Parking Availability System

Alpha Build Report

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

**Usability**

**Interface**

*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.

A screenshot of a phone

Description automatically generated

**Figure 1.** Parking App Frames

**Navigation**

*Mobile App*

The mobile app currently implements all features necessary to be able to view real-time parking facility updates by reading from 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.

**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.

**Responsiveness**

*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.

**Build Quality**

**Robustness**

*Raspberry Pi*

The raspberry pi is incased in a locked 3D-printed container with holes allowing for the camera module to capture pictures of its surroundings. This makes the external interaction with its environment secure. Now, we have migrated to using a raspberry pi 5 after encountering inconsistencies and crashes when using the raspberry pi model 3 b+. There have been no crashes experienced yet, and the model runs flawlessly on content already contained within the raspberry pi.

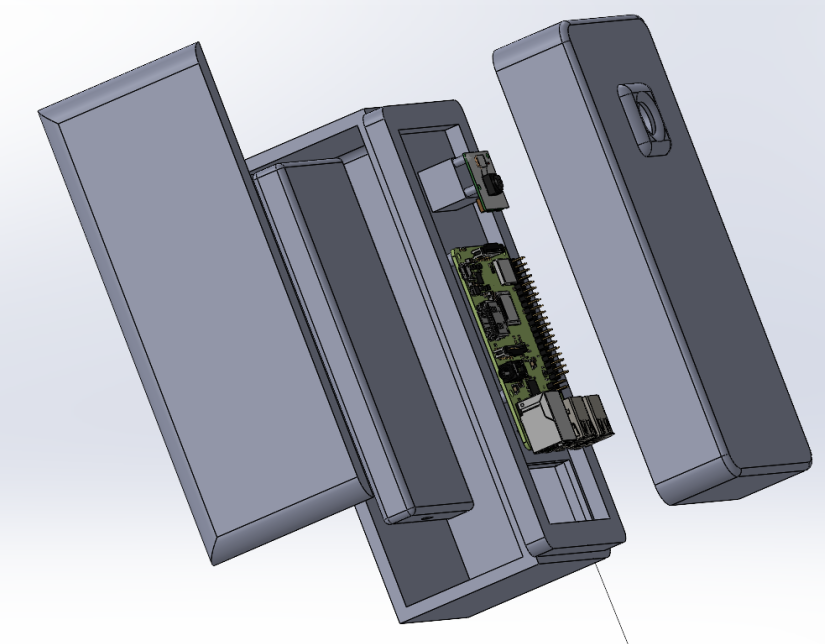
**Consistency**

*Raspberry Pi*

The raspberry pi, under current circumstances, behaves as expected. While efforts to test the pi in a non-controlled environment have been mitigated by hardware problems. Currently, with an image that has already been taken, the pi works perfectly to inference the image and determine whether a license plate is present or not. It is then already capable of updating the database assuming a plate has been detected. The current issue is that because the model is hosted in a virtual environment on the pi, accessing the camera port is posing a challenge as the virtual environment does not have system wide packages and hardware.

**Aesthetic Rigor**

For the physical design, all hardware components should be well-integrated and visually organized, ensuring that the installation appears professional and not haphazard. Any exposed wiring, enclosures, or mounting mechanisms should be arranged to maintain a structured and polished look. The components currently are a raspberry pi, camera, and a charging port. The design is going to change quite a bit because of outside variables changing but the main idea is going to stay the same. The bottom two images are what the final product should look like with a few changes due to heat testing and environmental changes that we discovered.



**Figure 2. Model of the box (Assembly View)**



**Figure 3.** Model of the physical box

**Vertical Features**

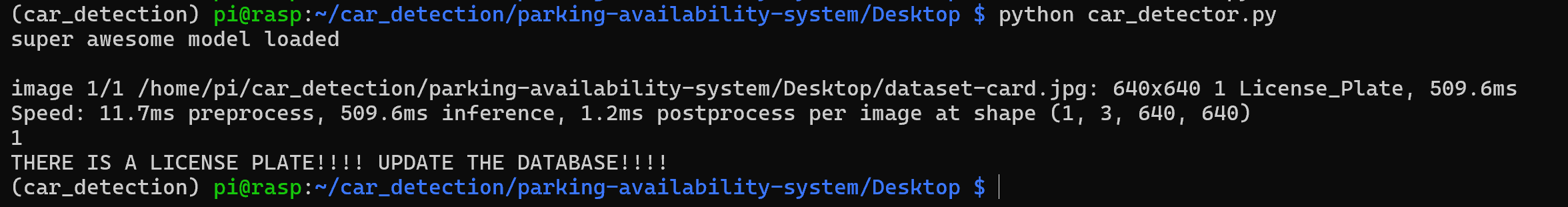
**External Interface**

*Raspberry Pi*

The Raspberry Pi (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 image is then transferred to the model residing in AWS for a more comprehensive feature extraction. The license plate values are then recorded and added to or subtracted from the database. The update method functionality can be seen in Figure 4, where the RPi is able to query all of the cars in the database. In figures 5 and 6, we can observe an example output of when the model detects a car and the resulting image the model has produced. Figure 7 demonstrates the result of when a car is not detected. 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 enters 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.



**Figure 4.** Querying the car database

**Figure 5.** Object detection mode detects license plate

**Figure 6.** Model bounding box results



**Figure 7.** RPi Flow Diagram

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. 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.

**Persistent State**

*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 the model confirms that a car is present. We are capturing these image frames constantly, but the project is dependent on the model detecting whether a car license plate is present or not. Assuming a plate is detected only then do we update the database and pass this image to AWS for further inference. This ensures that the file being passed to the object detection model contains a car license plate and can allow for the license plate to be read.

*ESP32*

The Wi-Fi connection in the Reitz parking garage does not extend to the entrance of the garage. However, there is a corner of the garage which has a Wi-Fi connection. Therefore, we are utilizing an ESP32 as an access point. This has been done by using the Wi-Fi library built into the ESP32. The ESP32 has been configured with an ssid and a password, as well as a max connection which defines the maximum simultaneous connected clients to the access point. The ESP32 temporarily stores data packets temporarily and then sends them out to the clients. The Raspberry Pi can connect to this network rather than attempting to connect to the UF network directly.

**Internal Systems**

*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 6 and 7, 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.

A screenshot of a computer

Description automatically generated

**Figure 8.** PostgreSQL CARS Table

A screenshot of a black and white screen

Description automatically generated

**Figure 9.** PostgreSQLLOTS 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.

A screenshot of a computer

Description automatically generated

**Figure 8.** 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.

**Effort**

Effort for the project is documented in many forms, but it 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>.

