Parking Availability System

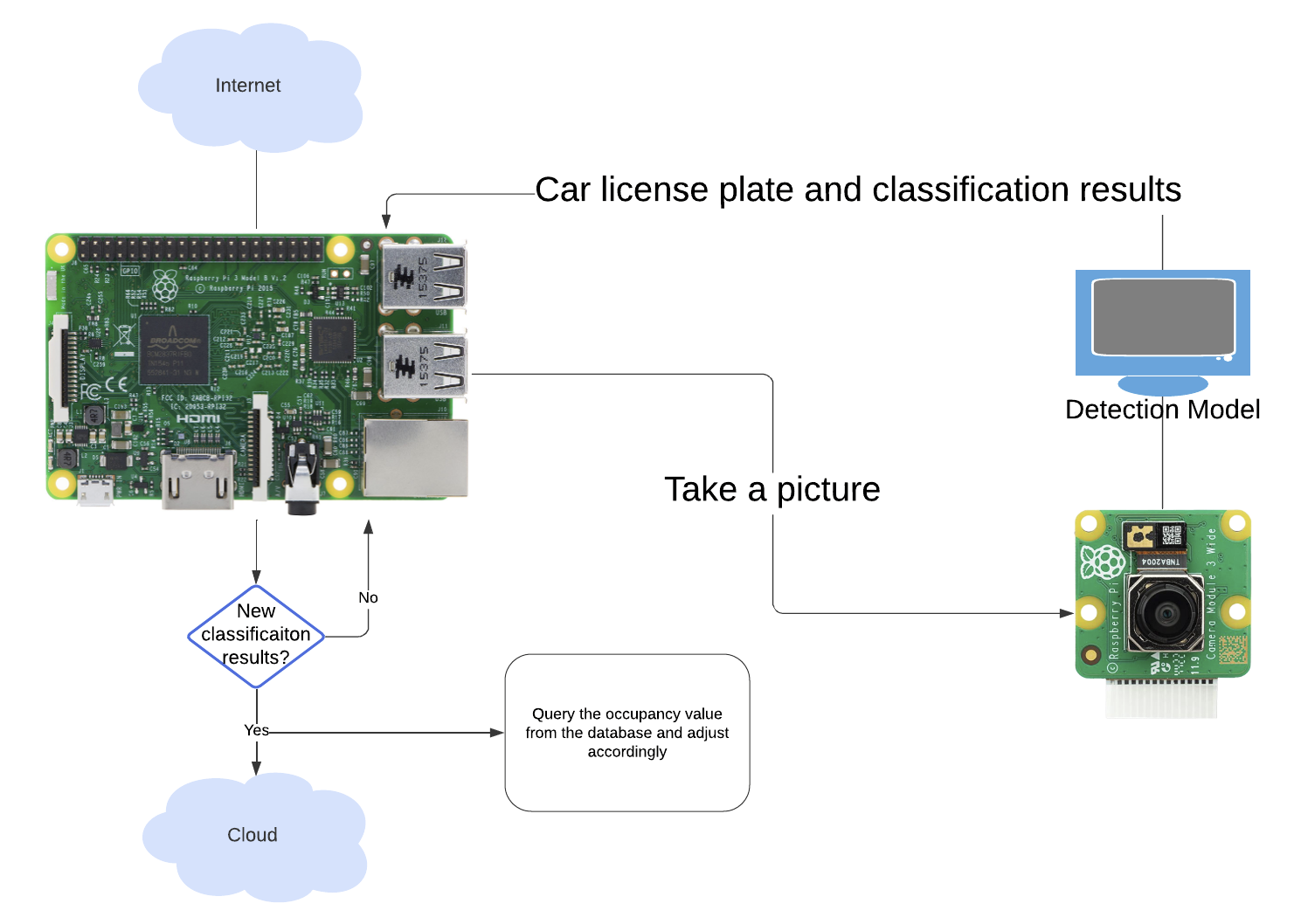
Alpha Test Plan

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

**Expected Behavior**

*Raspberry Pi*

The Raspberry Pi (RPi) is meant to be set up in a garage at the appropriate angle to capture cars entering or exiting the parking facility. As cars enter/exit, a camera module is expected to capture images at the appropriate rate to maintain a real-time, accurate count of the availability within a parking facility. These images are then fed to a lightweight preliminary model hosted on the RPi and inferred. The inferences are then extracted and if it yields that a car license plate is detected the image will be forwarded to AWS for further information extraction. Due to a lack of WiFi coverage at the entrance of our pilot garage, the Reitz Union garage, the RPi will connect to the UF network through an ESP32 repeater. The RPi itself though will be able to query the database for a current count of vehicle occupancy in the facility and increment it if the vehicle is entering or decrement it if the vehicle is exiting. A flow chart of this can be seen in Figure 1. The source code for the current implementation can be seen [here](https://github.com/emeurrens/parking-availability-system/blob/rpi_files/Desktop/car_detector.py).

**Figure 1.** Raspberry Pi flow chart

****

**Figure 2.** Wireless Communication Flow Diagram

*License Plate Recognition Model*

License plate detection and recognition models will run on both the RPi and the AWS backend. When a car enters or exits, the RPi detects the presence of a license plate within the image frame as the car passes. The RPi will then send this image frame to the AWS backend, which will perform license plate recognition to identify the car that has entered or exited.

*Mobile App*

The mobile app follows a particular user flow and data flow, as seen in Figures 4 and 5. Users would open the app and be introduced to the map screen. As the app starts up, the app loads ParkingLocation objects and initializes them with hard-coded data. Following the instantiation of these objects, the app attempts to query the database for up-to-date data within the database and update the existing ParkingLocation objects. If the query failed, parking information will still be available in the app, albeit not up-to-date, and real-time availability will not be available. If successful, a ParkingLocation’s corresponding information screen will have the most up-to-date information. If the initial query of the database timed out, a user can still access up-to-date information later by navigating to a particular ParkingLocation object’s information screen, which queries the database periodically for updated information on that particular location. This ensures that users will still have access to the most up-to-date, real-time information, supposing nothing is wrong with the database connection.

A diagram of a parking system

AI-generated content may be incorrect.

**Figure 3.** Mobile App User Flow Diagram

A diagram of a software process

AI-generated content may be incorrect.

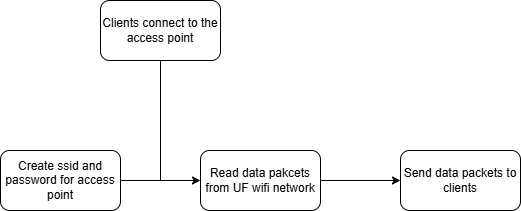
**Figure 4.** ParkingLocation Data Update Diagram

*Backend API*

****

**Figure 5.** Lot Database API Interaction

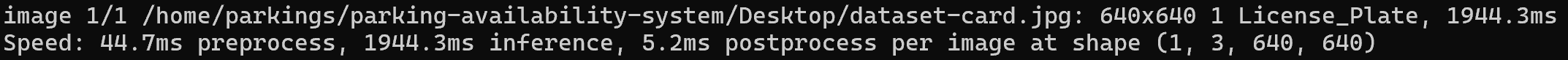
*ESP32*



**Testing Procedures**

*Raspberry Pi*

How we have currently tested this is with images found on the internet of cars with and without license plates, as well as random images. We fed them into the model randomly and recorded its output to see if it correctly determined whether a plate was present or not. How we plan to the RPi’s efficiency at detecting cars from the images it takes is to maneuver the RPi manually in a parking lot while it is taking images and producing results. We will also set it up in a parking lot in a similar position to the one that would be in the garage and drive our cars by it to see if it correctly inferences those images. Additionally, to ensure that the images are taken fast enough to provide the model with an accurate representation of its environment we are going to try to optimize the inference time value. This can be seen in Figure 6 where the model report provides us with the time it took to determine whether a plate was present. We will use this metric to determine whether our system is operating within our desired time constraints.

**Figure 6.** Model inference time

*License Plate Recognition*

The testing procedure involves both testing the model on existing license plate datasets and collected image frames from the field and evaluating performance metrics of the model such as accuracy and confidence. Additionally, while testing the model, we will monitor the inference speed and adjust configurations and parameters to try to improve performance. Moreover, because a license plate is a rectangle with text in it, we want to explore potential flaws in the model by testing the model with figures that appear like license plates, like a sign, or a drawn license plate, or license plates that are not on a car, like a person holding a license plate. Since the model was tested on images of cars with license plates, we expect the model to perform well, as our assumption is that the model was trained on the features of cars as well as license plates.

*Backend API*

The backend API undergoes automated testing using GitHub Actions, which executes a suite of unit tests to verify the correct operation of both internal API functions and external endpoint functions. These automated tests help ensure the correctness of the backend.

The unit tests for the internal API functions validate the following core operations:

**Cars**

* **Saving a Car** to the Database
* **Getting a Car** from the Database
* **Updating a Car** in the Database
* **Getting all Cars** for a provided *lot\_id*
* **Deleting a Car** from the Database

**Lots**

* **Saving a Lot** to the Database
* **Getting a Lot** from the Database
* **Updating a Lot** in the Database
* **Getting all Lots** in the Database
* **Deleting a Lot** from the Database

Additionally, the unit test for the external endpoint functions tests the following endpoints:

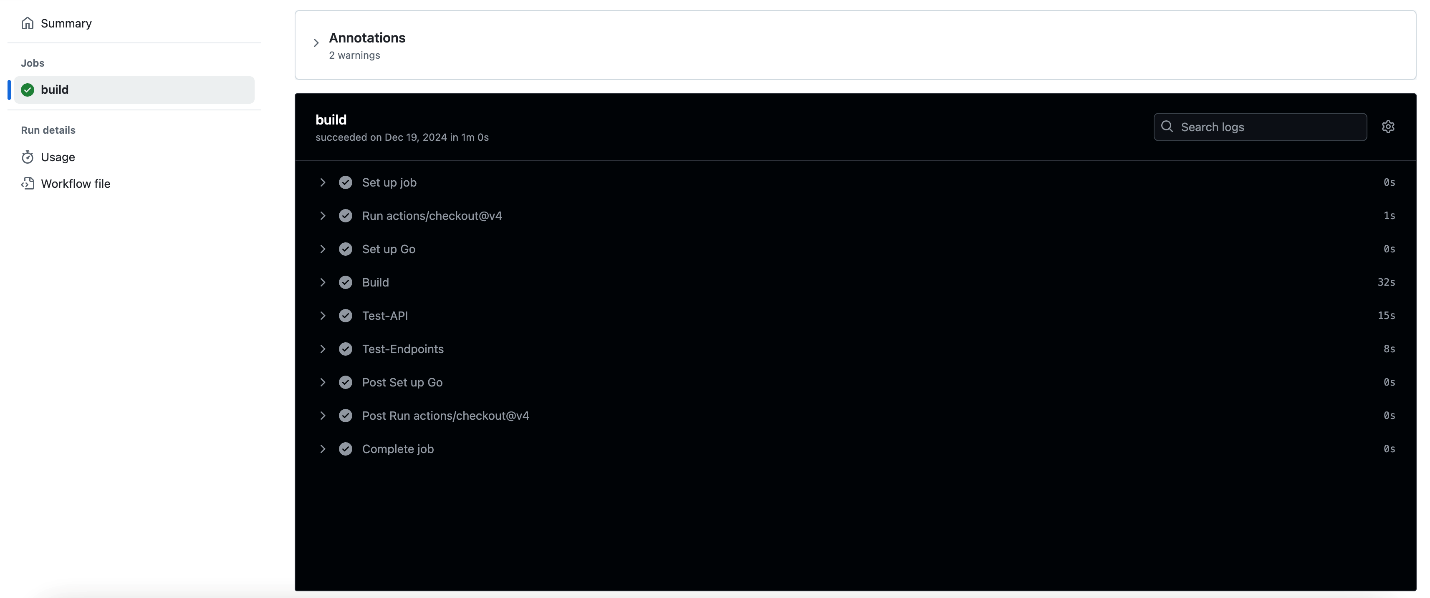
**Cars**

* SaveCar
* GetCar
* UpdateCar
* GetAllCars
* DeleteCar

**Lots**

* SaveLot
* GetLot
* UpdateLot
* GetAllLots
* DeleteLots

These unit tests are automatically triggered via GitHub Actions upon code commits or pull requests to ensure the correctness of our code changes to enable continuous integration and prevent regressions.

**Figure 3.** GitHub actions workflow

*Mobile App*

The mobile app will be tested by following the ParkingLocations’ data states as the app runs and reaches points in the data flow diagram and user flow diagram that should result in changes in the data. This will be done by using the debugging tools within Android Studio as the program executes. Front-end features will be tested by acting as the user and navigating the user flow diagram, looking for bugs.

*Hardware (3D Prints)*

The physical enclosure and mounting structures for the system will be designed and fabricated using 3D printing, ensuring a precise fit. Due to climate conditions, particularly heat buildup, 3D-printed enclosures must be designed with ventilation in mind. Electronic components generate heat during operation, and without proper airflow, this could lead to thermal issues, reduced efficiency, or even hardware failure over time. To mitigate this, the design will incorporate: Ventilation holes or slotted openings in non-critical areas to allow passive airflow. Strategic placement of openings to prevent dust or water intrusion while still dissipating heat. Since last semester, significant design changes have been implemented in the 3D-printed components of the parking availability system due to a major hardware upgrade: the shift from battery power to a direct power source. This transition brings new opportunities for efficiency but also requires adjustments in the physical design to accommodate the updated power infrastructure.

*ESP32*

The ESP32 access point will need to be tested to ensure that the connection will remain stable for long periods of time. Further, we need to ensure that the connection is secure, and that only the raspberry pi can connect to the access point. Multiple measures will be implemented to ensure that the ESP32 access point is secure and stable. The number of connections and the security of the ESP32 can be tested by overloading the ESP32 with connection requests from other devices, such as a mobile phone. This test would ensure that only a certain number of clients can connect to the network, as well as ensuring that the network will remain stable while receiving many requests.