Technical Specification Document

1. Overview

This project is a Flutter-based mobile application designed to help FIU students identify the most suitable parking garage for their next or upcoming in-person class. By analyzing class schedules, building locations, and garage availability, the system intelligently recommends parking options using a scoring algorithm that prioritizes proximity and space availability. Additionally, users have the option to manually sort garages by distance to class, distance from their current location, or number of available spaces for added flexibility.

2. Technology Stack

Layers	Tools
Design	Figma
Programming Language	Dart
UI Framework	Flutter
Data Format	Json
Location Calculation	Haversine formula
App Target	ISO / Android

3. System Architecture

The application is divided into four primary layers:

- **UI Layer**
  - main.dart: Entry point of the application
  - homepage.dart: Core interface for interacting with class and garage data
  - recommendation.dart: Handles the logic for scoring, sorting, and generating garage recommendations based on class schedule, user location, and real-time availability.
- **Model Layer**
  - garage.dart: Data model representing each parking garage
  - building.dart: Data model representing university buildings
  - class_schedule.dart: Data model representing class sessions and metadata
- **Services**
  - fetchUsers(): Calls the Class Schedule API to retrieve the current or upcoming class inperson for a student based on their ID.
  - fetchParking(): Calls the Parking API to fetch real-time availability data for all garages.
  - fetchBuilding(): Calls the Building API to retrieve a list of campus buildings and their coordinates.

- **Parsing Layer**
  - garage_parser.dart: Parses the API calls data into Garage objects
  - building_parser.dart: Parses the API calls data into Building objects
  - class_schedule_parser.dart: Parses the API calls data into ClassSchedule objects

- **Logic Layer**
  - Logic.dart: Core logic for calculating distances, availability, and generating ranked garage recommendations

4. Data Object Models

```
class Garage {
  String type, name;
  int studentSpaces, studentMaxSpaces;
  double latitude, longitude, score; // Calculated at runtime
}

class Building {
  String name;
  double latitude, longitude;
}

class ClassSchedule {
  String courseName, meetingTimeStart, meetingTimeEnd;
  String buildingCode, mode, subject, catalogNumber;
  String classSection, meetingDays, today, pantherId;
}   Course: $courseName
    Subject: $subject $catalogNumber - Section $classSection
    Days: $meetingDays
    Time: $meetingTimeStart - $meetingTimeEnd
    Location: $buildingCode
    Mode: $mode
    Today: ${today == 'true' ? 'Yes' : 'No'}
    ''';
  }
}
```

5. Data Parser

- **class_schedule_parser.dart:** Parses API class schedule data to find the student’s current or next upcoming in person class.
- **building_parser.dart:** Parses API building data and filters MMC campus buildings from the API, returning Building objects.
- **garage_parser.dart:** Parses API parking data and returns a filtered list of Garage objects.

6. Sorting Algorithm

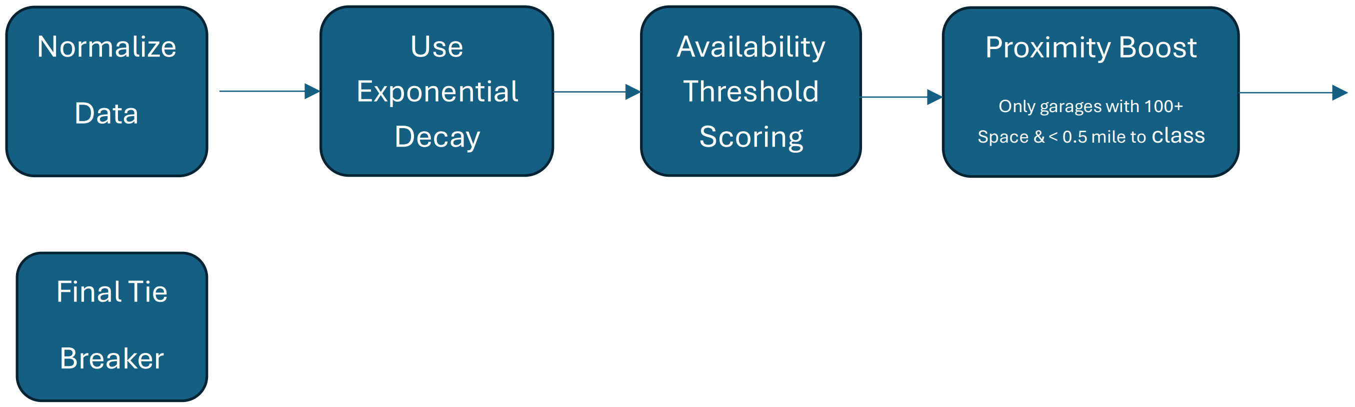
The Smart Parking app uses **an adaptive scoring system** to recommend the most optimal garages for FIU students based on proximity to their next in-person class.

STEPS

- **Get Class Data** : The system retrieves the student's current or upcoming class using the Class Schedule API. Only classes marked as "in-person" and occurring "today" are considered current garage availability, and their physical location.
- **Get Building Data**: The system extracts the building code from the matched class and uses it to fetch the corresponding building from the Building API. The building is then parsed and matched against MMC campus data to retrieve its coordinates, which are used to calculate walking distance to nearby garages.
- **Get Garage Data**: All garages are fetched from the Parking API and filtered to remove full or unavailable entries.
- **CALCULATE SCORE**: Each garage from getGarageData is scored based on distance to class, distance from the user, and available spaces. All values are normalized; distances use exponential decay, and availability is threshold-based garages with ≥100 spaces receive a full score. A proximity boost is applied to garages within 0.5 miles of the class building that also meet the availability threshold. These components are combined to calculate the final score and ranked using the **Adaptive Parking Rank (APR) Algorithm**.

Adaptive Parking Rank (APR) Algorithm Logic

Each garage is assigned a score based on: Distance to Class Building, Distance from User's Origin (Current Location), Available Parking Spaces


Normalize Data
Use Exponential Decay
Availability Threshold Scoring
Proximity Boost
Only garages with 100+ Space & < 0.5 mile to class
Final Tie Breaker

1) **Normalize** : Each scoring factor (class distance, origin distance, and available spaces) is normalized using the minimum and maximum values across all garages. For availability, if a garage has 100 or more available spaces, it is considered to have sufficient capacity and is assigned a score of 1.0. Garages with

fewer than 100 spaces are scaled proportionally. This ensures no factor unfairly dominates due to scale differences (e.g., large garages with thousands of spaces).

2) **Use exponential Decay:** For distance-based metrics (to class and to user origin), scores are calculated using an exponential decay rewards garages that are very close and sharply reduces the score as distance increases, aligning with how users prefer minimal walking distances.

3) **Adaptive scoring**: All three normalized components are combined using weighted factors.

4) **Proximity Boost**: Garages that are within 0.5 miles of the class building and have ≥100 available spaces receive a 1.5× score multiplier. This helps highlight highly convenient garages that meet both proximity and capacity needs.

5) **Final:** Garages are sorted in descending order by score. If two garages have the same score, we break ties by: Shorter distance to class then by Higher number of available spaces

Optional Sorting Filters

Users have the choice to manually sort by:

- Distance to class
- Distance from current location
- Number of available spaces