CS 307 Design Document - Group 6

BoilerPark

Pranay Nandkeolyar
Utkarsh Majithia
Anthony McCrovitz
Logan Portscheller
George Samra
Neel Vachhani

Index

Purpose

Purdue's campus, with nearly 50,000 students plus faculty, staff, and visitors, lacks a way to view real-time parking availability. Current resources only list parking options but provide no information on open spaces, leaving commuters frustrated and wasting time searching for spots. For 58% of the Purdue population, that means trying lots and searching for spots that may or may not be full. Instead of aimlessly searching, the Purdue Parking app can streamline spot-searching by providing live availability data to help users make faster, more informed commuting and parking decisions.

The purpose of the Purdue Parking app is to create a mobile-only app that will allow people to see the live number of parking spots left in a given parking structure by paging a cloud server (Redis) containing key-value pairs which serve as spot counters. Furthermore, a PostgreSQL server containing historical data will be used to allow users to see historical insights on parking structures to make informed decisions about future commuting strategies. Though similar platforms at Purdue exist, none track the available parking spots for Purdue staff and commuters holding A, B, and C passes. Because of this pitfall, we hope that the Purdue Parking app reduces the stress and repugnance of commuting to Purdue's campus by allowing stakeholders to make more-informed parking decisions.

Functional Requirements

1. **Authentication and Onboarding:**
    1.1. As a user, I want sign in with Apple to make creating a user account easier
    1.2. As a user I want sign in with Google to make creating a user account easier
    1.3. As a user I want sign in with email and password to allow me to make an account and save my preferences
    1.4. As a new user, I want a simple onboarding flow (tutorial/walkthrough) so I understand the app quickly.
2. **Core Functionality:**
    2.1. As a driver I want a screen that shows updated counts for each garage
    2.2. As a user I want to be able to see accurate availability of lots (Whether the lot is open or not e.g. football games)
    2.3. As a user, I want to see a "Last updated <timestamp>" label on each lot so I can trust freshness.
    2.4. As a user, I want to search for lots by name/code so I can find them quickly.
    2.5. As a user I want to have a map with all of the garages/lots listed on them
    2.6. As a user, I want a detailed view (hours, floors, rules, walking time to destination) so I can evaluate options.

2.7. As a user, I want a color gradient on each parking structure (from green to red) that signifies how full a given parking structure is (e.g. green for empty and red for full).

3. **Computer Vision:**
   3.1. As a developer I want to set up a camera in a parking lot
   3.2. As a developer I want to use computer vision to detect cars using online training data
   3.3. As a developer I want the computer vision algorithm to not detect other objects or vehicles that are not cars
   3.4. As a user I want an accurate count of total parking spots in all garages
   3.5. As a developer, I want to sync the camera to the CV model
   3.6. As a developer I want to be able to count the automobiles accurately and store the data in a database and thus find the available spots
4. **Mapping and Navigation:**
   4.1. As a user I want to know when to leave my house to make it to events on time using a maps API
   4.2. As a driver I want a link to a navigation app set to my lot of choice so I can easily see directions to the garage I'm going to
   4.3. As a user I want to be able to add significant arrival locations to reduce friction and make creating plans easier
   4.4. As a user, I want to set a default "home" or "commute origin" so travel estimates always start from there.
5. **Trip Planning and Calendar:**
   5.1. As a user, I want to link my calendar to the app
   5.2. As a user I want to see a bar chart that shows how full a given lot is at the given time
   5.3. As a user I want those logs to be used to see what garages I end up in most frequently by day
   5.4. As a user I want the garage I end up in and at what time to be logged by the app
   5.5. As a user I want garages suggested to me based on my logs and where I end up most
   5.6. As a user, I want the app to suggest the nearest alternative parking structure when my chosen lot is full.
   5.7. As a user, I want a calendar of known closures/events per lot so I can plan ahead
   5.8. As a user, I want walking time from lot to event/class added to ETA so plans are realistic.
   5.9. As a user, I want to see a 'Time-to-Full' estimate on the lot detail screen so I can determine whether the lot could be filled by the time I get there or pick an alternative garage.

5.10. As a user, I want a confidence level on each lot's count (High/Medium/Low) so I can better assess how much I can trust the data for certain lots.
5.11. As a user, I want to see the price of parking in different lots so that I can pick the best garage considering my parking budget. (Grant St and Harrison St Garages)
5.12. As a a user, I want to see an average user rating for each lot so that I can factor in the quality of different lots when deciding which lot to park in.
5.13. As a user, I want to see which lots have covered spots for shade during the hot summer months or protection from the elements like rain and snow.

6. **Eligibility, Accessibility and Personalization:**
   6.1. As a student or employee, I want to filter lots by my parking pass so I can know what garages I'm allowed to park in
   6.2. As a driver, I want to filter for ADA/accessible parking spots if the university provides that data.
   6.3. As a user, I want to favorite lots so they show at the top of my list.
7. **Analytics:**
   7.1. As a user I want to compare the insights from 2 or more garages
   7.2. As a user, I want a bar chart displaying historical average hourly insights to see when the best time to park in a given garage is.
   7.3. As a user, I want seasonal/weekly based historical insights to see which days it may be easier to park in a week and how seasonal weather affects parking
   7.4. As a developer, I want to train a model to predict parking spots based on historical hourly, weekly, and seasonal trends as well as current data and active users
   7.5. As a user, I want the app to prompt me with alternative transportation methods if most parking lots are expected to be full by a given departure time.
8. **Notifications and Preferences:**
   8.1. As a user I want push notifications that tell me when to leave
   8.2. As a user I want push notifications when lots are closing
   8.3. As a user I want push notifications when my car is in a lot that is getting towed
   8.4. As a user I want push notifications when my car is frequently associated with a lot that is being towed
   8.5. As a user, I want to manage notification preferences (which alerts I get and when).
   8.6. As a user, I want push notifications when a favorite lot drops below N spaces so I can leave sooner
   8.7. As a user I want a push notification that tells me when parking passes go on sale
   8.8. As a user I want a push notification after I enter a garage that tells me to park safely between the lines
9. **User Feedback and Sharing:**

   9.1. As a user I want to be able to share a parking schedule with others over text, email e.g.

- 9.2. As a user I want to see a map with the live counts displayed on top of the garages
- 9.3. As a user I want to be able to filter that map by parking pass or favorite lot
- 9.4. As a user, I want to toggle between a list view and a map view depending on my preference.
- 9.5. As a user, I want to report incorrect lot status (e.g., the app says open but it's actually full) so developers can improve data quality.
- 9.6. As a user, I want to rate accuracy ("was this lot accurate?") so the model improves.

10. User Interface:

- 10.1. As a user I want an attractive app icon that follows the new liquid glass design language for iOS users
- 10.2. As a user, I want the color scheme of the app to follow the Purdue color scheme as a Purdue-associated app.
- 10.3. As a user I want smooth animations that make using the app more pleasant
- 10.4. As a user, I want to choose dark mode/light mode so the app fits my phone theme

11. Miscellaneous:

- 11.1. As a student or staff member, I want the app to correctly work on the latest versions or ios and/or android so that I can use it no matter my phone
- 11.2. As a developer I want to set up a Redis cache and a Postgres SQL database that stores the data and sends it to the frontend
- 11.3. As a user, I want fallback messaging ("Data temporarily unavailable") instead of blank screens.
- 11.4. As a user, I want the app to integrate with my car's infotainment system (CarPlay/Android Auto) so I don't have to look at my phone while driving. (If time permits)
- 11.5. As a developer, I want health checks for cameras/CV services so I'm alerted when a feed drops.
- 11.6. As a developer, I want structured logs + metrics (latency, error rate, queue lag) so I can trace problems.
- 11.7. As a developer, I want to import permit rules from a campus feed so eligibility stays current.
- 11.8. As a developer, I want to ingest event calendars (athletics, concerts) so predicted demand is accurate.

Non-Functional Requirements

12. **Architecture and Performance:**
Our Project's Frontend will be written in React Native for cross platform delivery in iOS and Android. The backend will be implemented in Java/Python and will be deployed as containerized microservices. Redis will be present to provide low latency caching for live counts. PostgresSQL will be used as the main database to store daily counts, analytics and metadata. A lightweight computer-vision service will process camera streams and publish enter/exit events to the backend. We will target an end-to-end median response time of < 500 ms for API reads with 95% < 1s and a mobile app cold start time of < 2 s on mid-range devices. The live availability counter will reflect events with a freshness target of ≤ 10 s from vehicle passage to user display with 95% < 20s. The system should support ≥ 1,000 concurrent users. The app should have 24hr uptime, apart from scheduled maintenance and outages.

13. **Security and Privacy:**
All network communication will use TLS 1.2+. Authentication will support Sign in with Apple/Google and email/password and tokens will be short-lived with secure refresh. At rest, Redis and PostgreSQL volumes will be encrypted; secrets will be managed via environment variables with no secrets in source control. Camera processing will not store raw video frames unless explicitly authorized for test/debug, the default pipeline emits counting events only (no faces/plates). Any temporary debug captures will be time-boxed and access-controlled. Access to admin tools will require role-based access control.

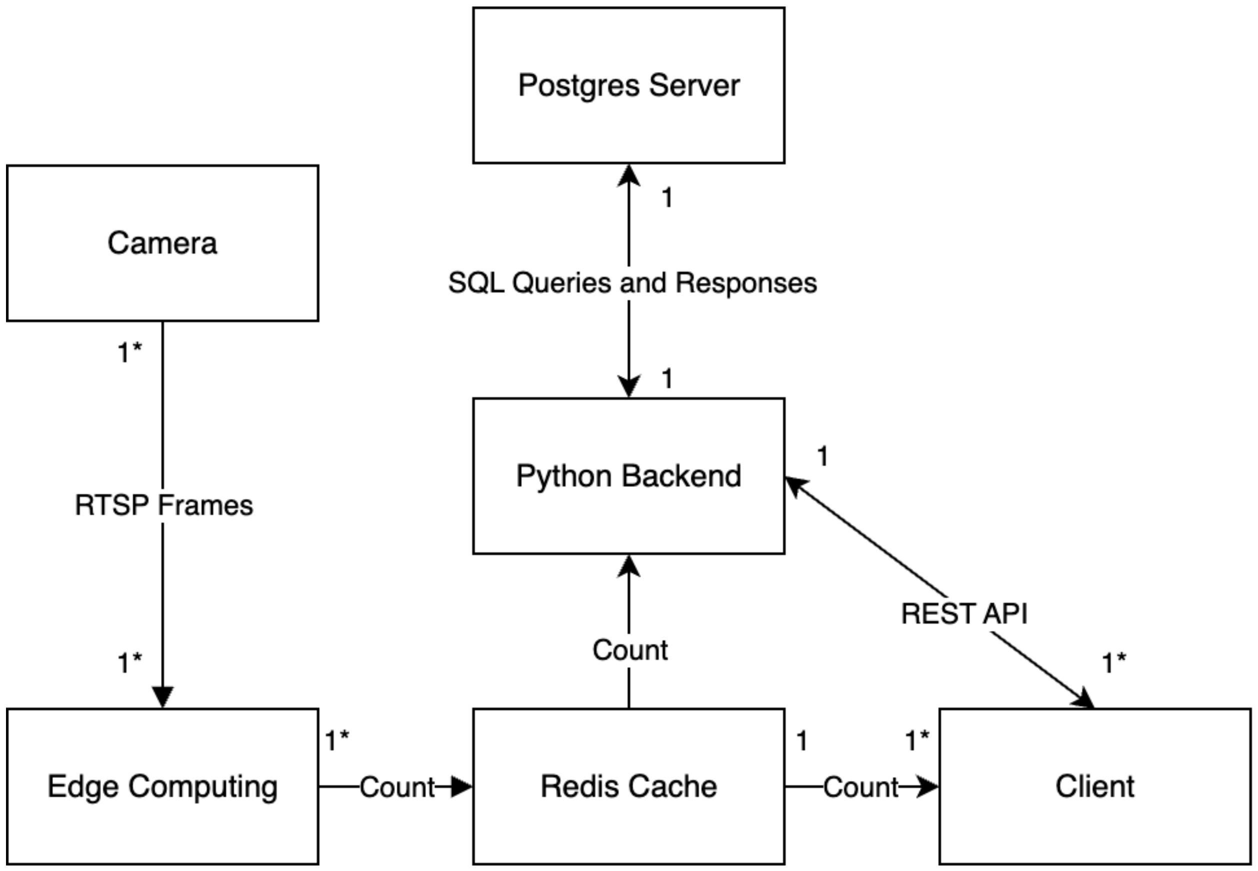
14. **Usability and Design:**
The interface will be designed for one-handed use, with primary actions reachable within 2 taps from the home screen. We will support dark/light modes, large-text settings, and ensure high contrast for key screens. Map and list views will be interchangeable, with persistent preferences. Critical information such as lot status and "Last updated <timestamp>" will be visible at a glance. We will implement a functional UI with smooth transitions that will look pleasing on both Android and iOS devices.

15. **Compliance and Ethics:**
We will respect campus policies on camera placement and data handling, and commit to privacy-preserving CV (no biometric identification, no license plate retention). Location and calendar access will be opt-in with clear purposes stated. User-generated feedback will be moderated to remove sensitive data or misuse. If required by campus policy, we will provide a Data Protection Impact Summary for review.

Design Outline

High-Level Overview:

The Purdue Parking app will mainly use a distributed edge-computing architecture and consist of the edge computers tracking incoming and outgoing vehicles, the servers/databases handling the information given from the edge-computers (counter updates), and the clients who will receive updates from the Redis Cache for parking spots counters of eligible parking lots/structures. The centralized data processing is held by the Redis Cache, which synchronizes parking counts on active clients using channel subscriptions and REST API calls. Historical parking data is sent from the Redis Cache to the PostgreSQL database through a Python backend, which sends the aggregate data to the backend on a set interval to update the parking insights and historical data on the client-side.



The CV app will take in RTSP video streams from cameras, which are processed at the edge using OpenCV and a lightweight detector to convert frames into occupancy events. These events are then sent to a Python backend built with Django, which validates them, updates the Postgres server for persistent storage, and maintains live availability data in Redis for fast reads. The backend exposes REST and WebSocket APIs that allow the frontend (mobile and web clients) to fetch current parking availability, receive real-time updates, and perform administrative tasks such as calibration and overrides. This design ensures privacy by doing inference at the edge, efficiency through caching with Redis, and reliability by persisting canonical data in Postgres.

Design Issues

Functional Issues:

1. Should this be a web app or a mobile app?
   Option 1: Mobile app
   Option 2: Web app
   Option 3: Both

   We chose option 1, the mobile app, because we think that the vast majority of our user base will be accessing our service from their phones. We envision the majority of use cases while a user is in their car either driving or getting ready to leave. It is more practical and efficient to have a mobile app for them to use. We also have more experienced mobile devs on our team as opposed to web devs.

2. Do we need to have user accounts?
   Option 1: Yes
   Option 2: No

   We chose option 1 because of the fact that we are going to have a calendar feature that will let the user upload their schedule and see what times they need to be places. While a lot of the functionality of the app could be accomplished without the use of user accounts, we felt that to make the best experience for our users, we should allow them to make accounts and store some basic calendar data.

3. What information do we need from each user for their accounts?
   Option 1: Name, Email, Password
   Option 2: Email, Password

   We chose to collect the names of our users along with their emails and passwords because it allows us to create a more personalized experience for our users. We will be able to greet them by name when they login and send personalized push notifications.

4. What external accounts should we allow to link to our app?
   Option 1: None
   Option 2: Apple & Google
   Option 3: Apple, Google, and others (Github, Facebook etc.)

We chose to offer Apple and Google authentication to our users because of the ease of signing up and logging in provided by these SSOs. We want to make the user experience as easy and frictionless as possible, and want to offer our users the option to opt for these providers.

5. How do we handle navigation in our app?
   Option 1: Simply provide the locations of all garages with Google Maps links
   Option 2: Use Open Street Map, an open source mapping API, to find the closest applicable lot and link to that one with an external maps app
   Option 3: Implement our own navigation system using Open Street Map

We chose Option 2 for its combination of functionality and convenience. While it would be nice to provide all of the directions directly in the app for our users, it is impractical. There are a lot of navigation providers that do navigation far better than we're able to. However, we also want to not make our users figure out the closest parking lot to them. We want to use OSM to do that for our users and then link to an external navigation app of their choosing.

<u>Non-Functional Issues:</u>

1. Which framework should we use for our frontend?
   Option 1: React Native
   Option 2: Swift (iOS only)

While there are members of our team that are more familiar with Swift, the majority are more familiar with React Native. On top of that, React Native offers the ability to port the app to both the iOS and Android operating systems, allowing us to reach more users.

2. How should we send data from our cameras to our Redis store?
   Option 1: Send the camera feed to the python backend and run a CV model there and send a counter to the Redis store

Option 2: Train a lightweight CV model and use edge computing and send a counter to the Redis store

We chose the edge computing paradigm for security reasons. While we will not be able to train a model on the edge computer, we do not want to send a livestream of cars entering and exiting lots as a malicious user could intercept license plates and other potentially identifying information. By keeping that on the edge computer linked to the camera and only sending a count across a network to the Redis cache we improve the safety and security of our model.

3. What language/framework should we build our backend in?

   Option 1: Java

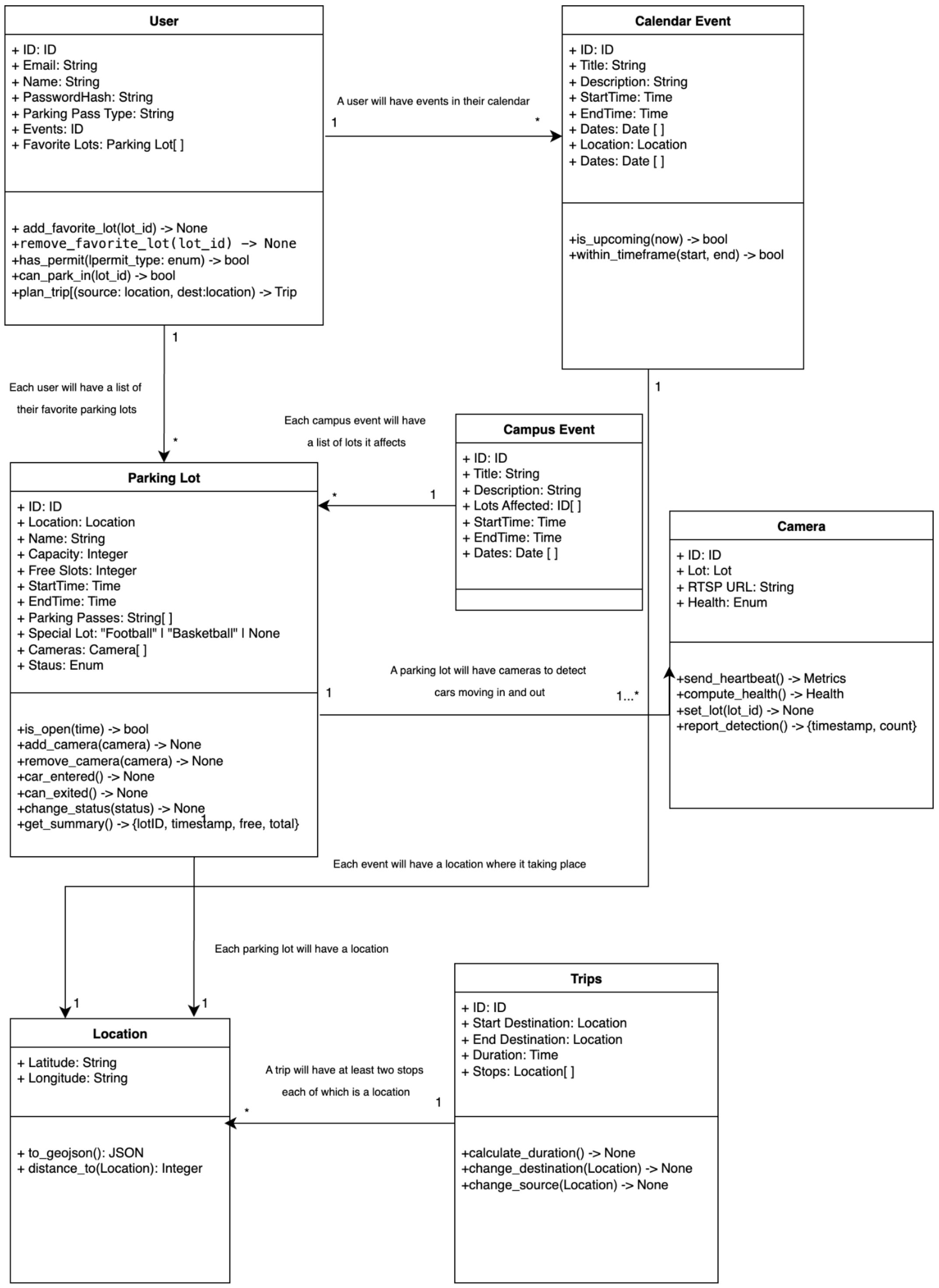
   Option 2: Python and Flask

   Option 3: Python and Django

We chose Python because of its compatibility with the OpenCV model. On top of that, Python is the data science language of choice, which makes it a great choice for the analytics that we're planning on adding to our app. We chose Django instead of Flask for its ORM model that will better structure our data and allow us to easily connect to our database.

Design Details

Class Design


User
+ ID: ID
+ Email: String
+ Name: String
+ PasswordHash: String
+ Parking Pass Type: String
+ Events: ID
+ Favorite Lots: Parking Lot[ ]
+ add_favorite_lot(lot_id) -> None
+remove_favorite_lot(lot_id) -> None
+has_permit(lpermit_type: enum) -> bool
+can_park_in(lot_id) -> bool
+plan_trip[(source: location, dest:location) -> Trip
A user will have events in their calendar
1
*
Calendar Event
+ ID: ID
+ Title: String
+ Description: String
+ StartTime: Time
+ EndTime: Time
+ Dates: Date [ ]
+ Location: Location
+ Dates: Date [ ]
+is_upcoming(now) -> bool
+within_timeframe(start, end) -> bool
1
Each user will have a list of
their favorite parking lots
*
1
Each campus event will have
a list of lots it affects
Campus Event
+ ID: ID
+ Title: String
+ Description: String
+ Lots Affected: ID[ ]
+ StartTime: Time
+ EndTime: Time
+ Dates: Date [ ]
*
1
Parking Lot
+ ID: ID
+ Location: Location
+ Name: String
+ Capacity: Integer
+ Free Slots: Integer
+ StartTime: Time
+ EndTime: Time
+ Parking Passes: String[ ]
+ Special Lot: "Football" I "Basketball" I None
+ Cameras: Camera[ ]
+ Staus: Enum
+is_open(time) -> bool
+add_camera(camera) -> None
+remove_camera(camera) -> None
+car_entered() -> None
+can_exited() -> None
+change_status(status) -> None
+get_summary() -> {lotID, timestamp, free, total}
Camera
+ ID: ID
+ Lot: Lot
+ RTSP URL: String
+ Health: Enum
+send_heartbeat() -> Metrics
+compute_health() -> Health
+set_lot(lot_id) -> None
+report_detection() -> {timestamp, count}
A parking lot will have cameras to detect
cars moving in and out
1
1...*
Each event will have a location where it taking place
Each parking lot will have a location
1
1
Location
+ Latitude: String
+ Longitude: String
+ to_geojson(): JSON
+ distance_to(Location): Integer
A trip will have at least two stops
each of which is a location
*
1
Trips
+ ID: ID
+ Start Destination: Location
+ End Destination: Location
+ Duration: Time
+ Stops: Location[ ]
+calculate_duration() -> None
+change_destination(Location) -> None
+change_source(Location) -> None

Description of Classes and Interactions

User

a. A User object is created when someone signs up with email/password or via SSO (Google).
b. Each user stores an id, name, email, and passwordHash (for email logins).
c. Users select a Parking Pass Type (permit) used for eligibility filters across lots.
d. Users can edit profile fields and manage Favorite Lots (add/remove).
e. A user may connect their calendar; created Calendar Events are linked to the user.
f. From the app, a user can view nearby lots, check eligibility, start a Trip, and see campus/closure notices relevant to their favorites.

Parking Lot

a. A ParkingLot object is created by an admin/import for each campus lot/garage.
b. Each lot stores an id, name, location, capacity, freeSlots, startTime, endTime, accepted parkingPasses[], specialLot flag (Football/Basketball/None), and a list of bound Cameras.
c. The lot's freeSlots are updated from camera detections or operator overrides.
d. A lot can compute isOpen(atTime) based on hours and active campus events, and isEligible(permitType) for user filtering.
e. Admins can attach/detach cameras to the lot and view the lot's recent availability history.

Camera

a. A Camera object is created when a physical camera is registered and bound to a Parking Lot.
b. Each camera stores an id, rtspUrl, the single associated lot, and a health enum (HEALTHY/DEGRADED/UNHEALTHY/OFFLINE/MAINTENANCE).
c. The camera periodically sends heartbeats (fps, latency, rtsp ok, etc.) that update health.
d. During operation, the camera reports detection events that the backend aggregates into the lot's freeSlots.
e. If health degrades or goes offline, the lot shows a warning and admins are notified.

Campus Event

a. A CampusEvent object is created by an admin/import to represent closures or high-traffic events.
b. Each event has an id, title, description, dates/startTime/endTime, and lotsAffected[].
c. When active, the event marks affected Parking Lots as closed or reduced capacity and surfaces alerts to users whose favorites are impacted.
d. Events can be added, edited, or ended early by authorized admins.

Calendar Event

a. A CalendarEvent object is created when a user imports a calendar or adds an event manually.
b. Each entry has an id, title, description, startTime, endTime, optional recurring dates[], and a location.
c. Calendar events are used to plan a Trip: the planner chooses an eligible lot near the event location and time.
d. Users can edit or delete calendar events; changes immediately affect future trip suggestions.

Trips

a. A Trip object is created when a user plans to travel to a destination or starts navigation to a selected lot.
b. Each trip stores an id, startDestination, endDestination, optional stops[], and estimated duration.
c. The trip selects a target Parking Lot using the user's permit, distance, and predicted availability at the arrival time.
d. Users can add/remove stops and finalize a trip; upon arrival, the app may log a session for analytics.

Location

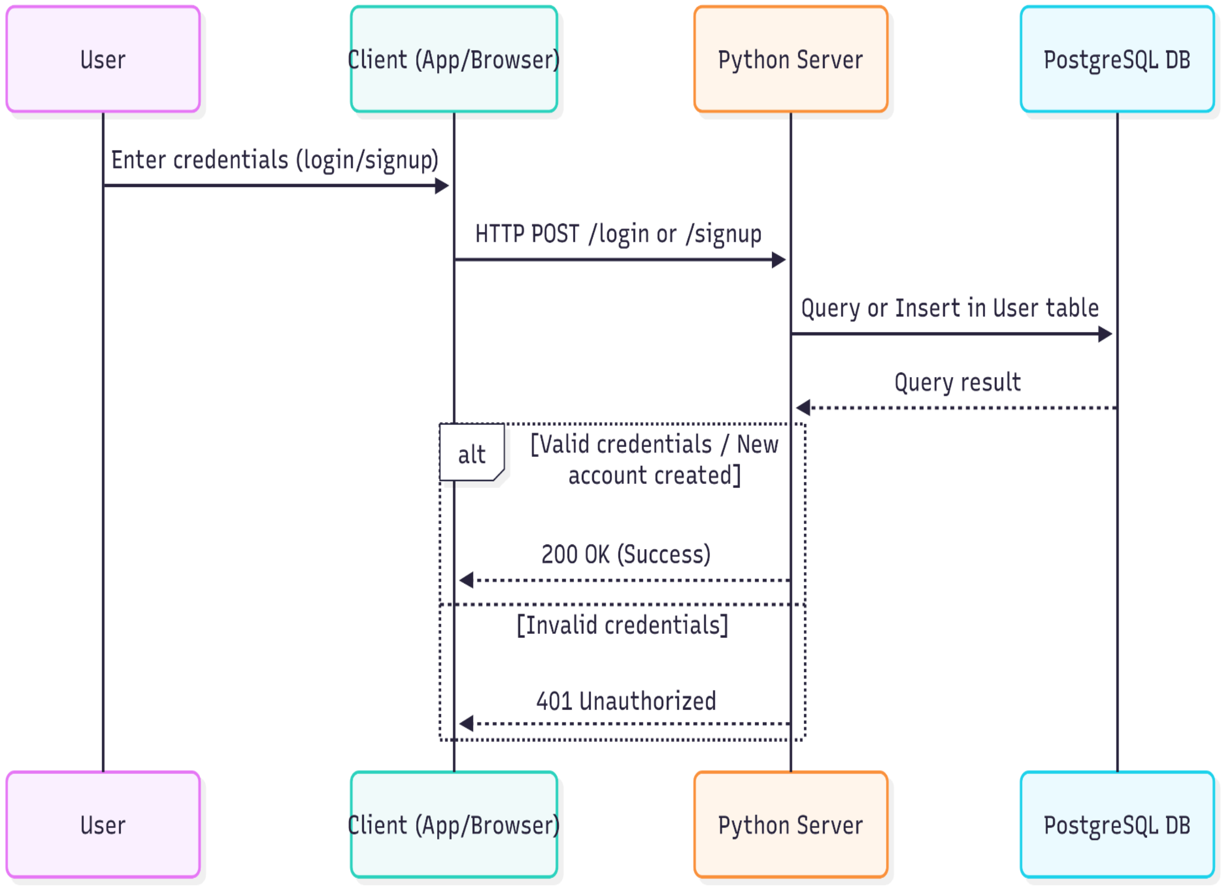
a. A Location object is created whenever a geo point is needed (lots, events, trip ends/stops).
b. Each location stores latitude and longitude.
c. Locations support distance calculations for sorting lots by proximity and mapping.

<u>Sequence of Events:</u>

In these sequence diagrams we have shown the interaction between the user, client, server, PostgreSQL database, Redis Cache, and the Edge Computing computer. The Edge Computing computer running the computer vision model and the Redis Cache interact to store accurate up-to-date counts of the number of free spots in a parking garage. The Redis Cache interacts with the main PostgreSQL database to store these counts for data-driven historical and predictive insights. The React frontend connects with the Python backend as an interface for connecting with the PostgreSQL and Redis Cache databases. The backend correctly directs the information retrieval to the correct database which gathers data from the camera. The backend then processes this data based on the user's REST API request and responds to the React frontend with the corresponding HTTP response code and data.

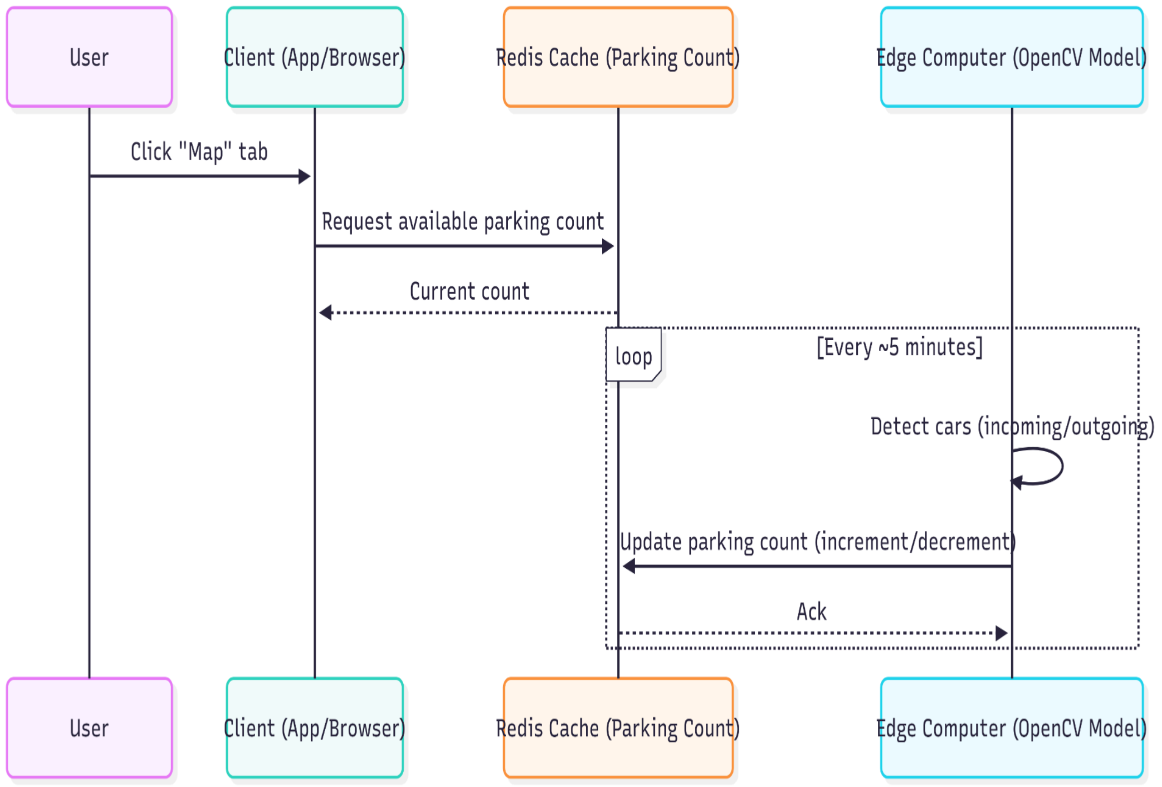
Sequence Diagram for Inserting and Updating User information:

The sequence begins when the user makes a new account or signs into their account. This data will then be given to the server which will either enter the user into the User table or run a query to validate the user credentials on the PostgreSQL. After the query, the Python server will process the query result and return a corresponding HTTP error code.



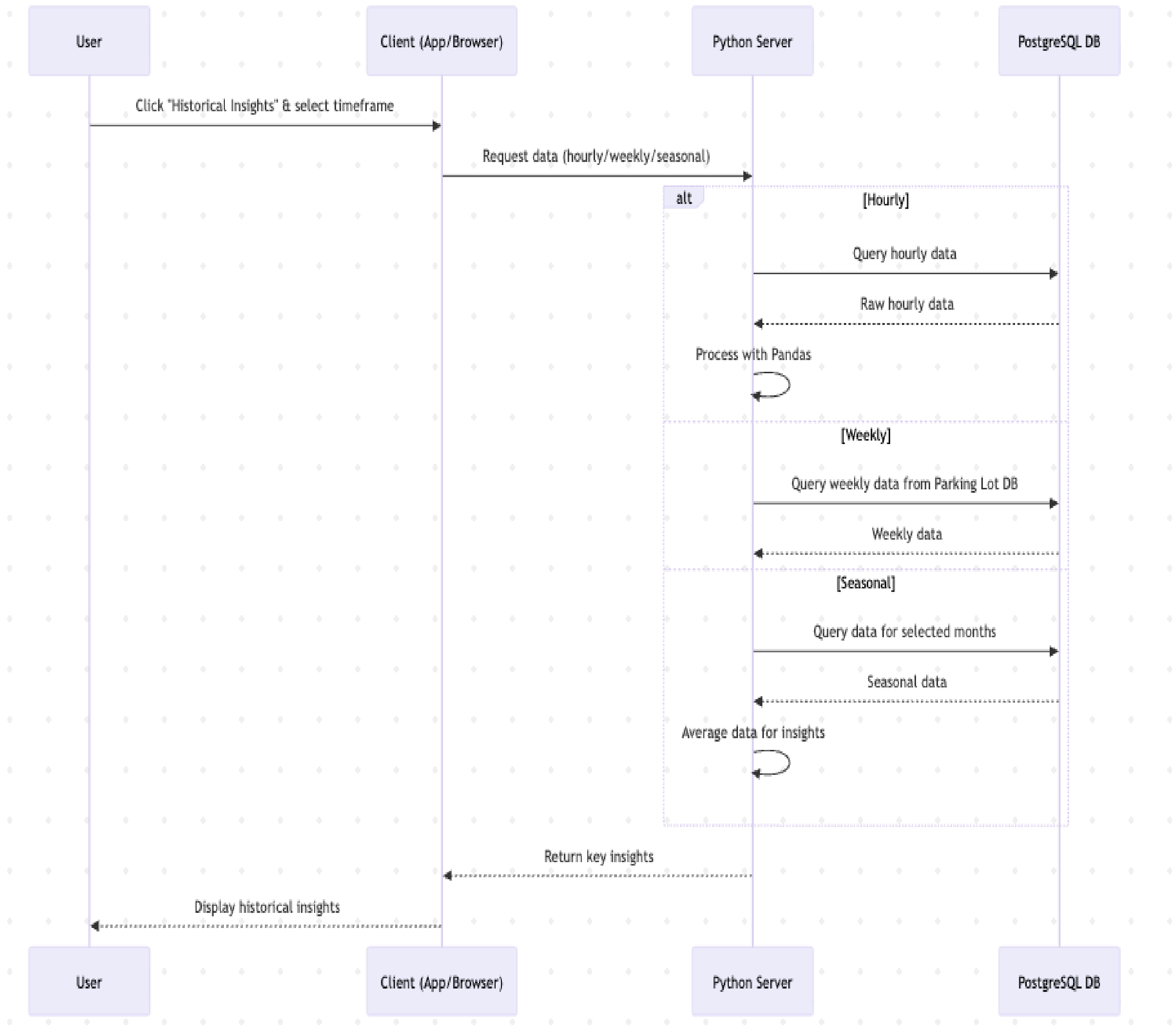
<u>Sequence Diagram for Updating Redis Cache and Map Feature:</u>

Once the account has been created, the user can choose to view the available parking spaces. They will do so by clicking on the map tab. Once that is complete, the user will send a request to the Redis Cache, which stores the count of parking spots available. The Redis Cache will receive this information from the Edge Computing Resource, a computer with a camera running an OpenCV model that tracks incoming and outgoing cars from a parking lot. This computer will then decrement or increment the count from the last stored count of open parking spots, defaulted at the total capacity of the parking spot. Every 5 minutes or so, this count is updated to the Redis Cache, where it is then fetched from the client.


User
Client (App/Browser)
Redis Cache (Parking Count)
Edge Computer (OpenCV Model)
Click "Map" tab
Request available parking count
Current count
loop
[Every ~5 minutes]
Detect cars (incoming/outgoing)
Update parking count (increment/decrement)
Ack
User
Client (App/Browser)
Redis Cache (Parking Count)
Edge Computer (OpenCV Model)

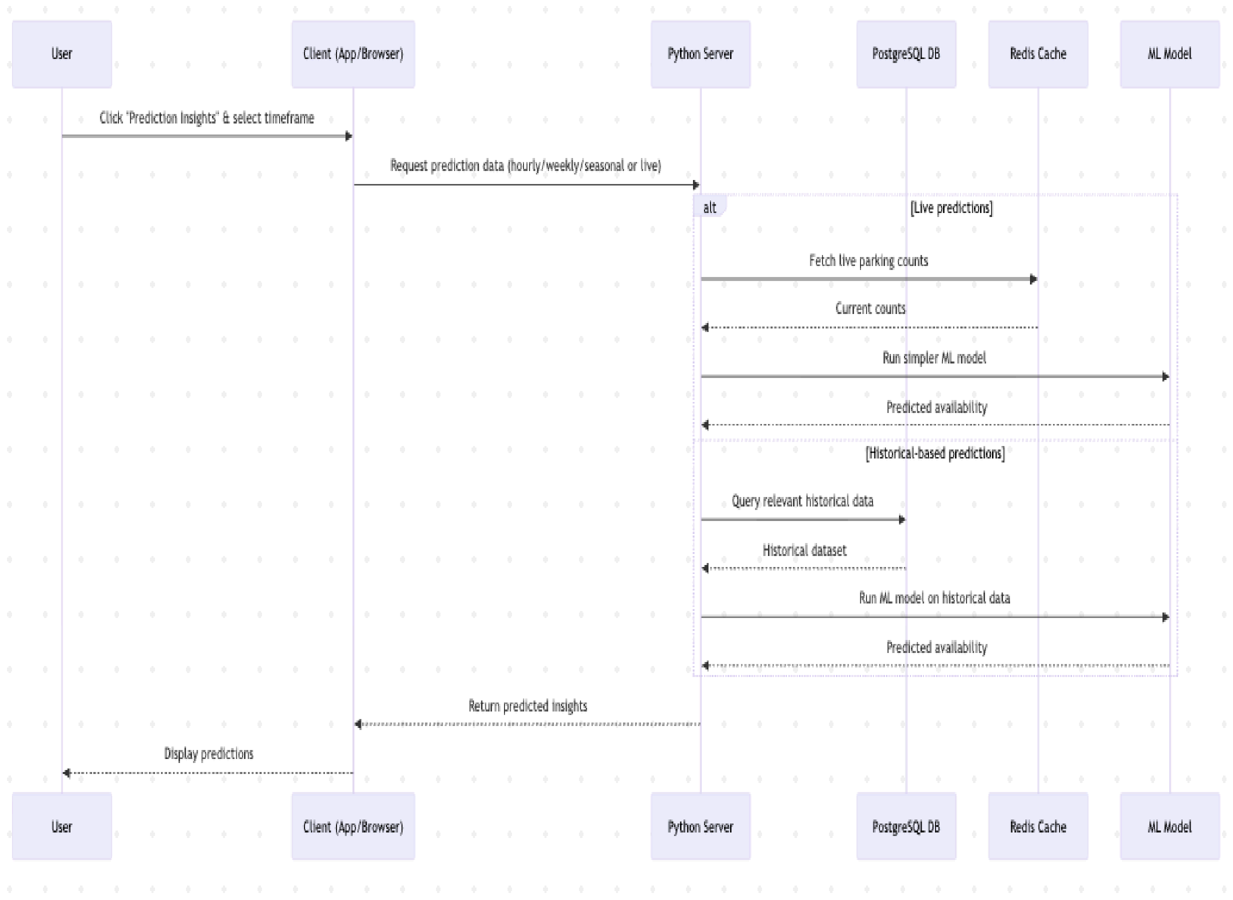
<u>Sequence Diagram for Viewing Historical Insights:</u>

The user can head to our historical insights page, which interfaces with our PostgreSQL database through our python server. The python server manipulates the data into the key insights the user requested before returning it back to the client. If the user requests hourly data, that will be executed through a PostGres query and potential pandas data manipulation as well. If the user asks for weekly insights, those will interface with the PostGres Parking Lot database for accurate data. Finally, if seasonal insights are requested, the database will query for certain months before averaging out.



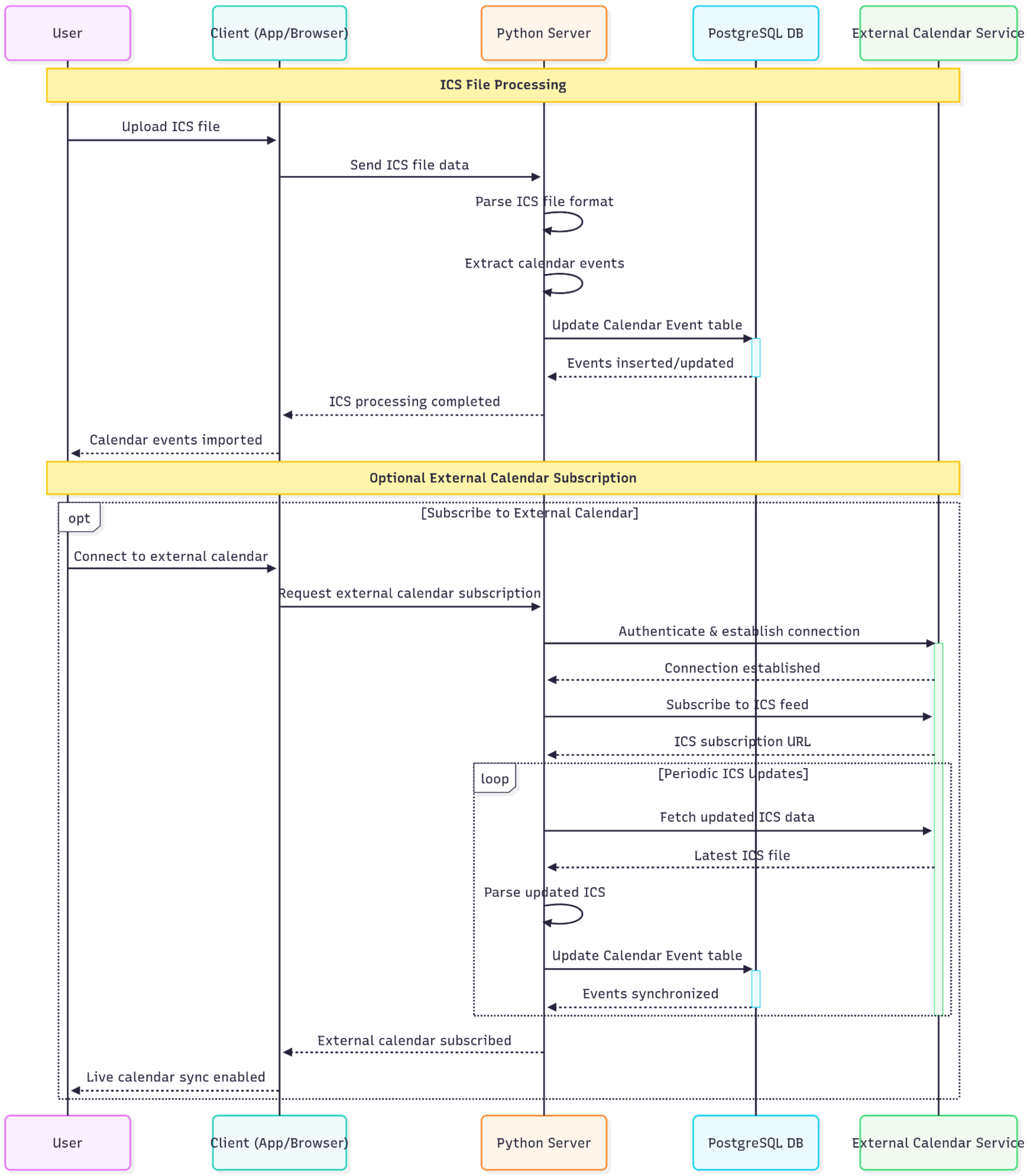
Sequence Diagram for Viewing Predictive Insights:

The user can also request predictions based insights. This will follow a similar flow to the historical insights with queries executed against the PostgreSQL databases. However, this data will then be interfaced with through a ML model which will utilize past data to predict availability of parking spots throughout the hours, weeks, and seasons. However, if live predictions are needed, the data will be gathered from the Redis Cache and a simpler model based on historical insights will be used instead.


User
Client (App/Browser)
Python Server
PostgreSQL DB
Redis Cache
ML Model
Click "Prediction Insights" & select timeframe
Request prediction data (hourly/weekly/seasonal or live)
alt
[Live predictions]
Fetch live parking counts
Current counts
Run simpler ML model
Predicted availability
[Historical-based predictions]
Query relevant historical data
Historical dataset
Run ML model on historical data
Predicted availability
Return predicted insights
Display predictions

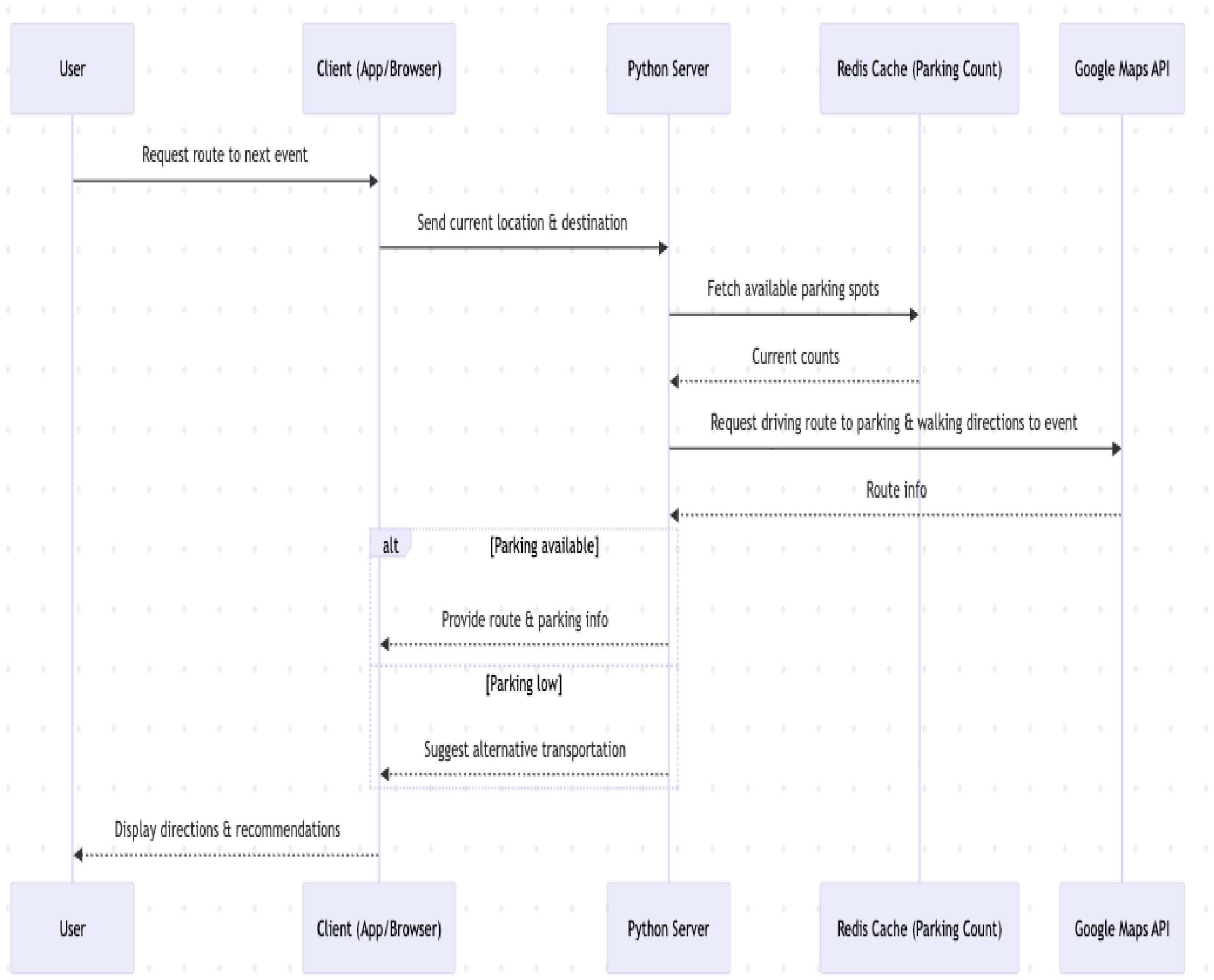
<u>Sequence Diagram for Calendar Information:</u>

The user uploads their ics file in the client application. This is then sent to the Python server, which updates the User and Calendar tables in the PostgreSQL database. After successfully updating the database, the server acknowledges the client, which then informs the user that the calendar has been updated. Additionally, users can subscribe to their calendar service, where the server then interacts with an external calendar service. Therefore, when the user updates their events, the python server parses the new ics file and updating the database correctly.



Sequence Diagram for Routing Requests

When the user requests directions to their next event, the client sends the user's current location and destination to the Python server. The server first checks all lot parking availability in the Redis Cache. Using this information, it calls the Google Maps API to generate a route to a parking spot near the event and walking directions from the parking spot to the destination. If parking availability is sufficient, the server returns the recommended route and parking information. If parking is limited, the server suggests alternative transportation options. The client then displays the directions, parking details, and/or alternative recommendations to the user.

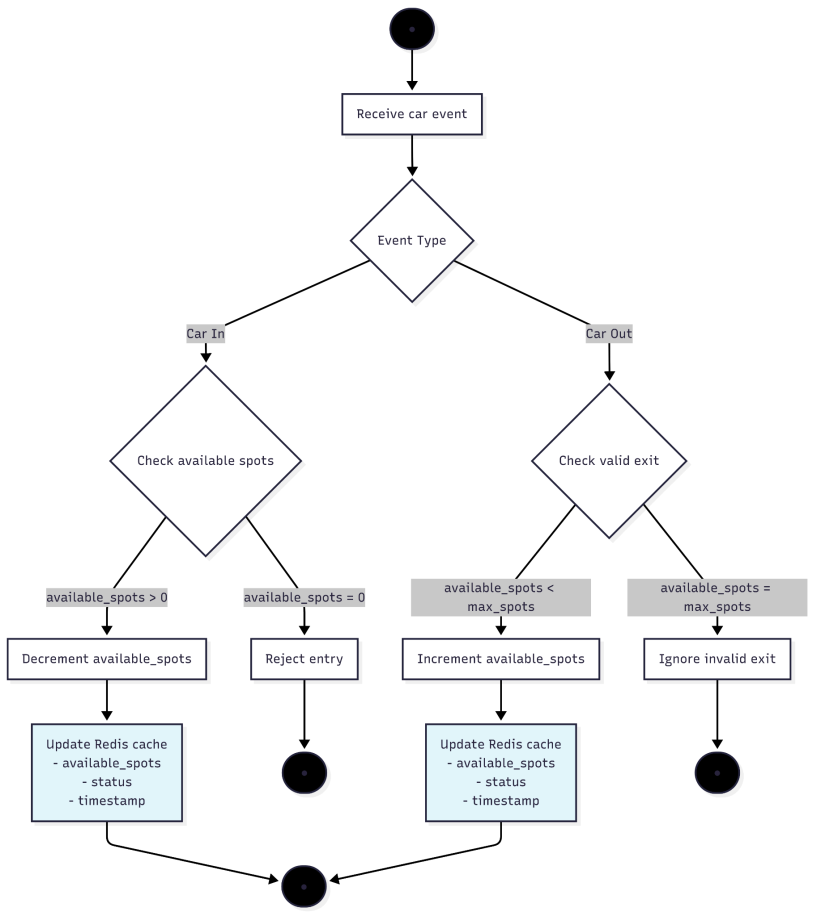

User
Client (App/Browser)
Python Server
Redis Cache (Parking Count)
Google Maps API
Request route to next event
Send current location & destination
Fetch available parking spots
Current counts
Request driving route to parking & walking directions to event
Route info
alt
[Parking available]
Provide route & parking info
[Parking low]
Suggest alternative transportation
Display directions & recommendations
User
Client (App/Browser)
Python Server
Redis Cache (Parking Count)
Google Maps API

Activity Diagrams

The following activity diagrams show how the internal system responds to internal events. The Redis Cache must respond to cars coming in and out of lots. Additionally, the PostgreSQL Database must request updated Redis cache information every few minutes as well to ensure that the data is up to date for the historical and predictive insight features.

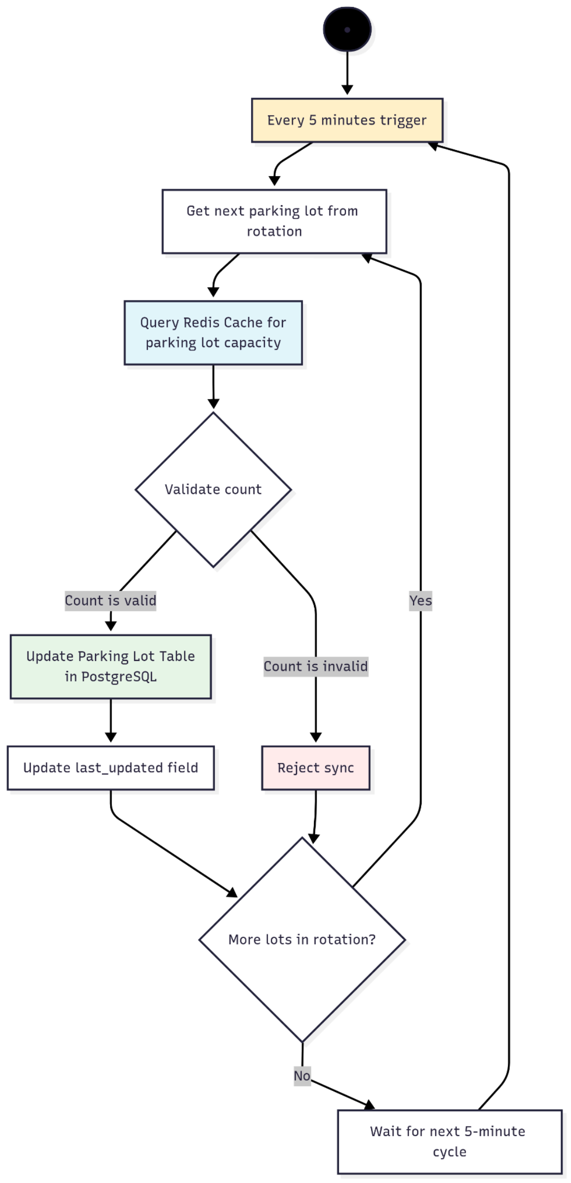
Activity Diagram for Handling Car Events

Whenever the camera detects a car, it compares the location of the car with the previous location of the car in the previous frame. If the car is moving into the parking lot between frames, a car in event is detected. Additionally, when a car is determined to be leaving, a car out event is detected. After checking that updating the available spots remains within 0 to lot capacity, the cache updates the available spots. Capacity is stored within the Reddis cache for each lot as well for quick access.

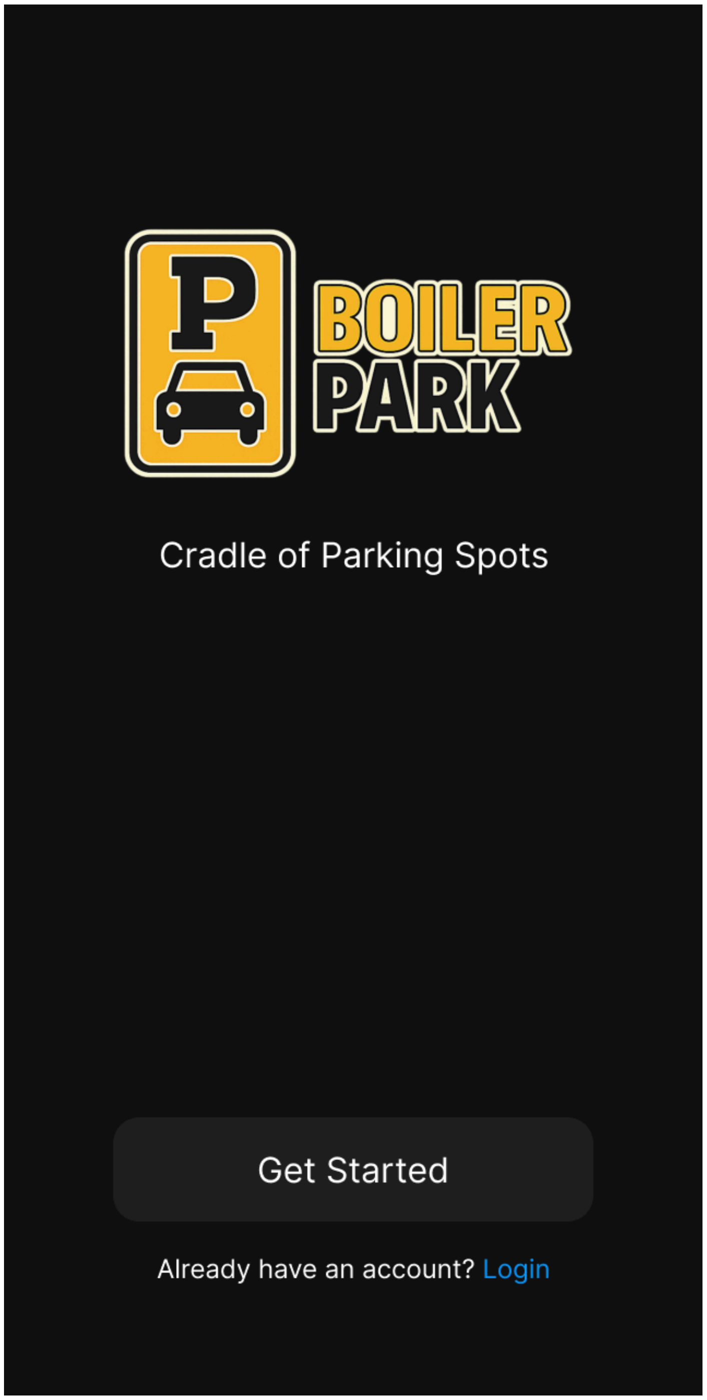

Receive car event
Event Type
Car In
Car Out
Check available spots
Check valid exit
available_spots > 0
available_spots = 0
available_spots < max_spots
available_spots = max_spots
Decrement available_spots
Reject entry
Increment available_spots
Ignore invalid exit
Update Redis cache - available_spots - status - timestamp
Update Redis cache - available_spots - status - timestamp

<u>Activity Diagram for Syncing Databases</u>

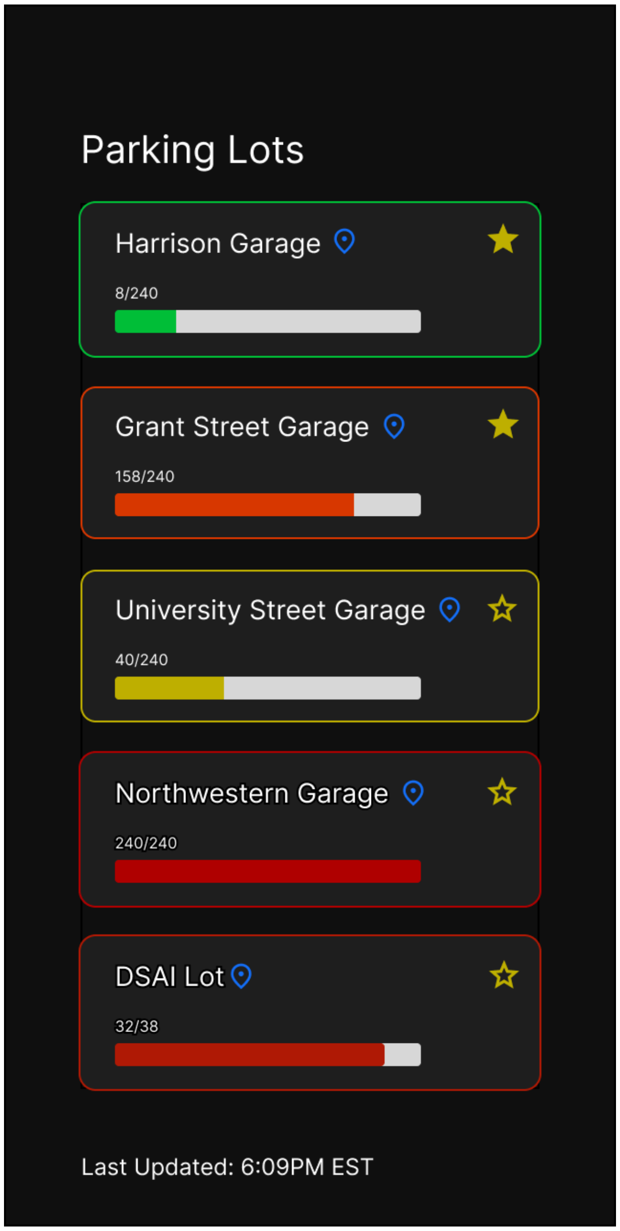
Every 5 minutes, the PostgreSQL Database will query the Redis Cache for the current capacity of each parking lot, operating in a rotation for each parking lot. After retrieving the available count, it checks whether or not the count is valid. If it is, it updates the Parking Lot Table and Class in PostgreSQL. If the count is invalid, it rejects the sync and does not update the last updated field either.


Every 5 minutes trigger
Get next parking lot from rotation
Query Redis Cache for parking lot capacity
Validate count
Count is valid
Count is invalid
Yes
Update Parking Lot Table in PostgreSQL
Update last_updated field
Reject sync
More lots in rotation?
No
Wait for next 5-minute cycle

UI Mockups



Homepage


Parking Lots

Harrison Garage ★
8/240

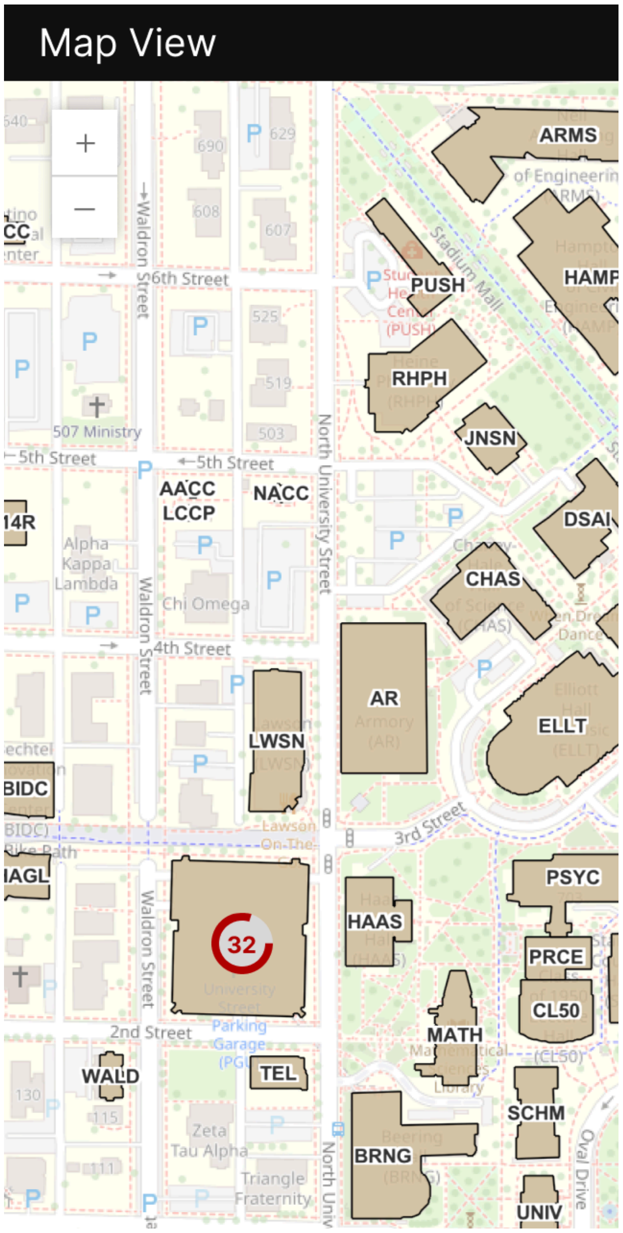
Grant Street Garage ★
158/240

University Street Garage ☆
40/240

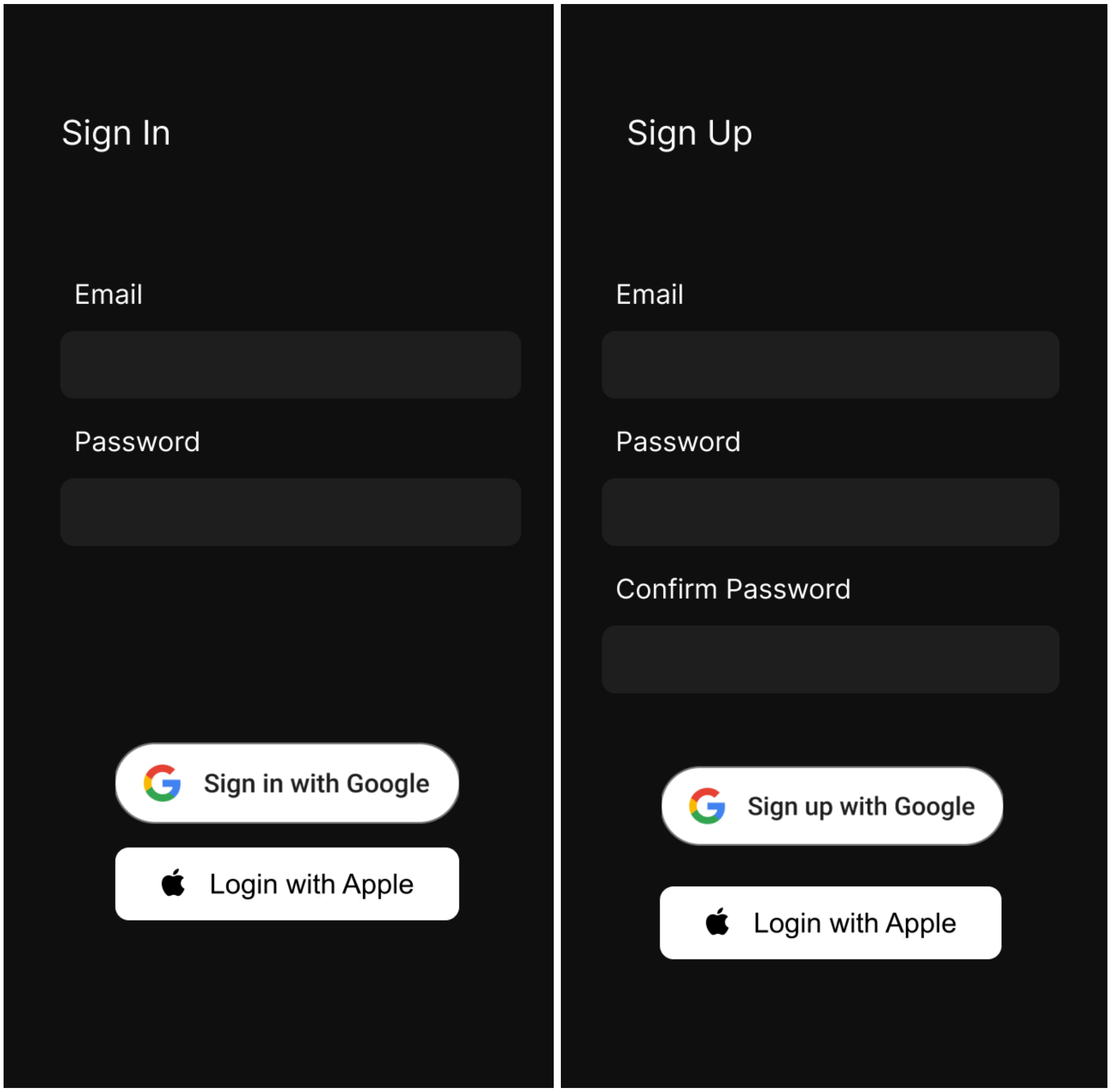
Northwestern Garage ☆
240/240

DSAI Lot ☆
32/38

Last Updated: 6:09PM EST



Parking Lot List and Map View



Sign In and Sign Up