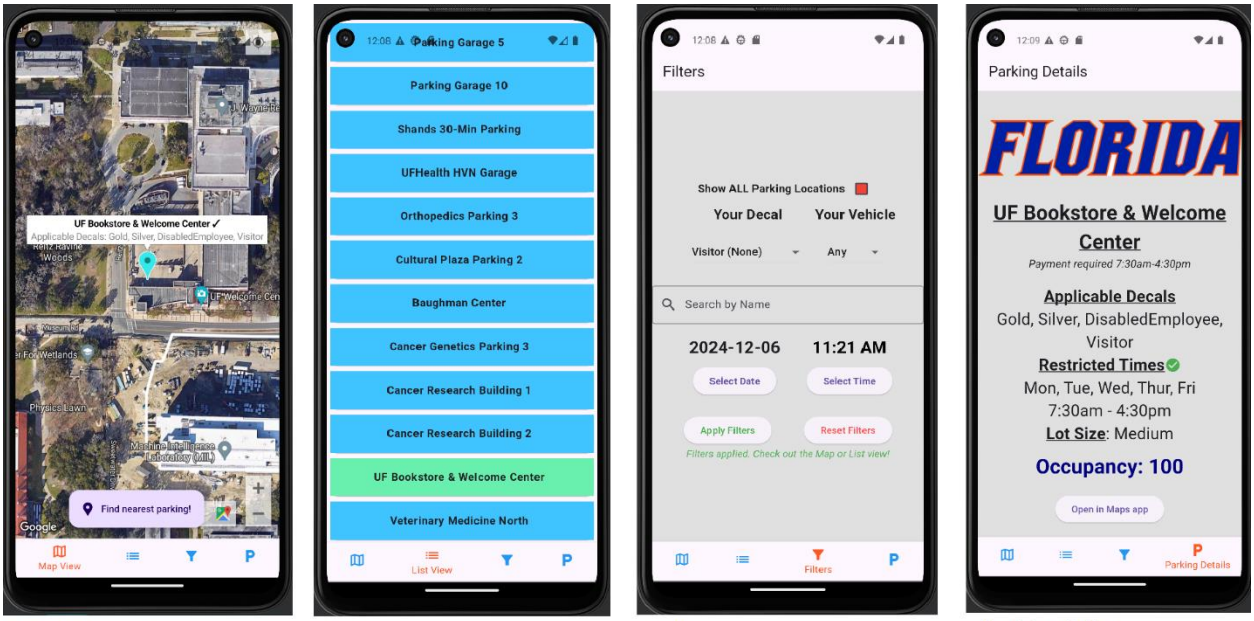
Throughout the development process of the hardware module, one aspect of the hardware design that was a cause for concern was powering the RPi device(s): should the device have a hardwired connection to a seemingly infinite power source or be powered by batteries? For the prototype, we decided to pursue the battery approach, as it kept the device easily portable to test with, did not require special hardware, and averted a key concern, which was finding outlets in a parking garage when we needed to test. The design for the first implementation of the hardware was created with battery-power in mind. Powering the RPi in the lab for multiple days and switching batteries when it would lose power, we determined that, at little load, the RPi could remain on with a single 10,000 mAh power bank for slightly more than 16 hours in practice. This being the best-case scenario, we concluded that even with the RPi in a "sleep" mode, only triggering a recording and processing of a feed through the object detection model on a trigger, this number would be hard to achieve. Moreover, since the system would need to be scaled to multiple units to provide full coverage of a parking facility, this would require replacing and charging a lot of batteries. Hence, for our next iteration of the hardware design, we will try to design a solution that would need a hard-wired power connection and choose a pilot garage that has available outlets for testing.

External Interface

Presentation

Mobile App

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.



Map/Home Screen | Parking List | Filter | Parking Info

Figure 3. Parking App Frames

Perception

Mobile App

Within the mobile app, users are presented with a ribbon bar to navigate between four page states. The associated page visible on the screen is highlighted within the ribbon bar in orange, whereas the rest are blue, indicating the pages can be selected or moved through. Selecting another page is associated with a change in the page state. Additionally, within the mobile app, users can navigate the map intuitively with touch controls akin to many similar navigation apps. Parking locations can be selected by clicking on the location within the map view or list view, which centers the map in the map view onto that location. Additionally, selected parking locations are highlighted in map view with a cyan pin and with an information pop-up on top, or in the list view with a green background on the parking location's card in the list.

Sensors

The way the hardware module will interact with its environment is by defaulting to a "sleep" mode until a motion detection sensor on the RPi detects movement. A persistently running loop polls for the sensor value, and if it receives the appropriate value, the RPi camera will turn on and the video feed begins, capturing image frames of the environment following the sensor trigger, feeding the vehicle detection model.

Usability

Mobile App

In the mobile application, users should always have access to the most up to date information according to what is stored in the LOT table. The table is polled at the start of the app, and periodically for a specific location when that location is selected. Under conditions where the table could not be connected to, parking location information is also hard-coded into objects in the codebase, which allows users to always have access to some form of parking information, even if it is not up-to-date.

Internal Systems

Component Architecture

AWS Backend

The cloud backend is hosted on AWS and implements two important components of the Parking Availability System, an object detection and license plate recognition model as well as the relational databases used to access lot and car information. As seen in Figures 4 and 5, a PostgreSQL database is hosted using AWS Relational Database Service, implementing two tables — CARS and LOTS — which tracks information, like license plate and color, for the cars parked in specific lots on campus detected by the system and information related to parking facilities on campus, respectively.

```
burrow=# TABLE cars;
                carid                 |  plate   |   color    |                lotid
--------------------------------------+----------+------------+--------------------------------------
 904f7f20-28f9-495e-afaa-f4d8bda1fd99 | JLF-1304 |            | f24dacb0-9513-45d9-90f8-b07e5dee5271
 1d8acb5e-1944-407f-ba24-0a3615feaf2e | T719257C |            | f24dacb0-9513-45d9-90f8-b07e5dee5271
 3a452865-6e30-46b0-83e9-c495556518c7 | 5K15     |            | f24dacb0-9513-45d9-90f8-b07e5dee5271
 5695e925-8527-4a22-964c-5adbe8aa8e44 | KNOJBL   |            | f24dacb0-9513-45d9-90f8-b07e5dee5271
 ae6f74b0-7647-4c89-b875-d40ef941fb68 | KN3 JBL  |            | f24dacb0-9513-45d9-90f8-b07e5dee5271
 6d62375a-be5c-4b05-ac75-4b31e4d8f9cc | "NMTE84" |            | f24dacb0-9513-45d9-90f8-b07e5dee5271
 24028284-8201-4c49-87b1-6b81d54b18c5 | TEST-CAR | TEST-COLOR | f24dacb0-9513-45d9-90f8-b07e5dee5271
 8e049944-94e5-4d87-8d70-6e8f84a2c768 | ADJ JBL  |            | f24dacb0-9513-45d9-90f8-b07e5dee5271
 07ba9230-09bc-48e1-933d-1f3156bdb901 | EVT R34  |            | f24dacb0-9513-45d9-90f8-b07e5dee5271
(9 rows)
```

Figure 4. PostgreSQL CARS Table

lotid [UUID]	latitude [NUMERIC]	longitude [NUMERIC]	name [VARCHAR]	address [VARCHAR]	open [TIME]	close [TIME]	days [_VARCHAR]	decals [_VARCHAR]
0210feeb-cf1d-42a7-95(	29.647502	-82.359249	Fraternity Row 2		08:30:00	15:30:00	{M,T,W,R,F}	{parkAndRide,red1,disabl
0226350f-5dc0-433e-a(	0.407479	0.345672	test name	5678 Elm St	12:35:22	12:35:22	{M,T}	{Red,Green}
02bf2c6a-1810-4fe2-90	29.645655	-82.337330	Parking Garage 8		08:30:00	15:30:00	{M,T,W,R,F}	{orange,red1,red3,disable
0405bdde-778a-4de7-9	29.639044	-82.346679	Parking Garage 2		07:30:00	17:30:00	{M,T,W,R,F}	{gold,silver,visitor,disable
060dc221-bd70-408a-a(	29.649438	-82.339431	Tigert Hall/13th Street M		07:30:00	16:30:00	{M,T,W,R,F}	{motorcycleScooter}
06fa4bf0-3ecb-4ef7-a0l	29.645198	-82.348377	UF Bookstore & Welcome		07:30:00	16:30:00	{M,T,W,R,F}	{gold,silver,disabledEmpl
09549251-14f3-44ab-9:	29.645754	-82.352848	Flavet Field		08:30:00	15:30:00	{M,T,W,R,F}	{green}
1114af86-d158-4b14-b4	29.640779	-82.341599	Parking Garage 10		08:30:00	15:30:00	{M,T,W,R,F}	{visitor,motorcycleScoot
11945155-ac9e-45c7-86	29.647512	-82.342973	Inner Road Motorcycle		07:30:00	16:30:00	{M,T,W,R,F}	{motorcycleScooter}
1284e0bd-fe3f-4156-84	0.083723	0.871200	test name	1234 Main St	12:38:42	12:38:42	{M,T}	{Red,Green}
128ce783-ff24-4a47-be	29.639699	-82.356552	IFAS Parking 3		08:30:00	15:30:00	{M,T,W,R,F}	{orange,motorcycleScoo
1429020c-9ba3-4662-a	29.646837	-82.350160	East Hall		08:30:00	15:30:00	{M,T,W,R,F}	{orange,motorcycleScoo
18a4bda1-b294-47a9-af	29.634154	-82.351701	Veterinary Medicine Wes		08:30:00	15:30:00	{M,T,W,R,F}	{parkAndRide,green}
18a9b8f3-7f9c-41bc-ac	29.637907	-82.367623	Southwest Rec Center		08:30:00	15:30:00	{M,T,W,R,F}	{parkAndRide,disabledSt

Figure 5. PostgreSQL LOTS table

The information in the LOTS table is used to track real-time parking availability, being updated by the RPi hardware or read from by users' mobile applications. Both the CARS and LOTS tables are interacted with through a RESTful API written in Golang hosted on an AWS EC2

instance, implementing 10 REST endpoints to perform CRUD (Create, Read, Update, Delete) operations on both tables through HTTP requests from the other system components.

Object Detection and License Plate Recognition

Object detection occurs both within the RPi and within the cloud. Object detection is implemented with Ultralytics's YOLOv11 nano model on the Pi to detect license plates within a scene and send frames with a detected license plate to the Optical Character Recognition (OCR) pipeline in the cloud. The OCR pipeline runs another YOLO model to verify the existence of the car and its license plate within the frame, sends that frame to be processed with OpenAI's GPT-4 to interpret the license plate, and saves the car information to the database using the Save Car API endpoint.

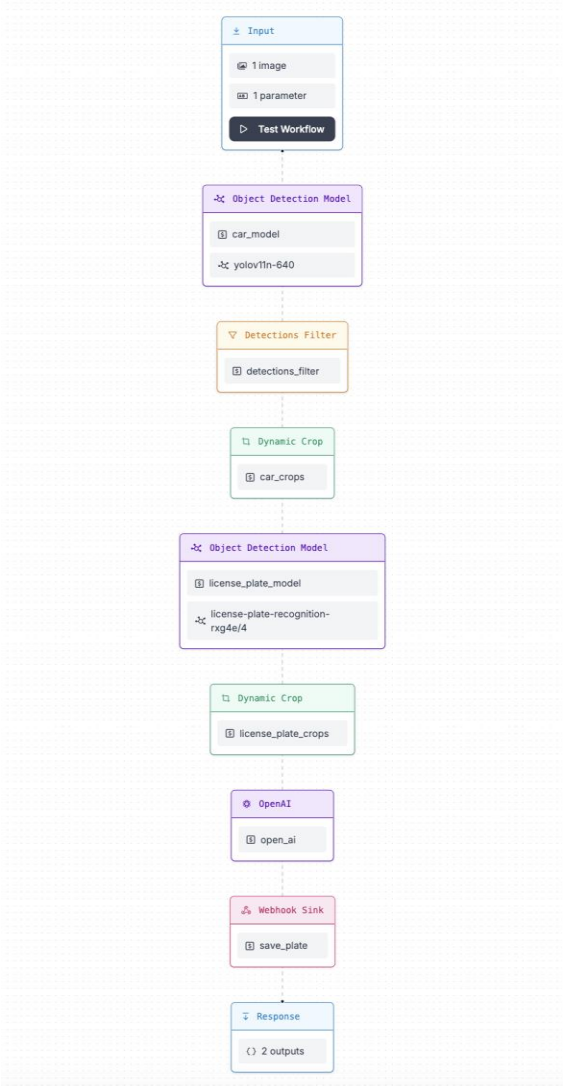


Figure 6. OCR Pipeline

Mobile App

With permission from the original developer, Drew Gill, a UF Computer Science alum, the mobile application component and the real-time parking information feature were built on top of his codebase. The app is built using the Flutter framework using the Dart programming language, which allows applications to be developed using a single codebase. The Google Maps API is used to display the map and location pins within the app. Parking location data is both hard coded into the code and read from the database, though information read from the database is prioritized. Upon starting the app, hard-coded data is overwritten with information from the database. When a parking location is selected and the information page is visible, the app polls the database periodically, ensuring that the user has the most up-to-date information.

Raspberry Pi

The Raspberry Pi (RPi) is configured to a sensor and camera to capture a feed of the environment to perform object detection. Until the sensor is triggered the RPi does not capture a feed and remains in a sleep state. When triggered, the RPi starts capturing a feed from the camera. Once the RPi is done capturing the feed, the images are interpreted, and the PostgreSQL database is updated accordingly. The update method functionality can be seen in Figure 6, where the RPi is able to query for all of the cars in the database. In the future, once this query is complete, the RPi will determine if the license plate already exists in the database. If so, this will mean the car is leaving the garage. If not, then the car is entering the garage. If an entity is determined to have entered the parking facility, the database is updated with the license plate value, and the lot occupancy field is incremented. If a vehicle is determined to have left, the license plate stored in the database is removed and the lot occupancy field is decreased. Finally, once this process has completed, the RPi will return to its “sleep” state and wait until the motion detection device again detects movement. The product flow diagram can be seen in Figure 7 on the next page.

```
    getAllCars(url)

#     app.run(host='0.0.0.0', port=5000)
#     captureFrames()


if __name__ == "__main__":
    main()
```

```
Opening secrets
[
    {
        "CarID": "df9a66c1-2589-46c1-95e1-dc830b000f76",
        "license_plate": "621-ABC",
        "color": {
            "String": "Orange",
            "Valid": true
        },
        "LotID": "c845a7d2-03c6-4dcd-a241-2db0b8905c1f"
    }
]
```

Figure 6. Querying the car database

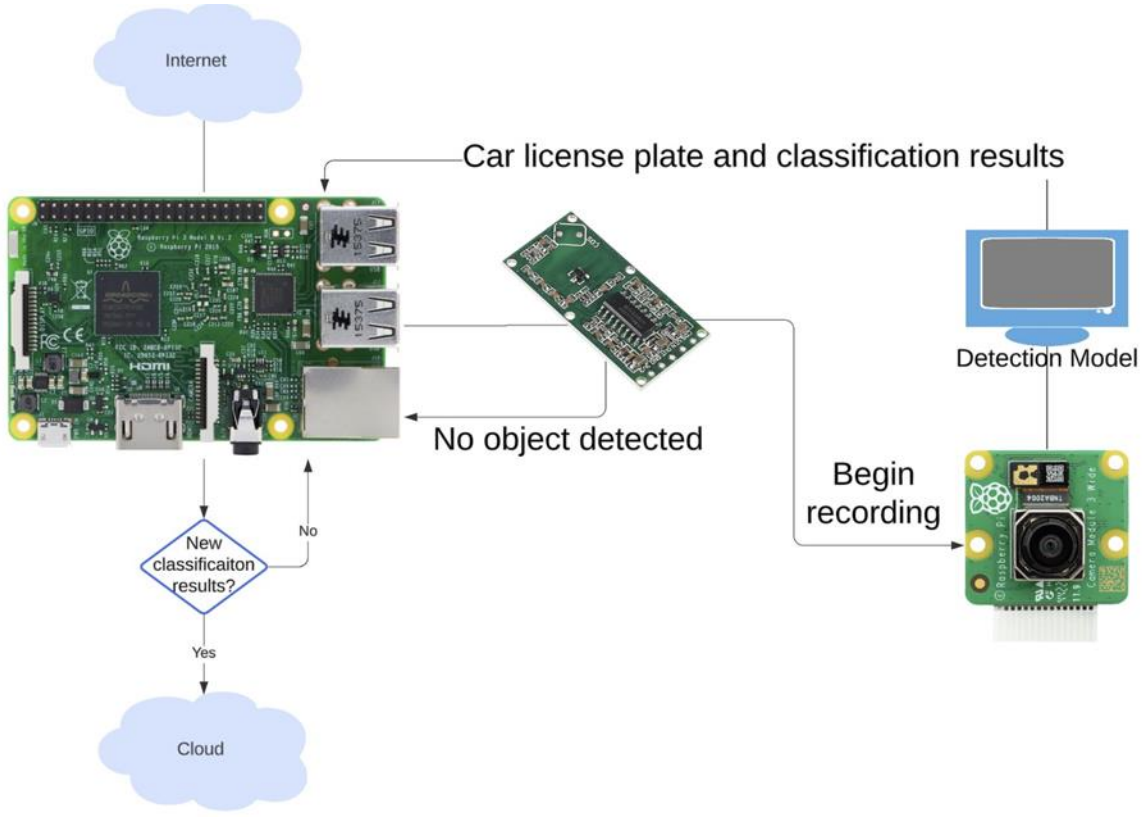


Figure 7. RPi Flow Diagram

The external interface relies on a sensor module and camera for detecting and recording vehicles entering and exiting the parking facility. When a vehicle disrupts the sensor beam, the sensor triggers the camera module to begin capturing a video feed of whatever entity disrupted it. This video feed instantiates a license plate object detection model and will observe the frames provided by the video 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. If the model successfully interprets the provided frame and identifies a license plate, the features of the license plate, such as the text, are extracted. A PostgreSQL database is updated accordingly with the interpreted/classified license plate value. It is also updated with the respective entity either leaving or exiting the facility depending on which RPi has detected and classified the entity.

Communication Mechanisms

AWS Backend

The AWS backend interacts with the other components of the system using HTTP requests from other components that interact with the RESTful API implemented in the EC2 instance. The API implements CRUD operations that allow access to the relational databases storing car and lot information.

Mobile App

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.

Raspberry Pi

Network connectivity is established via the Raspberry Pi's onboard wireless module (model 3 B+). The garage we intend to roll out the solution to, Parking Garage 4, provides sufficient network connection and served as the main supplier for the data/images that the model was trained on. This connectivity supports data transfer to the PostgreSQL database that is being hosted via AWS via communication with the API endpoints via HTTP requests, similar to the mobile app.

Resilience

Object Detection and License Plate Recognition

The object detection pipeline relies on a connection to the OCR pipeline to be established before license plate recognition can be done. Because of this, the RPi currently caches frames temporarily to be sent to the OCR pipeline in the event a connection is lost and needs to be reestablished so that data is not lost.

Mobile App

When the mobile app is first launched, it calls the `getAllLots` function to retrieve the most up-to-date parking location information from the database. However, in the case that this call is unsuccessful or times out, to prevent the user from being unable to access any lot data, ParkingLocation objects have hard-coded data in the app**.**

Raspberry Pi

The Raspberry Pi is responsible for sending frames from the camera to the database when triggered by the sensor. Since we are recording these frames dependent on the triggering of the sensor, there needs to be a conditional statement implemented within the conditional for the triggering of the sensor which ensures that a mp4 file has been properly created and exists before attempting to access it. This ensures that the file being passed to the object detection model is an accurate file and can allow for the license plate to be read.

Effort

Effort for the project is documented in many forms but is difficult to aggregate and quantify in a single report. All effort is logged across the GitHub repository, the GitHub Kanban board, the Contribution Log Excel sheet, and the meeting check-in sheet. All of these resources can be accessed in the project's GitHub repository, according to the README: https://github.com/emeurrens/parking-availability-system.

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