

Software Requirements Specification (SRS)

Smart Parking System

Authors

Aakarsh Verma (02)

Devansh Khodaskar (05)

Devyani Keché (06)

Dhruv Tambekar (07)

Gunjan Ghate (09)



Shri Ramdeobaba College of Engineering & Management, Nagpur 440 013

(An Autonomous Institute affiliated to Rashtrasant Tukdoji Maharaj Nagpur University Nagpur)

Table of Contents

1. INTRODUCTION

- 1.1. Purpose
- 1.2. Scope
- 1.3. Definitions, acronyms, and abbreviations
- 1.4. Overview

2. OVERALL DESCRIPTION

- 2.1. Product Perspective
- 2.2. Product Functions
- 2.3. User Characteristics
- 2.4. Constrains
- 2.5. Assumptions and Dependencies
- 2.6. Apportioning of requirements

3. REQUIREMENT SPECIFICATION

- 3.1. Function Requirements
 - 3.1.1. Performance Requirements
 - 3.1.2. Design Constraints
 - 3.1.3. Hardware Requirements
 - 3.1.4. Software Requirements
 - 3.1.5. Other Requirements

3.2. Non-Functional Requirement

3.2.1. Security

3.2.2. Reliability

3.2.3. Availability

3.2.4. Maintainability

3.2.5. Supportability

4. **DIAGRAM**

4.1. Use Case Diagram

4.2. Class Diagram

4.3. State Diagram

4.4. Sequence Diagram

4.5. Data flow Diagram

5. **PROTOTYPE**

5.1. Screen Shots

6. **REFERENCES**



Introduction

1. Introduction

This Software Requirements Specification (SRS) provides a detailed and structured documentation for the **Smart Parking Management System**, describing its [purpose, scope, definitions, and an overview](#) of the entire system. This project aims to solve real-world urban parking challenges using AI-driven automation and intelligent location-based services. The system includes machine learning models for space availability detection and prediction, location-aware parking recommendations, and a seamless online booking experience with QR-based authentication. The system integrates real-time mapping, booking, and user-friendly interaction features to enhance the parking experience, reduce congestion, and eliminate manual overhead. Both users and administrators will benefit from a reliable, fast, and predictive platform for parking space management.

1.1 Purpose:

The purpose of the **Smart Parking Management System** is to develop an intelligent platform that simplifies and automates the process of finding, reserving, and managing parking spaces. Users can view real-time parking availability, receive predictions on spot availability, and reserve slots remotely. Admins can manage lots efficiently with a centralized dashboard. The system uses machine learning for:

- Detecting available parking spaces via sensors or camera feeds.
- Predicting the probability of availability at a given time.

It also incorporates mapping services (HERE Maps, Leaflet, OSM) and geolocation for smart parking spot suggestions. Reservations include generating a unique QR code that users can scan to claim their booked spot.

1.2 Scope :

This system enables:

1. Real-time detection and display of parking availability.
2. Predictive analytics to estimate likelihood of spot availability.
3. Location-based discovery of nearest parking spaces using current location or manual input.
4. Booking of parking spots with time slots and vehicle number.
5. QR code-based access validation.
6. Admin-level control over parking slots, user history, and analytics.

The system will be accessible via both web and mobile platforms, making use of interactive maps and browser geolocation capabilities.

1.3 Definitions, Acronyms, and Abbreviations:

This should define all technical terms and abbreviations used in the document

Term	Definition
SRS	Software Requirements Specification
ML	National Train Enquiry System
OSM	Create, Retrieve, Update, Delete
Here Maps API	Interactive Voice Response System
QR Code	Passenger Name Record
UI	Data Flow Diagram
API	Entity Relationship Diagram
IOT	State Transition Diagram
GPS	Passenger Reservation System

1.4 Overview:

The subsequent sections of this Software Requirements Specification document provide a detailed and structured description of the Smart Parking Management System. Section 2 outlines the overall description of the system, including its context within the current urban mobility challenges, the characteristics of its intended users, the system environment, and a high-level view of the main features and operations.

Section 3 presents both functional and non-functional requirements, outlining the expected capabilities, performance metrics, and constraints the system must adhere to. Section 4 provides detailed diagrams such as use case, class, state, sequence, and data flow diagrams to visually represent system behavior, structure, and interactions. Section 5 showcases the prototype with annotated screen layouts to offer a clear understanding of the user interface and system flow.

Lastly, Section 6 lists all references and third-party technologies used throughout the development of the system.



Overall Description

2. Overall Description

This section describes the broader environment in which the Smart Parking Management System will operate. It outlines the key problems the current urban parking systems face, the stakeholders involved, their needs, and how the proposed system addresses these concerns using modern technology such as machine learning and location-aware services. The smart parking system is developed to replace traditional, inefficient, and manually managed parking setups with an AI-driven, real-time, and user-friendly digital platform.

2.1 Product Perspective:

The current parking systems in urban areas often suffer from limited visibility, unpredictable availability, and high dependency on manual management. Drivers waste considerable time and fuel searching for free spots, leading to traffic congestion and frustration. Furthermore, traditional systems lack dynamic updates, predictive insights, and digital convenience.

The proposed **Smart Parking Management System** addresses these issues by leveraging machine learning and geolocation technologies to automate the process of parking discovery, availability prediction, and space reservation. Unlike conventional systems, our platform offers real-time updates and predictive modeling using two ML models: one for determining the current status of parking slots, and another for estimating the probability of getting a spot during a future time window using regression analysis.

It integrates HERE Maps API, Leaflet, and OpenStreetMap for spatial services and utilizes the user's real-time location or manually selected coordinates to recommend the nearest and most suitable parking spots based on availability. The booking process is fully digital, enabling users to reserve a specific slot by selecting a time window and entering their vehicle number. Upon confirmation, a unique QR code is generated, which can be scanned to access the spot at the designated time.

This product is designed as a cross-platform application accessible via web browsers and mobile devices, and will connect to a centralized backend that handles ML predictions, booking validation, and QR code generation.

2.2 Product Functions:

The **Smart Parking Management System** provides a comprehensive set of features designed to enhance the user experience, optimize parking space utilization, and improve operational efficiency for administrators. Below is a summary of the core system functions:

1. **Detection of Available Parking Spots**

The system uses an integrated machine learning model to analyze sensor data or camera feeds (depending on implementation) and identify currently vacant parking slots in real-time.

2. **Prediction of Parking Spot Availability**

A second ML model, based on regression analysis, predicts the likelihood of parking space availability for a given time and location, helping users plan their parking in advance with higher confidence.

3. **Location-Based Spot Listing**

The system determines the user's location using browser geolocation, place search, or by manually dropping a pin on the map. Using HERE Maps API, Leaflet, and OpenStreetMap, it lists available parking spaces based on proximity and real-time availability.

4. **Parking Spot Booking**

Users can view detailed information about each parking lot and select a desired time slot for reservation. During booking, users provide their vehicle number and preferred slot duration. The system confirms the booking and generates a **unique QR code**.

5. **QR Code Generation and Validation**

Upon successful booking, a QR code is issued which acts as the user's digital ticket. This QR code is later used at the parking lot to verify the reservation and allow entry.

6. **User Account Management**

Users can register, log in, and manage their profiles. Booking history, active reservations, and notifications are accessible through a user-friendly dashboard.

7. **Admin Portal**

Administrators can manage parking spaces, view analytics on usage patterns, monitor occupancy data, and configure settings related to ML models, availability thresholds, and lot configurations.

8. **Search and Filter**

Users can search for parking lots using keywords, filters (e.g., availability, distance, price), and sorting options to quickly find suitable parking locations.

Each of these functions is tightly integrated to provide a seamless, predictive, and location-aware smart parking experience.

2.3 User Characteristics

The Smart Parking Management System is designed for a broad range of users with varying degrees of technical expertise. To ensure usability across the board, the system offers an intuitive, accessible interface with visual map-based interactions and guided booking flows.

Primary user groups include:

- **General Users (Drivers):**
These are everyday users who seek available parking spots in real time or want to reserve a space in advance. They are expected to have basic smartphone or computer literacy, including the ability to use a browser or mobile app, search for locations, and follow standard booking processes. Users are assumed to be familiar with QR code scanning and GPS-based services.
- **Administrators (Lot Operators / Managers):**
Admin users are responsible for managing individual parking lots or zones. They have access to dashboards that allow them to update spot availability, monitor usage patterns, view and verify QR code check-ins, and configure system parameters. Basic knowledge of system navigation, parking management, and moderate technical skills are expected.
- **System Modules / Services (ML models, APIs):**
These are non-human actors that operate in the background. They handle machine learning-based predictions, availability scanning, and location-to-listing logic via APIs and backend integrations.

The system ensures that all user interfaces are responsive, user-friendly, and accessible across devices to reduce friction and maximize adoption.

2.4 Constrains:

The development and deployment of the Smart Parking Management System will adhere to several constraints arising from hardware, software, integration, and operational dependencies. These constraints must be considered during system design and implementation:

- **Browser and Device Compatibility:**
The system relies on browser-based geolocation and mapping services. Therefore, it requires modern browsers that support the `getCurrentPosition()` function. Devices must also allow location access for optimal performance. Mobile devices with GPS are preferred for real-time tracking.

- **Internet Connectivity:**

As a web and cloud-based platform, the system requires a stable internet connection for users to view real-time availability, make bookings, and access map-based services.

- **Third-party API Limitations:**

Services like HERE Maps API, Leaflet, and OpenStreetMap are subject to rate limits, API key restrictions, and usage quotas. These limitations could affect the performance or availability of certain features if thresholds are exceeded.

- **QR Code Validation Dependence:**

QR code-based check-in assumes that parking lots have QR scanners or staff capable of scanning and validating codes. In the absence of scanning hardware, manual verification might be required.

- **Real-time Data Accuracy:**

ML-based detection of vacant slots depends on input from external sensors or image feeds, which may introduce inaccuracies if the input data is delayed or incomplete. Predictive analysis is probabilistic and may not guarantee 100% availability.

- **Privacy and Permissions:**

The system requires permission to access location data, which users may deny. In such cases, the user must manually input or select their location to proceed with listing nearby spots.

- **Device Resource Limitations:**

Low-end devices may experience delays while rendering real-time maps or generating QR codes, especially if multiple services (location, ML prediction, booking) run concurrently.

These constraints form critical checkpoints in the system's design, ensuring it performs reliably within its operational environment.

2.5 Assumptions and Dependencies:

The successful operation of the Smart Parking Management System relies on certain assumptions and external dependencies. These are essential for ensuring expected functionality and smooth integration with hardware, software, and third-party services.

Assumptions:

- Users have access to a device (smartphone, tablet, or computer) with internet connectivity.
- Users will grant permission to access location services when prompted, or will manually input their coordinates or search for a location.
- Parking lot operators have access to a backend system and QR code verification tools (scanner or app) to validate bookings.
- Parking lot infrastructure is equipped with sensors or input feeds (e.g., cameras, IoT devices) to provide real-time availability data to the ML model.

Dependencies:

- The system depends on third-party APIs such as:
 - **HERE Maps API** for location search and geocoding.
 - **Leaflet + OpenStreetMap** for interactive map rendering.
- Machine learning models used for:
 - **Real-time parking slot detection** (e.g., image or sensor input classification).
 - **Availability prediction** using regression based on historical and real-time data.
- Backend services are responsible for handling:
 - Location-to-parking space mapping.
 - ML model inference.
 - Booking storage and QR code generation.
- External services (email, SMS APIs) may be required for sending booking confirmations or QR codes to users.

Failure or unavailability of any of these components may result in partial or degraded system functionality.



Requirement Specification

3. Function Requirements

3.1 Functional Requirement:

The Smart Parking Management System is structured around several core modules that interact with each other to provide a seamless user experience. These modules define the main operations of the system, ensuring a fully functional parking ecosystem.

3.1.1 User Module:

- **User Registration & Login**
Users can register using email or phone, and log in securely to access the system. OAuth or token-based authentication is supported.
- **Profile Management**
Users can update personal info, vehicle details, and view booking history.
- **Geolocation Access**
System prompts the user to share their real-time location or allows input via search bar or pin drop.

3.1.2 Parking Spot Discovery Module:

- **Real-Time Availability Detection**
A backend ML model analyzes input from sensors or image feeds to determine vacant or occupied spots.
- **Proximity-Based Listing**
Based on coordinates (via GPS, search, or map pin), the system lists nearby parking lots, sorted by distance and current availability.
- **Availability Prediction**
The second ML model predicts the chance of getting a spot based on historical data and real-time inputs.

3.1.3 Booking and Payment Module:

- **Slot Reservation**
Users select a parking lot, specify time (e.g., 2:00–3:00 PM), and provide their vehicle number.
- **QR Code Generation**
On successful booking, a unique QR code is generated and sent to the user. This code is required to claim the parking slot.
- **Booking Validation**
Admins or gate staff scan the QR code to validate the user's slot and grant entry.

3.1.4 Admin Module:

- **Parking Lot Management**
Admins can add, edit, or remove parking zones and spots.
- **Live Monitoring**
Admin dashboard shows real-time occupancy and upcoming reservations.
- **QR Code Verification Tool**
Admin panel includes scanner tools for validating user QR codes on entry.

3.1.5 Notification Module:

- **Email or SMS Confirmation**
Booking details and QR code are sent to the user's registered contact method.
- **Alerts for Expiring Slots**
Notifications warn users as their reserved time nears its end, to avoid overstaying.

3.2 Performance Requirement

3.2.1 Performance:

The Smart Parking Management System is designed to deliver high performance and responsiveness across all devices and use cases. The following performance requirements outline how the system is expected to behave under typical and peak conditions:

- **Fast Response Time:**
All user interactions — including map loading, location detection, booking actions, and QR code generation — must respond within **2 seconds** under normal network conditions.
- **Scalability:**
The backend system should be able to handle simultaneous usage by up to **10,000 concurrent users** without performance degradation. It must support scalable architecture using cloud-based infrastructure.
- **Real-Time Updates:**
Availability status of parking spaces must refresh at a minimum interval of **every 15 seconds**, ensuring up-to-date visibility for users.
- **ML Model Inference Speed:**
Predictions (for availability probability and current occupancy detection) should complete within **1.5 seconds** after request submission.
- **Data Synchronization:**
All changes made by admins or triggered by sensors (e.g., a spot becomes vacant) must reflect across all user interfaces in **real time** or with minimal delay (<5 seconds).
- **Cross-Platform Optimization:**
The system will perform consistently across web browsers (Chrome, Firefox, Safari, Edge) and mobile browsers, with adaptive UI for various screen sizes.
- **System Uptime:**
The system must maintain an uptime of **99.5%** per month to ensure reliable access for both users and administrators.

3.2.2 Design Constraints:

The Smart Parking Management System must conform to a range of design constraints arising from integration requirements, platform limitations, and industry-standard security protocols.

These constraints guide how the system is built and deployed:

- **Compliance with Web Standards:**

All frontend components must follow HTML5, CSS3, and JavaScript (ES6+) standards to ensure cross-browser compatibility and maintainability.

- **Responsive and Mobile-First Design:**

The system must adopt a responsive design framework (such as Tailwind CSS or Bootstrap) to ensure a smooth experience across desktops, tablets, and smartphones.

- **Hardware Integration Constraints:**

The system may rely on parking lots having IoT devices, cameras, or QR scanners. Lack of uniform hardware across locations may affect data input quality or user experience.

- **ML Model Deployment Constraints:**

The trained machine learning models must be containerized (e.g., Docker) and served via RESTful APIs. They must run efficiently on cloud instances with limited GPU access.

- **Security Standards Compliance:**

The system must enforce SSL encryption, JWT token-based authentication, and OAuth 2.0 standards for secure login and session management.

- **Rate Limiting and API Quotas:**

Third-party services (e.g., HERE Maps, Leaflet, and OSM) are subject to API call limits. The system must implement caching and fallback logic to avoid service disruptions.

- **Data Privacy and Protection:**

All personal user data (e.g., location, vehicle number) must be stored securely in compliance with GDPR or other relevant privacy frameworks. These constraints are to be factored into every phase of development — from UI design to backend service orchestration — to ensure reliability, scalability, and regulatory compliance.

3.2.3 Hardware Requirements:

The Smart Parking Management System will operate in both cloud-hosted environments and client-side devices such as user smartphones or admin desktops. The hardware requirements below apply primarily to the **server infrastructure** and **admin-side systems**. User-side access is designed to work on standard mobile and desktop devices.

Minimum Hardware Requirements (Server & Admin Panel Use)

- **Processor:** Dual-Core Intel/AMD (2.0 GHz or higher)
 - **RAM:** 4 GB
 - **Hard Drive:** 100 GB SSD
 - **Network:** 10 Mbps dedicated internet connection
 - **Operating System:** Windows 10 / Ubuntu 18.04+ / macOS 10.13+
 - **Display Resolution:** 1280x720 or higher
 - **Additional:** QR code scanner device (USB or mobile-based) for admin validation
-

Preferred Hardware Requirements

- **Processor:** Quad-Core Intel i5/i7 or equivalent AMD Ryzen
 - **RAM:** 8–16 GB
 - **Hard Drive:** 250 GB SSD
 - **Network:** 50 Mbps or higher (Fiber recommended for cloud sync)
 - **Operating System:** Latest LTS version of Ubuntu / Windows 11
 - **GPU (Optional):** NVIDIA T4 or equivalent for ML inference acceleration
 - **Additional:** Barcode/QR scanner integration module, UPS for backup
-

Note: For user devices (web/mobile access), there are no strict hardware requirements. Any

smartphone or computer with a modern browser and internet access can fully operate system.

3.2.4 Software Requirements:

The Smart Parking Management System is built using a combination of modern web technologies, cloud infrastructure, and machine learning frameworks. The following software components are required for development, deployment, and runtime operations.

Development Environment

- **Frontend Framework:** React.js (with Tailwind CSS or Bootstrap for UI)
 - **Backend Framework:** Node.js (Express) or Django (Python-based backend)
 - **Database:** PostgreSQL or MongoDB
 - **Machine Learning Environment:**
 - Python 3.8+
 - TensorFlow or Scikit-learn
 - Flask or FastAPI for model serving (REST APIs)
 - **Mapping & Location Tools:**
 - HERE Maps API
 - Leaflet.js
 - OpenStreetMap
 - **QR Code Generator Library:**
 - **qrcode** (JavaScript/Python)
 - **Version Control:** Git (GitHub or GitLab)
 - **IDE/Tools:** VS Code / PyCharm / Postman for API testing
-

Deployment Environment

- **Operating System:** Ubuntu 20.04 LTS (Preferred), Windows Server (Optional)
- **Containerization:** Docker (for deploying ML models and backend services)

- **Web Server:** NGINX or Apache
 - **Cloud Hosting:** AWS EC2 / Azure / DigitalOcean
 - **Storage:** Amazon S3 (for QR codes or other static content)
 - **CI/CD Tools:** GitHub Actions or Jenkins
-

Client-Side Requirements

- **Browser Compatibility:**
 - Google Chrome (v90+)
 - Mozilla Firefox (v88+)
 - Microsoft Edge (v91+)
 - Safari (iOS 13+)
- **Mobile OS Compatibility:**
 - Android 8.0+
 - iOS 13+

3.2.4 Other Requirements:

To ensure that the Smart Parking Management System is robust, secure, and adaptable for future needs, it must meet the following general software quality requirements:

- **Security:**

The system must ensure data protection through end-to-end encryption (SSL/TLS), secure authentication (JWT tokens), and role-based access controls. All user data — especially location, booking history, and personal identifiers — must be securely stored and handled in compliance with data protection laws.
- **Portability:**

The software must be platform-independent. It should be easily deployable across various environments (Windows, Linux, MacOS) and cloud platforms (AWS, Azure, etc.) without major reconfiguration.

- **Correctness:**
The system must always reflect accurate, real-time data on parking slot status, predictions, and booking availability. Data validation checks should be enforced at both frontend and backend layers.
- **Efficiency:**
Backend services and ML models must be optimized for quick responses. Redundant API calls should be minimized, and caching mechanisms should be in place for frequently accessed data like map tiles and static lot info.
- **Flexibility:**
The architecture must support future expansion — such as adding more cities, integrating dynamic pricing, or deploying a native mobile app — without rewriting core components.
- **Testability:**
Each module (frontend, backend, ML models, APIs) should have unit and integration tests. CI pipelines should automatically run test suites on each commit to maintain stability.
- **Reusability:**
Components like the QR code module, booking system, and location service modules should be loosely coupled and designed for reuse in future projects or spin-off products.

3.3 Non-Functional Requirements

3.2.1 Security:

The Smart Parking Management System places a high priority on secure communication and data integrity. All communication between clients and servers must be encrypted using HTTPS protocols with SSL/TLS to prevent interception or tampering. User authentication is handled using JSON Web Tokens (JWT), ensuring secure and tamper-proof session management. Passwords are never stored in plain text but instead are securely hashed using algorithms like bcrypt. For sensitive operations such as booking confirmation and admin access, additional layers of verification such as token validation and optional two-factor authentication (2FA) are enforced. Role-based access control ensures that users cannot access administrative or restricted functionalities. Furthermore, the system includes rate-limiting, intrusion detection, and monitoring to protect against abuse or DDoS attacks. To safeguard user sessions, automatic logout is triggered after a period of 15 minutes of inactivity.

3.2.2 Reliability:

Reliability is essential for a real-time, user-facing system like this one. The system is designed to handle partial failures gracefully by falling back to cached data or estimated values when sensor feeds or external services become temporarily unavailable. Critical data, including booking records and availability metrics, is backed up incrementally every 15 minutes, with full backups occurring every 6 hours to ensure data durability and fast recovery in the event of a failure. In case any core service—such as the machine learning model or location API—fails or does not respond within a predefined time frame, the system will notify administrators immediately through email alerts, and services will default to safe fallback modes with user warnings.

3.2.3 Availability:

The system must be available to users 24/7, and must maintain a minimum uptime of 99.5% monthly. To achieve this, it uses a cloud-based infrastructure with load balancing and auto-scaling capabilities to handle traffic spikes, especially during peak hours or special events. In the event of hardware failure or database issues, users are redirected to a fallback page that provides basic functionality and a clear error message, ensuring minimal disruption. Redundancy is built into all critical services, especially for features like QR validation and booking, to guarantee high availability even during maintenance or updates.

3.2.4 Maintainability:

Maintainability is ensured through the use of modular architecture, allowing different components of the system—such as user management, ML services, and booking engine—to be developed, tested, and updated independently. Code quality is enforced through consistent standards, and every major module is version-controlled and fully documented. Log data is collected and centralized for efficient debugging and auditing, while admin dashboards allow real-time visibility into system health, parking slot usage, and prediction accuracy. This design enables quick identification of issues and minimal downtime during updates.

3.2.5 Supportability:

To support long-term scalability and ease of handover, the system includes thorough documentation of all modules, deployment procedures, and developer workflows. Inline help, onboarding tutorials, and tooltips are integrated into the user interface to guide new users. Machine learning models are built using reproducible and versioned pipelines, with proper documentation of training data, hyperparameters, and performance metrics. Deployment is containerized using Docker, allowing DevOps teams to quickly set up or replicate environments across staging and production servers. These measures ensure that both technical and non-technical stakeholders can easily manage, support, and evolve the system over time.



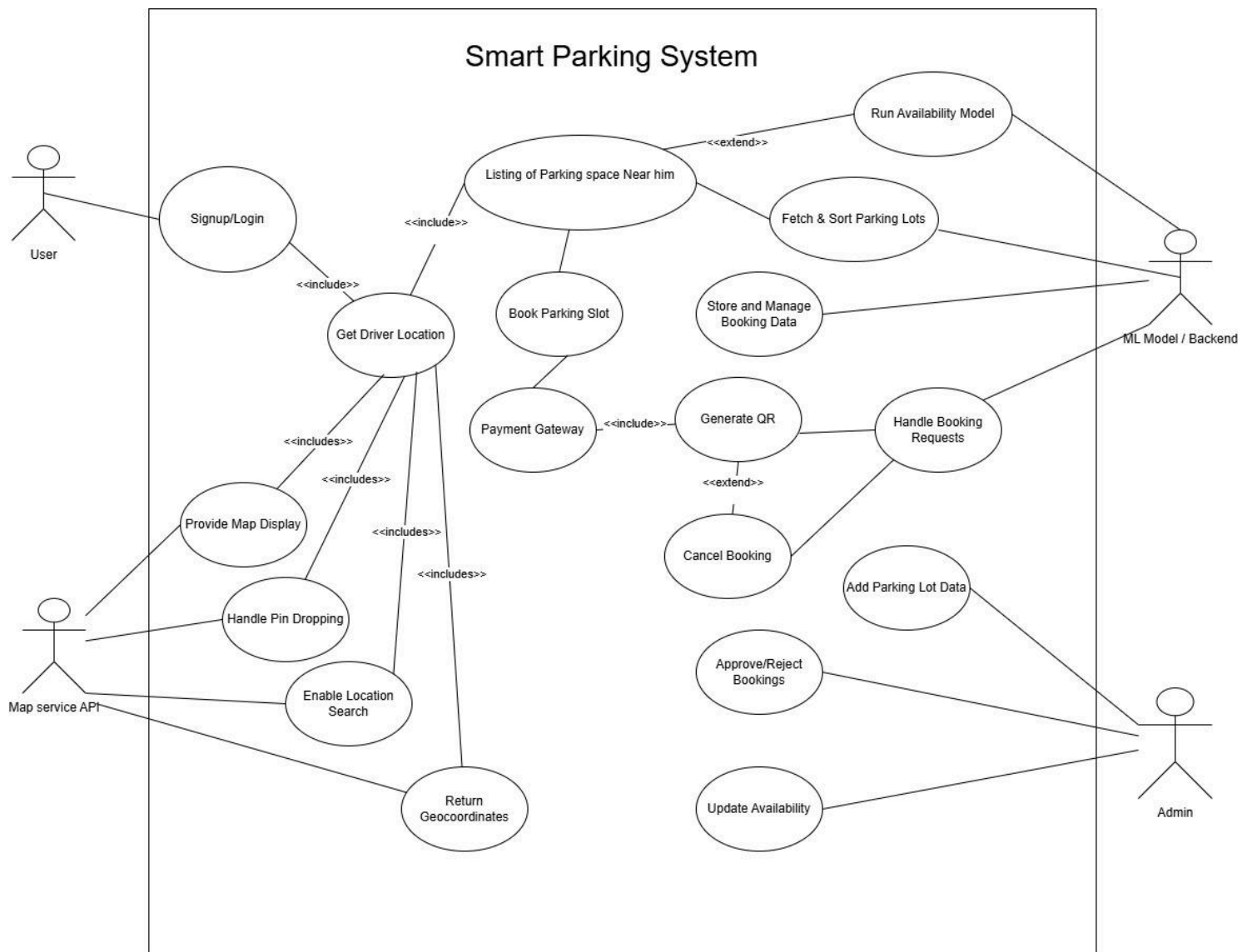
Diagram

4.0 Use Case Diagram

A **Use Case Diagram** is a behavioral UML diagram that represents the functional requirements of a system from the user's perspective. It shows the interactions between **actors** (users or external systems) and the **use cases** (functions or services) the system provides. Each use case represents a specific functionality or goal that the system performs in response to a user's action.

Actors are shown as stick figures, while use cases are represented as ovals. Relationships between them include **associations**, **includes**, **extends**, and **generalizations**. “Include” is used when a use case always uses another, while “extend” is for optional or conditional behavior.

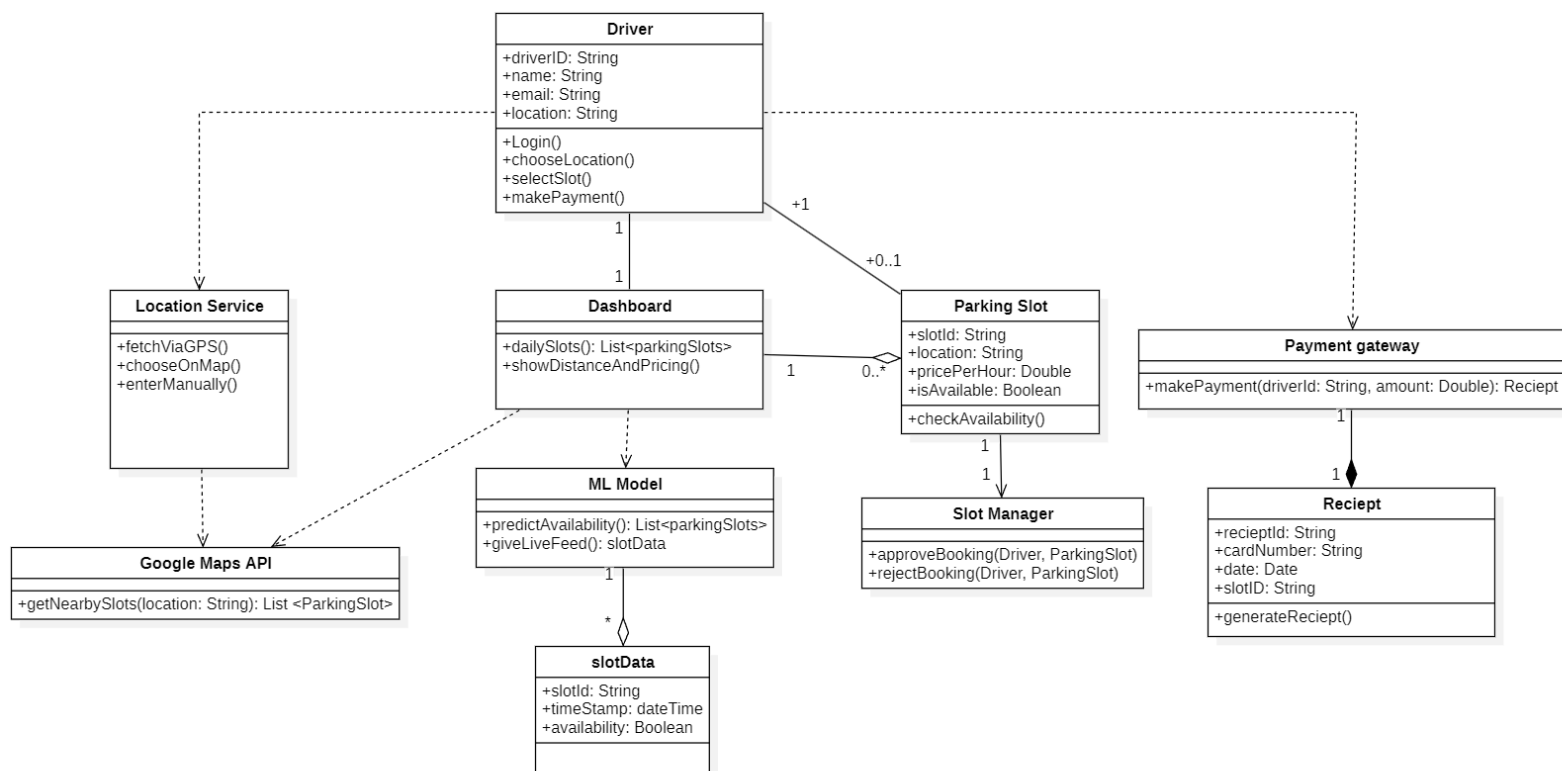
Use case diagrams help stakeholders understand what the system is supposed to do and clarify system boundaries. They are essential during the requirement-gathering phase and guide system design by ensuring all user interactions are identified and planned for. This makes them a key tool for communication between developers and clients



4.1 Class Diagram

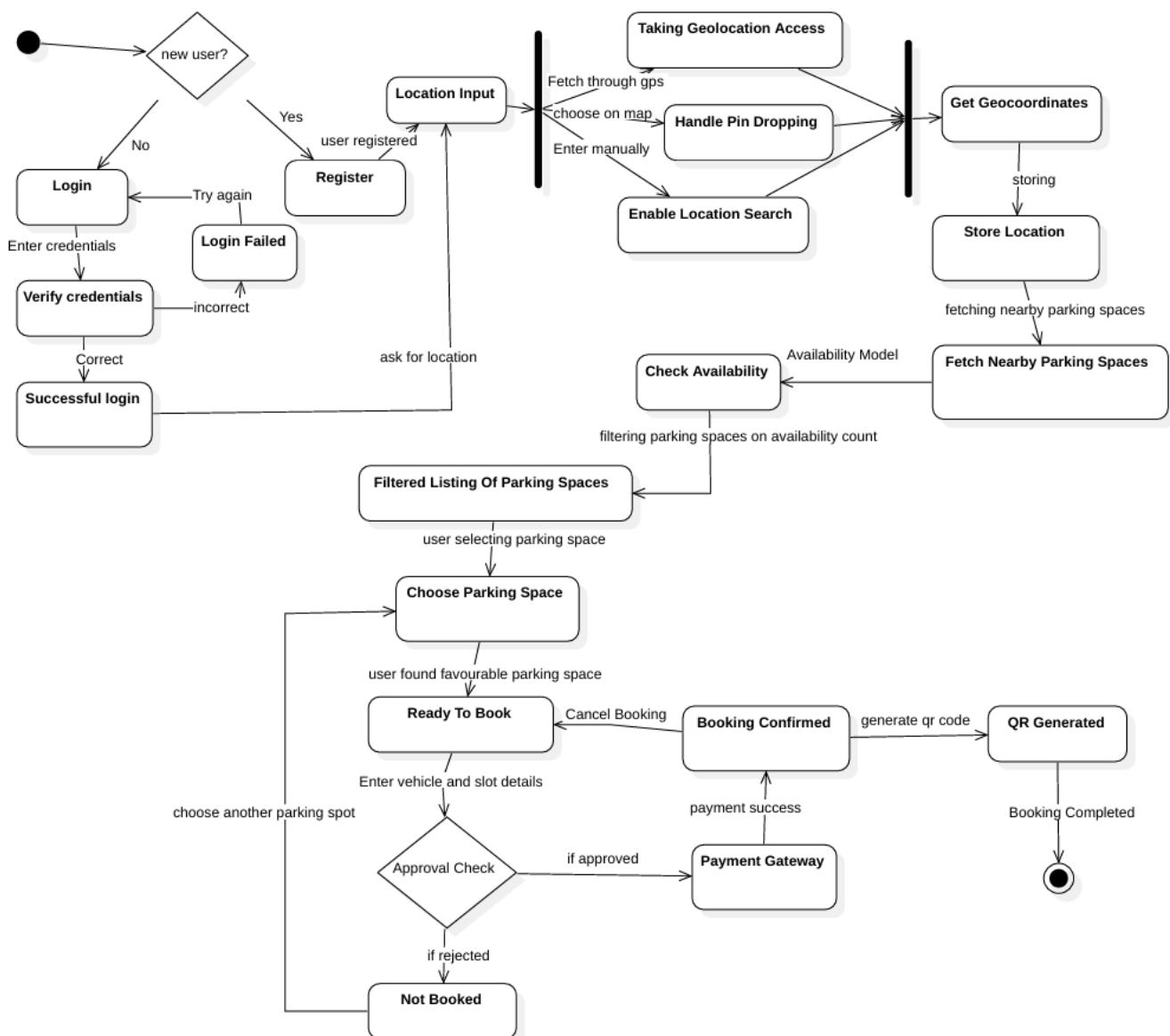
A **Class Diagram** is a type of static structure diagram used in software engineering and UML (Unified Modeling Language) to represent the structure of a system. It shows the system's classes, their attributes, methods (operations), and the relationships among objects. Each class is represented by a rectangle divided into three parts: the class name, attributes, and operations. Class diagrams help developers visualize and design the blueprint of object-oriented systems before coding begins.

Relationships like association, aggregation, composition, and inheritance depict how classes interact with one another. Association shows a general connection, while inheritance shows that one class is derived from another. Aggregation and composition represent whole-part relationships, with composition implying ownership. Class diagrams play a key role in object-oriented analysis and design by simplifying complex systems into manageable structures. They also aid in communication among team members and act as a reference throughout the development process.



4.2 State Diagram

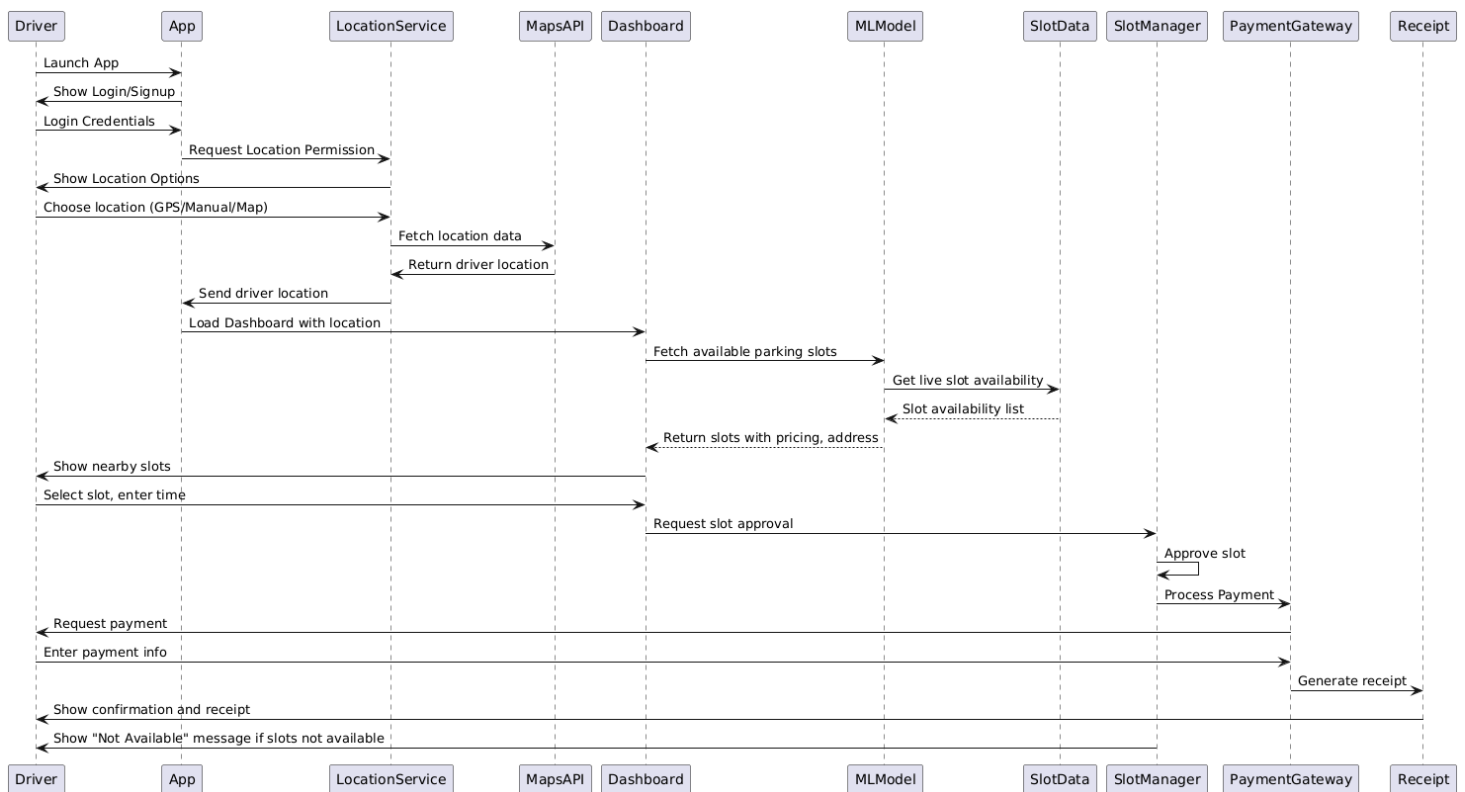
A **State Diagram** is a behavioral diagram in UML used to represent the dynamic behavior of a system by showing its different states and the transitions between those states. It illustrates how an object responds to events by changing from one state to another. Each state represents a condition or situation during the life of an object where it satisfies some condition, performs an activity, or waits for an event. Transitions are triggered by events and may include actions. A state diagram typically starts with an initial state (shown as a filled black circle) and ends with a final state (depicted as a bullseye symbol). State diagrams are especially useful for modeling the behavior of reactive systems, like vending machines, traffic lights, or user interfaces, where the system responds to external inputs. They help in understanding object lifecycles and are used in both system design and documentation.



4.3 Sequence Diagram

A **Sequence Diagram** is a type of interaction diagram in UML that shows how objects interact in a particular scenario of a use case. It focuses on the sequence of messages exchanged between objects over time. The diagram is arranged vertically with time progressing downward, and horizontally with objects (or participants) represented as lifelines. Each lifeline has a rectangle at the top showing the object name and a dashed line extending downward to represent its existence over time.

Messages between objects are shown as arrows, with solid arrows indicating synchronous calls and dashed arrows for return messages. Activation bars (thin rectangles on lifelines) show when an object is active or performing an operation. Sequence diagrams help developers understand how components collaborate to achieve a specific functionality, making them useful for both design and documentation. They are especially helpful in identifying object interactions, timing issues, and responsibilities during software development.



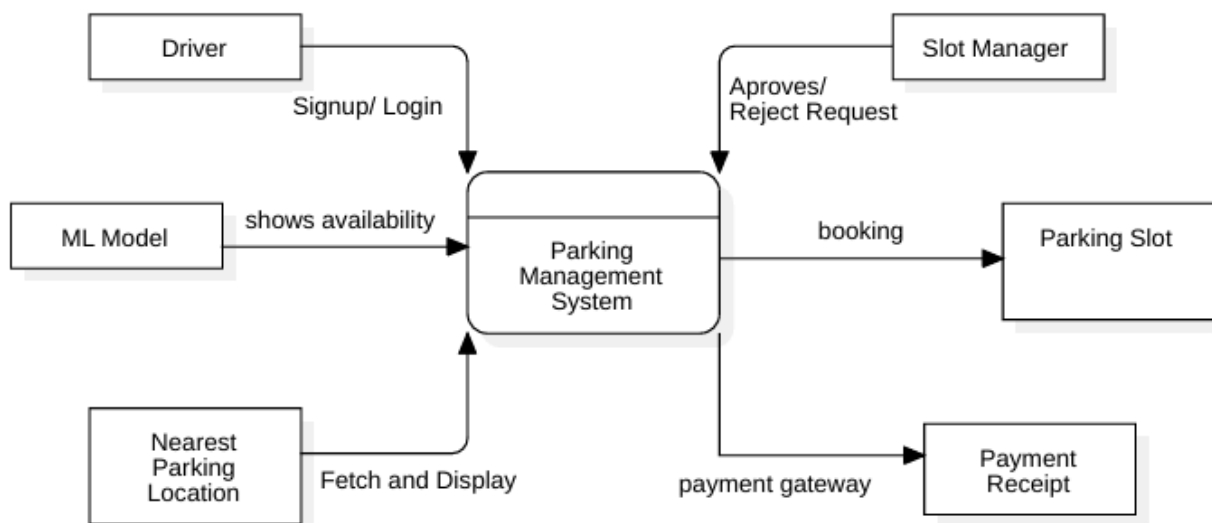
4.4 Data Flow Diagram

A **data flow diagram (DFD)** is a graphical representation of the "flow" of data through an information system. DFDs can also be used for the visualization of data processing (structured design). On a DFD, data items flow from an external data source or an internal data store to an internal data store or an external data sink, via an internal process. A DFD provides no information about the timing of processes, or about whether processes will operate in sequence or in parallel. It is therefore quite different from a flowchart, which shows the flow of control through an algorithm, allowing a reader to determine what operations will be performed, in what order, and under what circumstances, but not what kinds of data will be input to and output from the system, nor where the data will come from and go to, nor where the data will be stored (all of which are shown on a DFD).

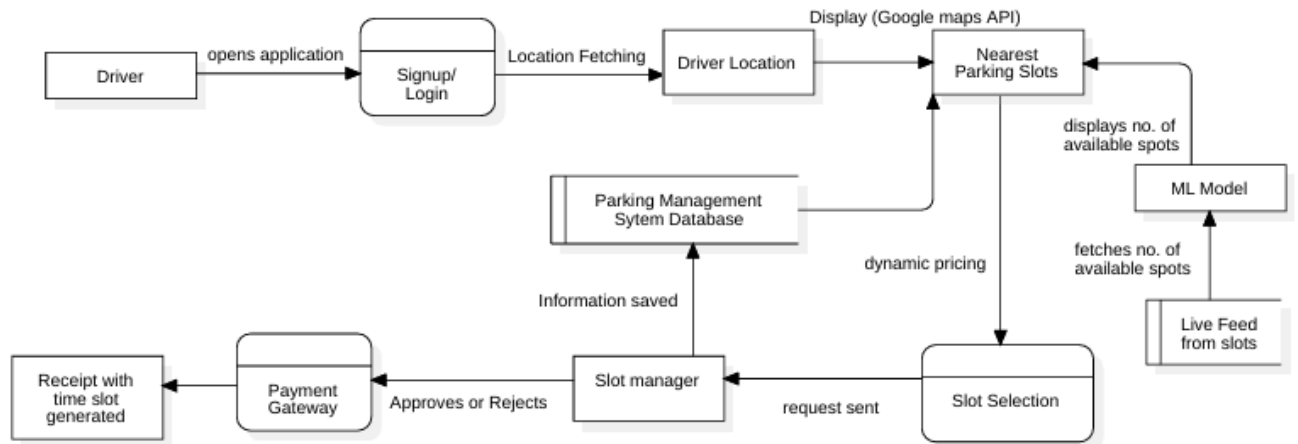
It is common practice to draw a context-level data flow diagram first, which shows the interaction between the system and external agents which act as data sources and data sinks. On the context diagram (also known as the 'Level 0 DFD') the system's interactions with the outside world are modelled purely in terms of data flows across the *system boundary*. The context diagram shows the entire system as a single process, and gives no clues as to its internal organization.

This context-level DFD is next "exploded", to produce a Level 1 DFD that shows some of the detail of the system being modelled. The Level 1 DFD shows how the system is divided into subsystems (processes), each of which deals with one or more of the data flows to or from an external agent, and which together provide all of the functionality of the system as a whole. It also identifies internal data stores that must be present in order for the system to do its job, and shows the flow of data between the various parts of the system.

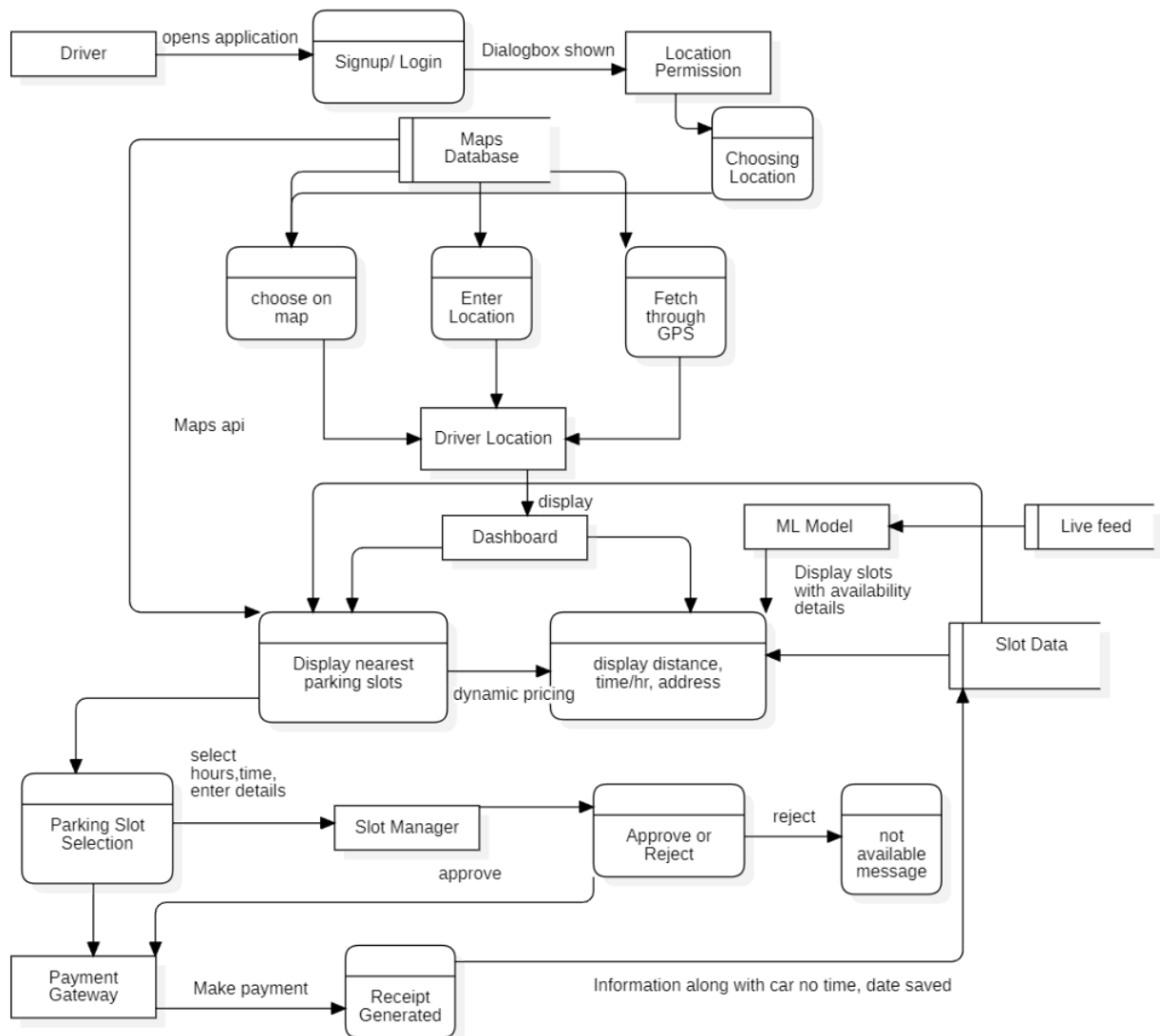
Level 0:



Level 1:



Level 2:





Prototype

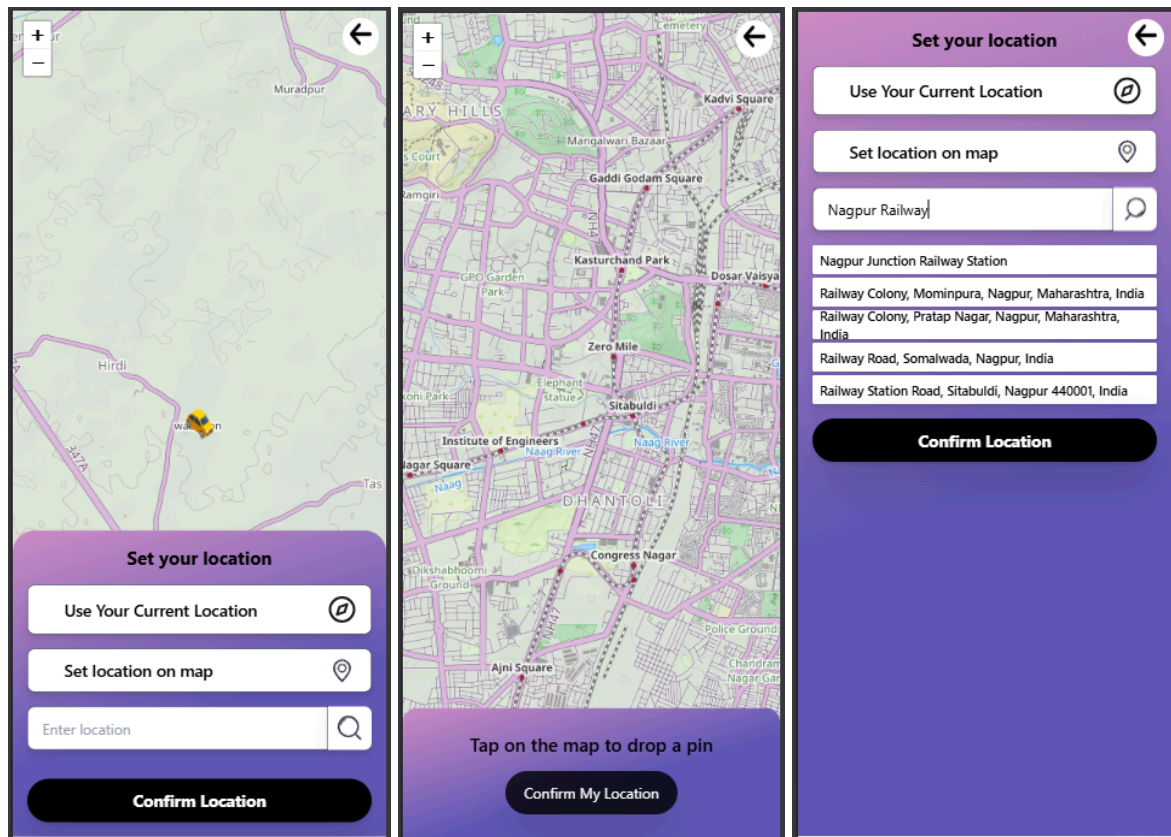
1.1 Screenshot

The system shall provide a uniform look and feel between all the web pages.

Home Page:



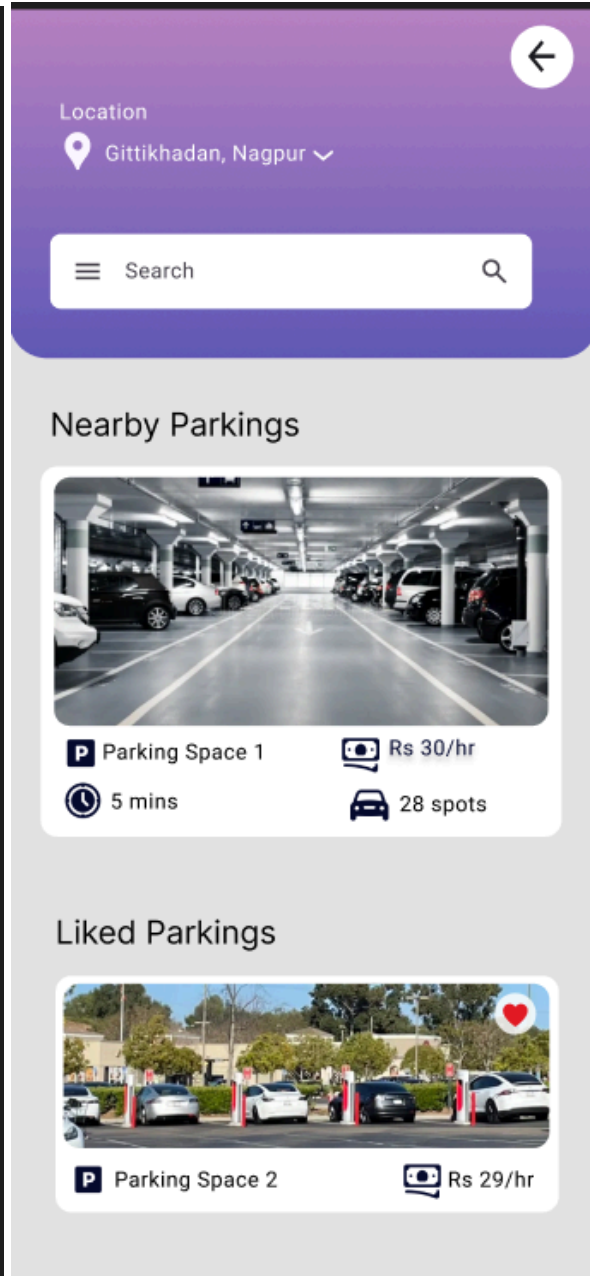
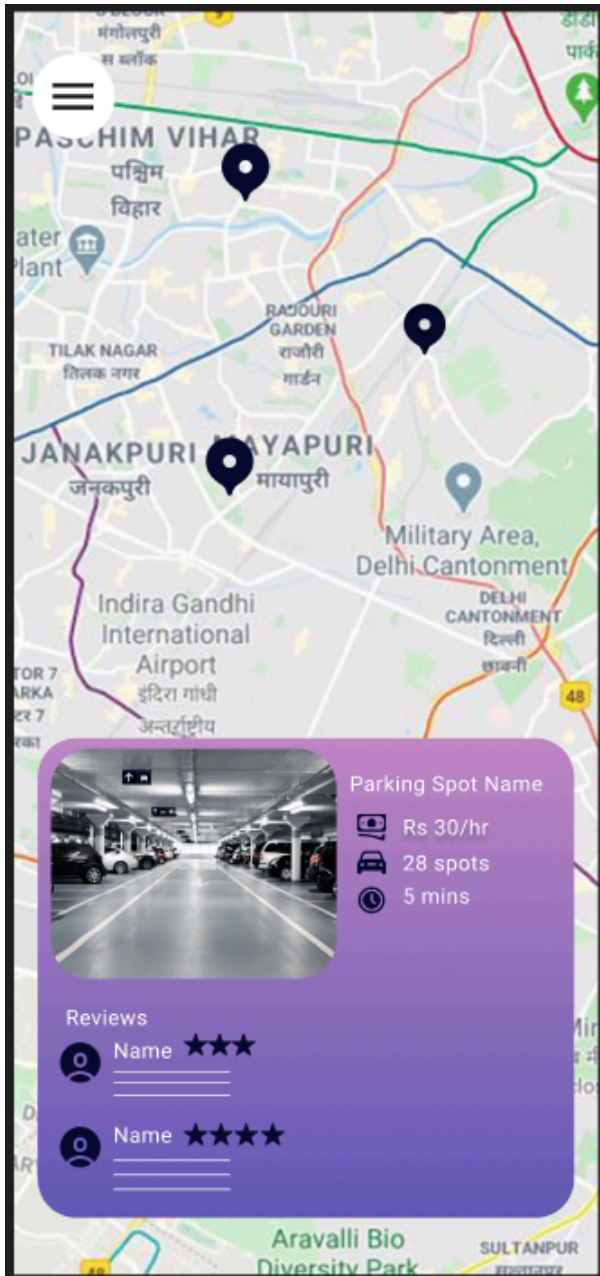
Asking For Location:




Checking Availability:



Selecting Parking Spaces:



Other Screen:





Hi, User!


Where do you wanna park today?


Set your Location


Recently booked





 Parking Spot A

 Rs 30 Rs 15

 2 km

 10 spots

 Location A

 5 spots

Cancel booking

Generate Your Parking Ticket


John Doe

4:00 PM - 5:00 PM

Generate Ticket

Parking Ticket

Name: John Doe
Date: 4/22/2025, 11:56:53 PM
Time Slot: 4:00 PM - 5:00 PM



Ticket ID: 399652



Appendices

Appendix A:

Glossary of Terms and Acronyms

This section defines technical terms and acronyms used throughout this document:

- **SRS (Software Requirements Specification)**: A comprehensive description of the intended functionality, behavior, and constraints of the software system.
- **ML (Machine Learning)**: A branch of artificial intelligence that enables systems to learn from data and make predictions or decisions without being explicitly programmed.
- **API (Application Programming Interface)**: A set of protocols and tools that allow different software components to communicate with each other.
- **QR Code (Quick Response Code)**: A two-dimensional barcode that encodes information such as booking ID, enabling users to verify their parking slot.
- **GPS (Global Positioning System)**: A satellite-based system used for determining the precise location of a device.
- **OSM (OpenStreetMap)**: A collaborative project that provides free editable maps of the world, used for geolocation and routing.
- **HERE Maps**: A mapping and location services platform that provides APIs for geolocation, navigation, and place searches.
- **UI (User Interface)**: The space where users interact with the system through graphical elements such as maps, forms, and buttons.

Appendix B:

Machine Learning Models Description

The system uses two machine learning models:

Model 1: Real-time Parking Space Detection

This model utilizes sensor or camera data to detect available parking spaces. It uses object detection techniques to identify vacant and occupied slots in real-time. The input includes live video feeds or image snapshots. The model is trained using labeled datasets of parking lots and achieves accuracy above 90% in controlled environments. The model is periodically retrained and updated weekly based on new data inputs.

Model 2: Parking Availability Prediction (Regression Model)

This model forecasts the probability of a parking slot being available in the near future using historical usage trends and current sensor data. It is based on regression techniques trained on timestamped occupancy records. Inputs include time of day, day of week, past occupancy, and location. The model supports dynamic adjustments and is updated daily with new slot activity data.

Appendix C:

Third-Party APIs and Services

The system integrates several third-party services:

- **Maps API:** Used for place searches, routing, and reverse geolocation based on user's coordinates. It requires API key authentication and adheres to rate limits as per the service provider's policy.
- **Leaflet + OpenStreetMap (OSM):** Used for rendering interactive maps where users can drop pins or visually explore parking lots.
- **Email and SMS Services:** Notifications for booking confirmations are sent via email and optionally through SMS APIs, which require user consent and verified contact information.
- **Fallback strategies** are in place in case third-party APIs are unavailable, such as defaulting to cached map data or disabling certain location features with user alerts.

Appendix D:

System Constraints and Assumptions

This appendix outlines limitations and assumptions:

- The system must be compatible with modern browsers (Chrome, Firefox, Safari) and responsive across desktop and mobile devices.
- A stable internet connection is required for real-time map updates, ML predictions, and booking confirmation.
- QR code scanning assumes the availability of either a smartphone or scanner at the parking lot entrance.
- Accuracy of ML predictions depends on the quality of input data (camera resolution, sensor calibration).
- Users must permit browser-based geolocation for seamless map and navigation features. If declined, the user can manually enter a location or drop a pin.
- It is assumed that the infrastructure has basic power and network access, and that users behave according to standard booking norms.

Appendix E:

User Interface Screenshots

The following prototype screens are included for demonstration purposes:

- Map Interface showing real-time parking availability, color-coded by slot status.
- Booking Page where users select time, enter vehicle number, and confirm payment.
- QR Code Generation screen showing the unique code to be scanned on arrival.
- Admin Dashboard with options to view occupancy reports, scan QR codes, and manually override slot status.

Appendix F:

Diagrams

This section supplements Section 4 with visual models:

- **Use Case Diagram:** Illustrates interactions between system and users.
- **Class Diagram:** Depicts system entities such as User, Slot, Booking, etc.
- **State Diagram:** Shows slot states (Vacant, Reserved, Occupied, Expired).
- **Sequence Diagram:** Describes booking process flow.
- **Data Flow Diagram (DFD):** Outlines flow of data between frontend, backend, and ML components.

6. References

1. HERE Maps API Documentation: <https://developer.here.com/>
2. OpenStreetMap Documentation: <https://wiki.openstreetmap.org/>
3. Scikit-learn Regression Guide: https://scikit-learn.org/stable/modules/linear_model.html
4. Leaflet.js Documentation: <https://leafletjs.com/>
5. Python QR Code Library: <https://pypi.org/project/qrcode/>
6. Academic references for ML models and public parking datasets (to be listed if used)