Boston University
Electrical & Computer Engineering
EC463 Senior Design Project

First Semester Report

WhereTo

Submitted to
Andreas Papadakis

by
Team 5

Team Members:
John Burke - jwburke@bu.edu
Muhammad Ahmad Ghani - mghani@bu.edu
Haochen Sun - tomsunhc@bu.edu
Erick Tomona - erickshi@bu.edu
Marta Velilla - martava@bu.edu

Table of Contents

Executive Summary

WhereTo

Team Number – Team 5

Introduction

In today's urban and suburban environments, it can oftentimes be stressful and confusing to navigate local parting availability and restrictions. Oftentimes there are local regulations that prohibit parking on certain streets at specific times, specific days, or always. The rules and regulations can vary wildly, even across a single city. Today's drivers would benefit from a solution that provides them up-to-date, real-time parking information.

WhereTo will be developed as a useful tool for these drivers to ease their navigational burden when it comes to parking. As an intuitive, real-time mobile application, WhereTo will connect users with the information that they want regarding parking in a given area.

We will approach implementing this application with a focus in Artificial Intelligence and integrating with existing APIs. At a high level, we aim to calculate the parking availability of a given segment of a street by querying the street from multiple coordinate points and angles using Google Street View and then feeding these images into a model that can determine parking regulations given images to analyze.

This system will solve the problem that drivers of today face, particularly as this solution is very generalizable. The use of image data from Google allows us to have data for streets all across the United States. The use of an AI model to read and detect parking rules in regulations also lends itself towards being a generalizable solution. As our solution will not include any local data sources, it is not limited to being used in just one area. Both the integration of Google APIs into our design as well as the use of AI to determine parking regulations are unique features which highlight the scalability of our design.

The user will be able to enter an address and a radius into the application frontend. Upon clicking search with these two parameters filled in, the frontend will send the address and the radius information to the backend and its AI model. The parking rules will then be evaluated for the region that they want to see and the rules will be visually presented back in a visual map form to the user. The user will then be able to click individual street segments on the map to understand more about the parking regulations of that stretch of road as well as the reason that the model believes those to be the regulations. This reason will likely be in the form of a cropped image that the model used in determining the regulations for that segment.

The final deliverable for solving this problem is a fully integrated end to end system comprising both this frontend and backend; both code for the frontend and the backend will need to be delivered. The frontend will be written using React Native. The backend will be created using Flask for Python to create an API. This Python API will be responsible for calling the various Google APIs and other APIs we will be using in the system. The Python API will be what the React Native UI calls in order to perform parking analysis. These deliverables will require us to design algorithms to efficiently parse geographical data as well as data from various Google APIs. We will likely implement a street segment caching system to avoid heavy redundant calls to our AI model.

The nature of solving the problem of locating and understanding parking and parking regulations, especially programmatically, is not an easy task. This is precisely why the solution to the problem itself requires complex algorithm design and integration with large data sources in conjunction with AI. The final design should exhibit efficient algorithmic design to ensure we do not waste compute time or resources in our implementation.

Concept Development:

Urban parking poses a significant challenge in today's fast-paced world, especially in densely populated cities. Drivers often spend a considerable amount of time searching for available parking spots, only to be confronted with complex and frequently changing parking regulations. This not only leads to frustration but also contributes to increased traffic congestion, environmental pollution due to prolonged vehicle running, and a general decline in the quality of urban life. The customer's problem, therefore, is not just finding a parking spot but finding one efficiently, in compliance with local regulations, and with minimal hassle.

Our team has begun working on WhereTo in order to address this pervasive urban challenge. Our goal is to develop a solution that simplifies the process of locating parking spaces, making urban driving less stressful and more efficient.

Our conceptual approach is to develop an application that can provide real-time information about parking availability and regulations within a user-defined area. The application will be a multi-platform mobile application, accessible to a broad range of users and designed with a focus on simplicity and effectiveness.

The application frontend will be developed using React Native, providing a cross-platform interface that is both responsive and user-friendly. The backend, comprising the core of our solution, will consist of a Python API built using the Flask framework. This backend will handle the application's complex tasks like processing large quantities of data from various sources such as the Google Street View Static API and Open Street Maps. Additionally the backend will have an AI component that will process image data.

We plan to employ artificial intelligence, particularly through a use of machine learning models, to analyze images from Google Street View. The AI component of our design is to be tasked with identifying and interpreting parking signs, translating these visual cues into understandable, actionable information for the user. By integrating this technology, we aim to provide a solution that is not only accurate but also adaptable to the varied and dynamic nature of urban parking regulations.

The decision to use a combination of React Native and Python stems from our desire to balance technical robustness with user accessibility. React Native allows us to create an intuitive interface that can engage users effectively, while Python offers the necessary capabilities for complex data processing and machine learning tasks.

In the initial stages of conceptualization, we explored several alternative solutions. One such approach was to create a database-driven application that relied on manually updated information about parking regulations and availability. While this method offered simplicity, it would fall short in providing real-time accuracy, a critical element given the dynamic nature of urban parking scenarios.

Another concept involved leveraging crowd-sourced data to map out available parking spots. This approach promised real-time data but lacked reliability and scalability, crucial factors for widespread adoption and effectiveness.

We also considered developing a purely AI-driven solution that bypassed the need for integrating multiple external APIs. However, this idea was set aside due to the immense challenge of creating an AI capable of independently processing and interpreting the vast and varied data required for such a task.

Our chosen approach with WhereTo effectively addresses the core needs of the customer: real-time accuracy, ease of use, and adaptability. By combining a React Native frontend with a Python API backend which leverages machine learning, we can ensure that our application is not only technologically sound but also accessible to a general user base.

The integration of Google APIs and Open Street Maps data provides us with a reliable source of up-to-date geographic and visual information. This is crucial in ensuring the accuracy of the parking information provided to the users. The AI component enhances this further by interpreting visual data in the form of parking signs to provide parking regulation insights to the users.

Our approach highlights the scalability of the design. While initially we are focused on certain urban areas, notably testing our backend algorithms in Boston, the application's architecture and lack of integration with local data allows for easy expansion to new locations, adapting to different urban layouts and parking regulations.

Our engineering design process centers on developing a system that is efficient, reliable, and user-friendly. The application UI must be intuitive and easy to navigate, ensuring that users of varying technical backgrounds can utilize it effectively. The backend, on the other hand, must be robust, capable of handling large datasets, and quick in processing and delivering information. The backend must also be capable of handling different failure cases as we are integrated with many different API endpoints.

For the success of WhereTo, the frontend must seamlessly communicate with the backend, sending user input in the form of location and radius and receiving processed data in the form of a visual map of parking availability and regulations in a format that is easily interpreted.

System Description:

We will offer an application that is capable of delivering real-time parking regulation information to users in need of parking. In order to create an application that performs this task, we will need to create a full stack mobile application. This application will consist of a React Native UI, a Python API, interaction with Google APIs, and use of a MongoDB client to cache our results. In addition to these components, we will need to use outside open source data from Open Street Maps, in order to traverse street segments to collect photos using Google Street View Static API.

Figure 1 in Appendix 3 shows a largely complete system diagram envisioned for the WhereTo application. Clearly visible on the figure are the interactions between the React Native UI, Python API, Google APIs, OSM data, and MongoDB. Each has a vital importance to the design and functionality of the system.

The React Native UI component is the component of our system which is user-facing. This component is code for the application which the user will be able to download from their respective app store. This UI is where the user enters in the address of the location they want to park near as well as a radius (< .15 miles) in which they would be comfortable parking. This information is sent as a request to our applications Python API, which eventually returns back to the React Native UI, either with an error stating that the request was unable to be completed, or with a visualization of parking results in the form of an image displayed onto the screen.

The Python API component of the design is the most complex, holding nearly all of the logic used in completing the end to end process. The main endpoint for this API is the find_parking endpoint. This is an endpoint that takes in an address and radius and returns an image of parking availability. A typical process flow for this endpoint will be as follows:

1. The Python API will receive the string address and number radius from the React Native UI

2. The Python API will utilize the Google Geocoding API in order to convert the string address into a more usable latitude-longitude coordinate pair

3. The Python API will query the OSM API for a square of OSM data with length equal to two times the provided radius, getting a list of streets in the area and the locations of these streets marked in latitude and longitude

4. The location information about the streets in the area is used to query the Google Street View Static API with multiple different angles, in order to get pictures of every relevant street in the area

5. These photographs are fed into an AI object detection model which is inside of the Python API. This model will be capable of detecting parking signs, making bounding boxes around the signs and cropping them into an image as shown in Figure 5 in Appendix 3

6. Any parking signs detected will be forwarded by the Python API to the Google Cloud Vision API to read any text on the signs

7. Depending on the text present on the sign, the Python API will determine if it signified parking availability or lack thereof for that particular road segment

    a. For every street segment run through the algorithm, a copy indexed by its location should be stored into MongoDB. This will allow us to not run images with indices already present in MongoDB multiple times through our model and the Google Cloud Vision API.

8. The Google Static Maps API is queried, getting a static image of the map to be sent forward to the React Native UI, centered at the provided address

    a. This image is overlaid with all of the relevant parking information, utilizing the Google Maps Static API endpoint in order to add markers to the map at dynamic locations, indicating parking availability.

9. The Python API returns success and provides the image to the React frontend

Both the set of Google APIs and the OSM data API are vital to the functionality of our Python API find_parking endpoint. We rely on Google for our image data, geocoding, and text detection, three vital parts of our application. We require the OSM data API for our implementation as it will be used to traverse the streets around the area of interest. Google does not have an API to query latitude and longitude coordinates of streets in an area, so we must workaround this by utilizing the OSM API hosted by Bunting Labs. This API allows us to query for street location data in a given bounding, which we can then process to fit our needs and query the Google Street View Static API.

This is a complex endpoint, relying on many different API endpoints to complete its entire pipeline. The hope is that by introducing the MongoDB cache, we will be able to limit excessive calls to these API endpoints, reducing the total cost of our system. The MongoDB cache allows us to modify the endpoint so that we do not run the same street segment multiple times through our algorithm; if we process a street segment once, we will not have to again until Google updates the Street View for that segment. Because of this, the above pipeline could be modified so that after the call to the OSM API, we first check each street segment if it is located in the MongoDB client, and if it is we pull the values from that segment into the Python API from the client, rather than recalculating it all using the various endpoints and detection models.

First Semester Progress

Much of the progress of our first semester is present in the design work that we performed and defining our requirements within documents such as the PDRR. A great deal of work went into designing and understanding the process flow for our system. Additionally, the majority of the work placed into implementing our design this semester has focused on the Python API, as it is arguably the more complex between itself and the user interface.

React Native UI

1: *Repository Creation and Implementation Research*

The frontend repository for storing the relevant code for the React Native UI was created. In conjunction with this, we began to research how to use React Native to develop multi-platform mobile applications. We determined that the best path forward would be to utilize a framework such as Expo for developing and testing our application. The reason for this is that it provides the capability to create multi-platform applications on any device. Expo simplifies the process of creating and designing multi-platform applications, especially as it is very difficult to simulate iOS when testing on a computer.

We are happy to report the successful completion and deployment of our application's first screen as of our most recent project update. The start of the functionality of our user interface is marked by this important development milestone. Users have the option to input a specific location and specify a desired radius on this screen. These characteristics are essential to the main functions of the program and provide a strong framework for the latter stages of development.

To improve user interaction and ease of use, we have added a 'Submit' button in addition to the location and radius input fields. This button represents an important component of the application's user interface, enabling users to interact with the app by entering their desired parameters. To ensure an intuitive and user-friendly experience, great thought has gone into the design and placement of these elements. Our team has put a lot of effort into designing a layout that is not only visually beautiful but also effective and simple to navigate.

Our team's future goal is to build on this initial success. To guarantee dependability and efficacy, the current screen will undergo extensive testing and improvement in the upcoming stages. At the same time, we are organizing the next set of screens and features that will improve and supplement the user experience. We are still fully committed to providing a solid, high-quality app, and we believe that the upcoming changes will greatly increase the app's overall worth and usefulness for our users.

Python API

1: *Repository Creation and API Configurations*

The backend repository for storing the Python API code was created. In conjunction with this, we began to configure Google Cloud Console accounts. These accounts are useful as they allow us to access Google APIs that require billing, such as the Google Static Street View API and the Google Cloud Vision API. Additionally, these accounts and the experience in using the Google Cloud Platform will be useful when we have to deploy to the cloud, as we will likely deploy our API to the GCP.

2: Creation of Algorithm to Query Google Street View API on Straight Line
An algorithm was created that was able to query the Google Street View API on a straight line segment of a street. In order to perform this, the algorithm accepted a final and initial coordinate as input. A list of coordinates separated by a small distance between these initial and final coordinates is created, then parsed into queries to the Google Street View Static API, using 8 different angled views, 45 degrees apart. This use of multiple headings ensures that any signs on the side of the road will be picked up by the algorithm.

3: Creation of Algorithm to Analyze Text on Images using Google Cloud Vision
An algorithm was created that added to the previously created algorithm for querying the Google Street View API. The algorithm queries Google Cloud Vision with the images collected from the Google Street View Static API algorithm. Google Cloud Vision reads the text from the images and sends the text found, indexed by image name, into a JSON output file. This was the algorithm that we used during our First Prototype Testing.

4: Creation of Algorithm to Query Open Street Maps Data
An algorithm was created that was capable of querying data from the Open Street Maps project. We found an API hosted by Bunting Labs that allows OSM data to be exported as JSON. The algorithm we created takes in an address as well as a radius. The address is converted into coordinates via the Google Geocoding API, then the radius is converted to coordinate offsets to find the corners of a bounding box for the area we want to analyze. The streets are then stored into a dictionary where the keys are the names of streets and the values are arrays of street segments in order.

Machine Learning Model Development

1:Dataset Acquisition and Model Training
The semester began with the acquisition of a specialized dataset comprising 877 images across four categories: Traffic Light, Stop, Speedlimit, and Crosswalk. This dataset, annotated in the PASCAL VOC format, was instrumental in training our Convolutional Neural Network (CNN) model. It provided a diverse range of urban scenarios and parking signs, allowing the model to learn and adapt to real-world conditions.

2:YOLO Model Integration and Challenges
A significant portion of our efforts was devoted to integrating and training the YOLO model with this dataset. Our goal was to enable the system to create precise bounding boxes specifically around parking signs, addressing the limitation we faced with the Google Vision API, which tended to read all text in the images. Prototype testing

demonstrated the algorithm's capability to detect and bound parking signs, but it was noted that the model took about 10 minutes to train the model, which was longer than ideal.

3:Transition to CNN for Enhanced Performance
To address these challenges, we transitioned to a more sophisticated CNN model, specifically ResNet-34. This model was retrained with the same dataset, focusing on minimizing two types of loss functions: classification loss for sign type prediction and regression loss for bounding box prediction. The shift to CNN significantly improved the precision and processing speed of our system. The model's training and processing time was reduced to around 8 minutes, marking a substantial improvement from the initial YOLO model.

4: Continuous Refinement and Testing
The latter part of the semester was dedicated to testing and validating our model. We conducted several prototype tests, where new images were introduced to the model to assess its accuracy and efficiency in real-time scenarios. The use of DataLoader from PyTorch enabled efficient handling of our image data, and the model's performance was evaluated using loss functions and accuracy metrics. These tests were vital in ensuring the model's reliability and effectiveness in a variety of urban settings.

Following the creation of our algorithm to analyze text using Google's Cloud Vision API, we had our First Prototype Test. At this test we showed off the part of the Python API that we had completed by this point, this consisted of running the algorithm for fetching Google Street View Static API images for a stretch of Commonwealth Avenue and subsequently analyzing the text present in each image.

This test gave us valuable insights for how we should continue to go about creating our design. It showed us that we would need to create a bounding box around road signs to crop and send to the Google Cloud Vision API. The reason for this was that the Cloud Vision model was not just reading text relevant to parking, but rather reading all of the text present in the image, including street signs and advertisements. Implementing road sign detection prior to text analysis will ensure we do not read irrelevant text. After the prototype test we worked and continue to work on the Python API, experimenting more with the machine learning model in order to reliably bound around road signs.

Technical Plan

Completing our system will require ample time and effort placed into designing and implementing our components. We have to complete the mobile UI, the backend API, and integrate these with MongoDB in the form of a Cache. Each one of these components can be broken down into many separate tasks which, when completed, would signify a working end to end system.

React Native UI

Task 1: Design Initial Application Screen

It is necessary to develop the user interface's basic input screen. Appendix 3 has a basic representation of this screen, Figure 2. Four things need to be presented on this screen: the program title, an address input field, a radius input field, and a search bar. In the finished version, the address and radius will be sent to the Python API in order to find parking when a user clicks this search button. As a result, in order to make sure that the application does not study an excessively vast area, this screen will need to verify the radius's boundaries.

Task 2: Implement the Map Return Screen

Implementing the basic map return screen is necessary. Appendix 3 displays this screen as Figure 3. Mock data can be used for the map return image to the user interface while this screen is being created. The user must be presented with the map, with the entered address, radius, and button to start a new search displayed underneath it. When this screen is fully implemented, the user will be able to click on overlay icons on the maps to open a detailed information modal.

Task 3: Implement the Detailed Information Modal Screen

The comprehensive information modal screen depicted in Figure 4 of Appendix 3 will be put into use. An address range, text about parking rules, a confidence score, and an image crop of the parking sign or meter that informs the AI model's choice will all be sent to this screen as input data from the Python API. This user interface element needs to process the data and create a user-friendly graphic that makes the address range and parking availability evident.

Python API

Task 1: Create Street Traversal Algorithm

An algorithm shall be created which takes in a string address and a number radius as input and return a dictionary. This dictionary is indexed by street name and the value is an array of latitude-longitude coordinate pairs. These pairs should be in the order as would be logical traveling in either direction down the street; each coordinate pair should be preceded and succeeded in the array by its closest coordinate point neighbors lying on the same street.

Task 2: Develop Object Detection Model - Parking Signs

We need to create an object detection model capable of detecting road signs that have to do with parking, or perhaps all road signs, so that they can be forwarded to an image text detection model. This will allow us to run such text detection models without fear that we will pick up irrelevant information (such as advertisements). This model should create a bounding box around anything it perceives as a parking road sign.

Task 3: Develop Object Detection Model - Parking Meters

We may need to create another object detection model. This one should be capable of detecting and bounding around parking meters. This is to supplement our plan of text analysis of the parking signs with the presence of parking meters. This model should signify for any image if there is a parking meter and draw a bounding box around it.

Task 4: Create Regulation Determining Algorithm

A function shall be created which is capable of taking image data previously gathered from Google Street View Static API and processing it to determine for each picture whether or not parking is likely possible. The hope for this function is that we will be able to send data we query from Google Street View Static API into our own model to detect road signs as well as parking meters. Any road sign crops should be sent to Google Cloud Vision to have the text analyzed. Based on the presence of meters and the text found inside the images, the model has to decide whether or not there is parking available there.

Task 5: Develop Algorithm to Display Information on Map Image

A function needs to be created to implement the final stage of the pipeline. Once we have a list of street segments with their coordinates and corresponding parking information, we need to query the Google Static Maps API in order to get an image of the map centered at the desired address (supplied in the find_parking endpoint parameters). We then need to overlay color-coded markers atop the street segments signifying the parking availability. This can be accomplished by modifying the URL parameters of the Google API request.

<u>Machine Learning Model</u>

Task 1: Comprehensive Model Enhancement with Diverse Data

The primary focus will be on expanding the machine learning model's capabilities by integrating a more diverse and extensive dataset. This dataset will include a broader range of sign types and more complex parking regulations, encompassing various geographic locations and urban settings. By doing so, the model will be able to recognize not only standard parking signs but also temporary and less common ones, thus enhancing its applicability and accuracy.

Extend the model’s capabilities to recognize a broader range of sign types, including temporary parking signs, and interpret more complex parking regulations. This enhancement will make the application more versatile and useful in different geographic locations.

Task 2:Implementation of Real-Time Processing
A significant part of this enhancement involves optimizing the model for real-time processing. The aim is to reduce the image analysis time without compromising accuracy, making the application more responsive and practical for real-time use. This could involve exploring and integrating more efficient neural network architectures like MobileNet or SqueezeNet, known for their speed and minimal computational demands.

Task 3: Automated Retraining and Model Updating
To maintain the model's relevance and accuracy, an automated retraining system will be established. This system will periodically update the model with new data collected from user feedback and updated street views. Such continuous learning will ensure the model adapts to changes in parking regulations and urban layouts over time. User interaction will be enhanced to include a feedback mechanism within the UI. Users can report the accuracy of the parking information provided, and this data will be used to further refine and train the model. This creates a dynamic feedback loop, where user inputs directly contribute to the system's improvement.

Task 4: Optimization of Model Training and Deployment Efficiency
A critical aspect of future development will involve significantly improving the efficiency of both the training and deployment phases of the machine learning model. Currently, the model’s training time and its ability to process new images in real-time are areas that require optimization. We plan to address this by refining the training process, potentially incorporating more advanced machine learning techniques such as transfer learning, which can expedite the training process by utilizing pre-trained models on large datasets. This approach can considerably reduce the time it takes for the model to learn to detect and analyze new parking signs.
Alongside this, we will introduce an intelligent caching system for locations or coordinates that the model has already processed. This means that for areas where the model has previously analyzed parking signs and regulations, it won’t need to reprocess or retrain on these locations unless there is new data or significant changes in the environment. This not only reduces redundant computational work but also speeds up the response time for the end-user, as the system can quickly retrieve previously analyzed data from the cache.

MongoDB

Task 1: Define the Data to be Stored

An exact schematic of the data that we will be storing inside of the MongoDB client needs to be created. We want to index our data by the unique coordinate pairs of our street segments. For each street segment, we want to store the coordinate point information of images processed in that section, along with information that would be displayed to the UI about that segment, such as its parking rules, the image cropping for the detailed modal, and other important information for the user to have access to.

Task 2: Implement an Interface for the Database for the Python API

A file should be created within the Python API that contains functions for writing and reading information to our MongoDB client. The following operations should be created: writing a street segment and its relevant information to the client, reading a street segment and its relevant information to the client, and checking if a street segment exists within the database. With these three functions created, the main Python API endpoint pipeline will be capable of reading and writing the relevant street segments to the MongoDB client. It will allow us to avoid analyzing street segments multiple times.

Deployment

Task 1: Deploy the Python API

The Python API needs to be deployed to the cloud so that we can begin integrating it with the React Native UI. Deploying the API to the cloud will require us to choose a platform. Google Cloud Platform may be a good choice as we are already familiar with the Google Cloud Console as well as some of the Google APIs. Another candidate to host our Python API is Amazon Web Services. When deploying the Python API to the cloud, we will need to ensure the configuration files are deployed properly, using a shared Google Account to access the API with credentials. This is also true for ensuring that the Bunting Labs and MongoDB APIs work properly with deployment to the cloud.

Task 2: Integrate the Python API with the UI

The UI shall be configured to be able to access the Python API hosted in the cloud. This will allow us to perform our end to end testing of the system. Until this task is completed, the UI design will be operating entirely under the use of mock data. In order to perform this integration, the React Native UI will have to be authenticated with a key for the Python API in the cloud. Upon this integration, comprehensive testing of the design shall be performed. This is to ensure that our error checking in both the frontend and the backend was sufficient, particularly in ensuring a proper address and radius are sent to the Python API endpoint for locating parking.

Budget Estimate

WhereTo as it stands is a purely software project, the only deliverables being the code for both the frontend and backend components of the system. Because of this, our entire budget will be used in usage of the various APIs that we connect with that charge money. The cost of each API can be found in Appendix 3, Table 1.

The cost of each of these APIs is not excessive for a single run of the endpoint, especially as the endpoint will only query in a radius no greater than .15 miles. This will reduce the amount of images that have to be collected and processed, reducing the amount of requests and subsequently the total cost of running our endpoint.

The actual cost of any given run of the endpoint will depend entirely on the density of streets in the area that the user wants to search for parking as well as on the amount of parking signage present. Higher density of streets will increase the price of the Street View Static API as more pictures will have to be collected to process. It would also increase the price of the Cloud Vision API if those streets all had parking signage to process through that API. For a given run of the endpoint, there will only be one call to the Geocoding API and one call to the Maps Static API, so these will be negligible in terms of our total cost of running the endpoint.

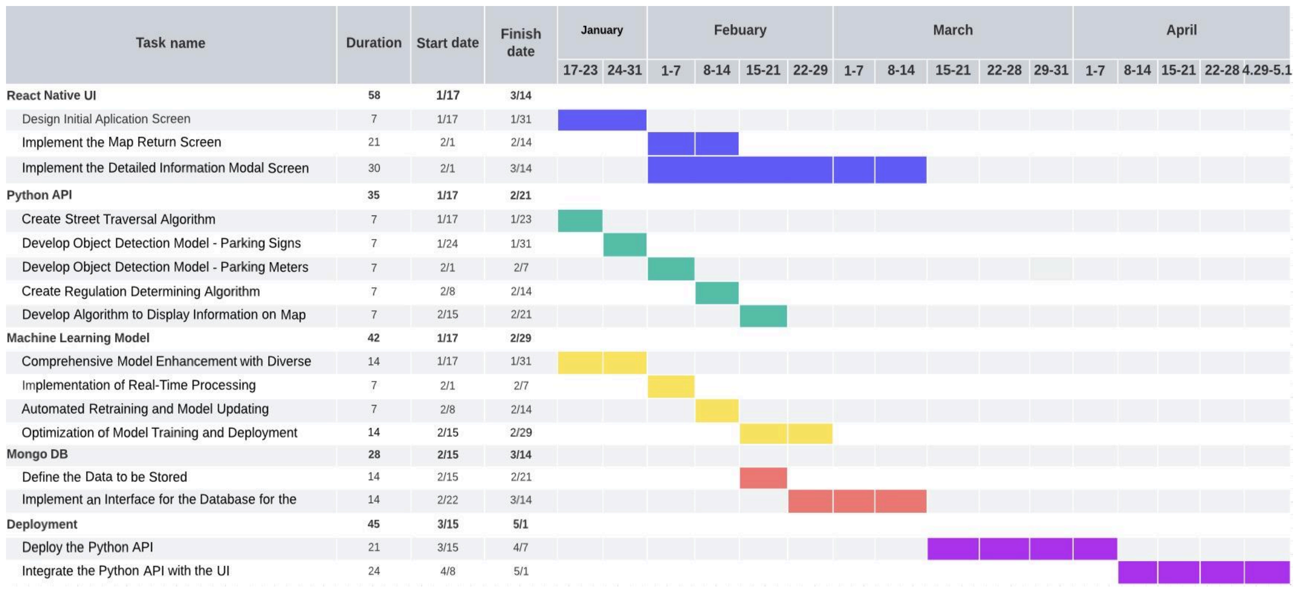
The only other cost that we will accrue during the project is the cost we will incur in hosting our backend Python API and AI model in the cloud. As it stands, $300 in free credits is given to new Google Cloud accounts for hosting and API services, so given our low marginal cost for hosting and API usage, we will likely not incur any real cost for the majority of the project's span.

Appendix 1 (Engineering Requirements):

Requirement	**Value, range, tolerance, units**
Real-Time Data Processing,	System must process and display parking information with a maximum delay of 60 seconds.
Accuracy of Parking Information	AI model must interpret parking signs with an accuracy rate of 95%.
User Interface Design	The application must offer an intuitive interface, easy to navigate for a broad user base.
Scalability	The system must be capable of scaling to different urban environments and adaptable to varying data sizes.
Reliability	The application must have a reliability rate of 99.5%, ensuring consistent performance.

Compatibility	The application must be compatible across multiple mobile platforms, including iOS and Android.
Backend API Security	The backend API must be secure and compliant with the latest security protocols.

Appendix 2 (GANTT Chart):



Task name	Duration	Start date	Finish date	January		Febuary				March					April				
				17-23	24-31	1-7	8-14	15-21	22-29	1-7	8-14	15-21	22-28	29-31	1-7	8-14	15-21	22-28	4.29-5.1
React Native UI	**58**	**1/17**	**3/14**																
Design Initial Aplication Screen	7	1/17	1/31	🟦	🟦														
Implement the Map Return Screen	21	2/1	2/14			🟦	🟦												
Implement the Detailed Information Modal Screen	30	2/1	3/14			🟦	🟦	🟦	🟦	🟦	🟦								
Python API	**35**	**1/17**	**2/21**																
Create Street Traversal Algorithm	7	1/17	1/23	🟩															
Develop Object Detection Model - Parking Signs	7	1/24	1/31		🟩														
Develop Object Detection Model - Parking Meters	7	2/1	2/7			🟩													
Create Regulation Determining Algorithm	7	2/8	2/14				🟩												
Develop Algorithm to Display Information on Map	7	2/15	2/21					🟩											
Machine Learning Model	**42**	**1/17**	**2/29**																
Comprehensive Model Enhancement with Diverse	14	1/17	1/31	🟨	🟨														
Implementation of Real-Time Processing	7	2/1	2/7			🟨													
Automated Retraining and Model Updating	7	2/8	2/14				🟨												
Optimization of Model Training and Deployment	14	2/15	2/29					🟨	🟨										
Mongo DB	**28**	**2/15**	**3/14**																
Define the Data to be Stored	14	2/15	2/21					🟥											
Implement an Interface for the Database for the	14	2/22	3/14						🟥	🟥	🟥								
Deployment	**45**	**3/15**	**5/1**																
Deploy the Python API	21	3/15	4/7									🟪	🟪	🟪	🟪				
Integrate the Python API with the UI	24	4/8	5/1													🟪	🟪	🟪	🟪

Appendix 3 (Figures and Tables):

Figure 1: System diagram of the proposed WhereTo application

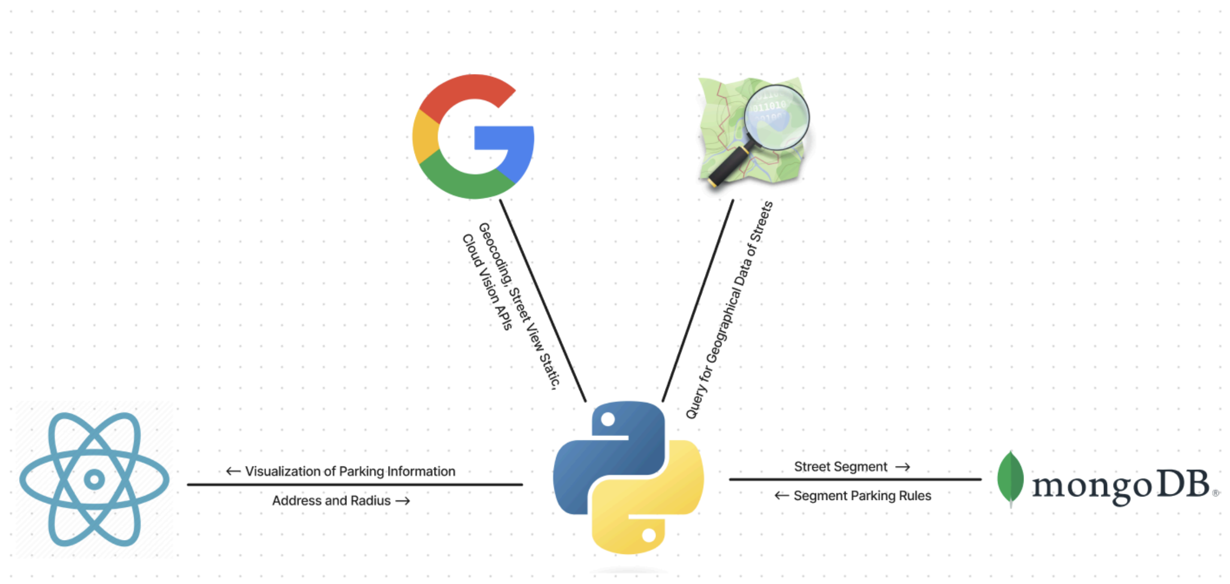

Geocoding, Street View Static,
Cloud Vision APIs
Query for Geographical Data of Streets
← Visualization of Parking Information
Address and Radius →
Street Segment →
← Segment Parking Rules
mongoDB

Figure 2: The input screen for the WhereTo application UI

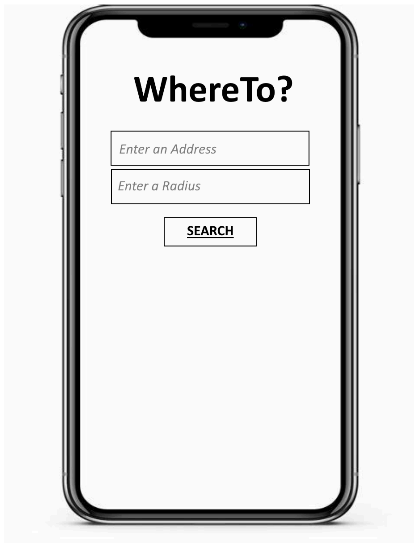


Figure 3: The map view screen for the WhereTo application UI

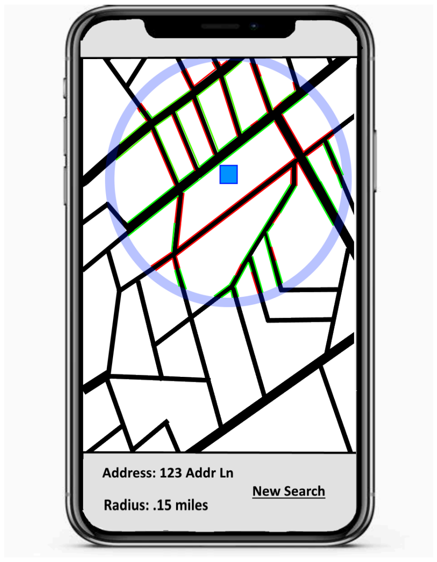
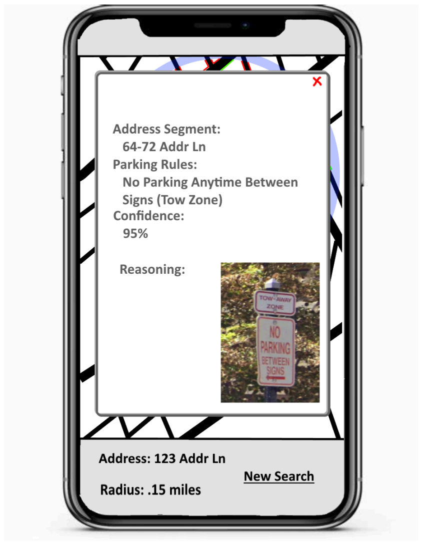


Figure 4: The parking detail screen for the WhereTo application UI



Address Segment:
64-72 Addr Ln
Parking Rules:
No Parking Anytime Between Signs (Tow Zone)
Confidence:
95%

Reasoning:

Figure 5: Bounding the Street Signs and Cropping them for interpretation of parking rules

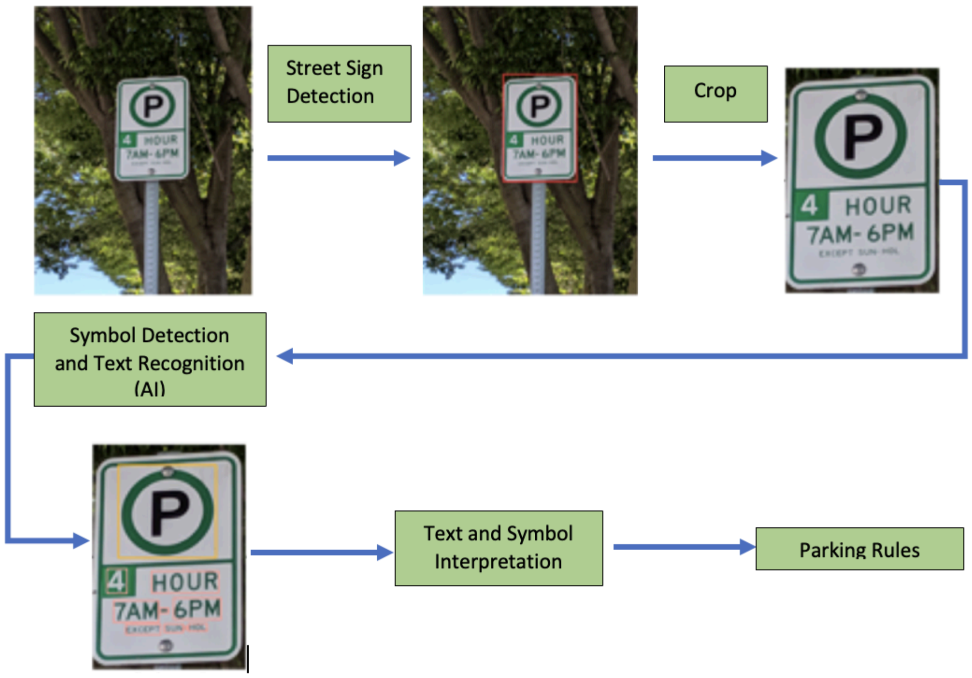

Street Sign Detection
Crop
Symbol Detection and Text Recognition (AI)
Text and Symbol Interpretation
Parking Rules
P
4 HOUR
7AM-6PM
EXCEPT SUN-HOL

Table 1: API Usage Marginal Cost Table

Item	Description	Cost
1	Google Geocoding API	$5.00 per 1,000 requests

2	Google Street View Static API	\$7.00 per 1,000 requests
3	Google Cloud Vision API	\$1.50 per 1,000 text requests
4	Google Maps Static API	\$2.00 per 1,000 requests