3.5.1 Architectural Design Strategy

Introduction

Choosing an architectural design strategy is crucial for the success of the Smart Parking System. The main strategies include decomposition, design based on quality requirements, and generating test cases. After careful consideration of the project's requirements and constraints, the **decomposition strategy** is the most suitable for our system. Here's a detailed justification for this choice:

Decomposition Strategy

Decomposition involves breaking down the system into smaller, manageable components, each responsible for a specific function. This strategy aligns well with the complexity and modular nature of the Smart Parking System, which includes multiple subsystems such as user interfaces, backend services, sensor integration, and database management.

Justification:

1. **Modularity and Maintainability**:
    - By decomposing the system, we can create modular components that are easier to understand, develop, and maintain. This modularity allows developers to focus on specific parts without being overwhelmed by the entire system's complexity.
    - For example, the user interface, backend services, and database interactions can be developed and tested independently.
2. **Parallel Development**:
    - Decomposition facilitates parallel development, where different teams can work on various components simultaneously. This accelerates the development process and improves productivity.
    - For instance, the front-end team can work on the web and mobile interfaces, while the back-end team focuses on APIs and database integration.
3. **Reusability**:
    - Components developed through decomposition can be reused in different contexts or future projects, saving development time and ensuring consistency.
    - For example, the user authentication module can be reused across different applications with minimal modifications.
4. **Scalability**:
    - Decomposing the system into smaller parts makes it easier to scale specific components independently. This is essential for handling a large number of concurrent users and parking lots.

    - For example, if the demand for real-time slot availability increases, we can scale the relevant backend services without affecting other components.
5. **Ease of Maintenance and Troubleshooting**:
    - When issues arise, it is easier to identify and isolate the problematic component, making debugging more efficient.
    - For instance, if there is a problem with the payment integration, it can be addressed without impacting the rest of the system.
6. **Adaptability**:
    - The decomposition strategy allows for flexible updates and modifications. As user requirements evolve, specific components can be updated without extensive changes to the entire system.
    - For example, adding new features like predictive analytics or dynamic pricing can be done by enhancing relevant modules.

Comparison with Other Strategies

1. **Design Based on Quality Requirements**:
    - While this strategy focuses on meeting specific quality attributes such as performance, security, and scalability, it might not provide the same level of modularity and parallel development benefits as decomposition.
    - It can lead to a more rigid structure, making it challenging to adapt to changing requirements or integrate new features.
2. **Generating Test Cases**:
    - This strategy emphasizes creating a design that facilitates thorough testing. Although it ensures high reliability and correctness, it may not address the broader architectural needs of the system.
    - It is better suited as a complementary approach rather than the primary strategy for a complex system like the Smart Parking System.

3.5.2 Architectural Strategies

In this section, we will detail the architectural styles and patterns chosen for the Smart Parking System, justifying each choice at a high level. The chosen styles include Client-Server, Structural (Layered), and Event-Driven styles, each bringing specific benefits that align with the project's requirements and constraints.

1. Client-Server Style

Description: The Client-Server style divides the system into two main components: clients and servers. The client is responsible for the user interface and user interactions, while the server handles the backend operations, such as processing requests, database management, and business logic.

Justification:

- **Separation of Concerns:** This style allows clear separation between the user interface and backend logic, facilitating independent development and maintenance.
- **Scalability:** The server can be scaled independently to handle increased load without affecting the client side. This is crucial for the Smart Parking System to accommodate a growing number of users.
- **Ease of Updates:** Backend updates can be made without disrupting the client applications, ensuring a smooth user experience.
- **Security:** Sensitive operations and data storage are confined to the server side, enhancing security.

2. Structural Style (Layered Architecture)

Description: The Layered Architecture style organizes the system into layers, each with a specific responsibility. Common layers include the presentation layer, business logic layer, data access layer, and data storage layer.

Justification:

- **Modularity:** Each layer encapsulates specific functionalities, making the system more modular and easier to manage.
- **Maintainability:** Clear separation of concerns allows for easier maintenance and updates. For instance, changes in the data access layer do not affect the business logic layer.
- **Reusability:** Layers can be reused across different projects. For example, the business logic layer for handling parking slot reservations can be reused in other parking-related applications.
- **Testability:** Each layer can be tested independently, ensuring higher reliability and easier debugging.
- **Scalability:** Individual layers can be scaled as needed. For example, the data access layer can be scaled to handle more database queries without impacting the presentation layer.

3. Event-Driven Style

Description: The Event-Driven style relies on the communication of system components through events. It typically involves a central broker that manages events published by producers and consumed by subscribers.

Justification:

- **Loose Coupling:** Components interact through events, reducing dependencies and allowing for more flexible and scalable system design.

- **Scalability:** The publish-subscribe mechanism enables horizontal scaling by distributing events across multiple handlers.
- **Extensibility:** New features can be added by simply creating new event handlers or producers, without altering the existing system.
- **Resilience:** Event-driven systems can handle failures gracefully, as components can operate independently and recover from errors asynchronously.
- **Real-time Processing:** This style is ideal for handling real-time data updates, such as the real-time availability of parking slots in the Smart Parking System.

3.5.3 Architectural Quality Requirements

For the Smart Parking System, the following quality requirements have been prioritized to ensure the system meets user needs and performs efficiently under various conditions. Each requirement is quantified or specified in a testable manner.

1. Availability

Requirement: The system must be highly available to provide uninterrupted service to users.

Quantification: The system is expected to have at least 85% uptime.

Justification: High availability is crucial as users rely on the system to find and reserve parking spots in real-time. Any downtime can lead to user dissatisfaction and operational inefficiencies.

Testable Specification: Conduct continuous monitoring and logging of system uptime, ensuring that any downtime incidents are documented and analyzed to maintain the 85% target.

2. Scalability

Requirement: The system should scale efficiently to handle an increasing number of users and parking lots.

Quantification: The system is designed to handle 50 user requests per second.

Justification: As urban areas grow and the number of users increases, the system must be capable of scaling to meet higher demand without compromising performance.

Testable Specification: Perform load testing to verify that the system can handle 50 user requests per second without significant degradation in response time.

3. Performance (Responsiveness)

Requirement: The system should respond quickly to user interactions to provide a seamless user experience.

Quantification: The system should process and display real-time slot availability within 2 seconds and complete reservation transactions within 5 seconds.

Justification: Fast response times are essential to maintain user satisfaction and ensure the system is practical for real-time use.

Testable Specification: Conduct performance testing to measure response times for displaying slot availability and completing reservations, ensuring they meet the specified targets.

4. Security

Requirement: The system must ensure data privacy and secure user interactions.

Quantification: Implement role-based access control (RBAC), data encryption in transit and at rest, and multi-factor authentication (MFA) for user accounts.

Justification: Given the sensitive nature of user data and payment information, robust security measures are necessary to protect against unauthorized access and data breaches.

Testable Specification: Conduct security audits and penetration testing to validate the effectiveness of RBAC, encryption, and MFA. Regularly review and update security policies.

5. Usability (Customizability)

Requirement: The system should be user-friendly and allow users to customize their experience.

Quantification: Users should be able to adjust at least five parameters (e.g., preferred parking locations, notification settings, reservation duration) through an intuitive interface.

Justification: Customizability enhances user satisfaction by allowing users to tailor the system to their specific needs and preferences.

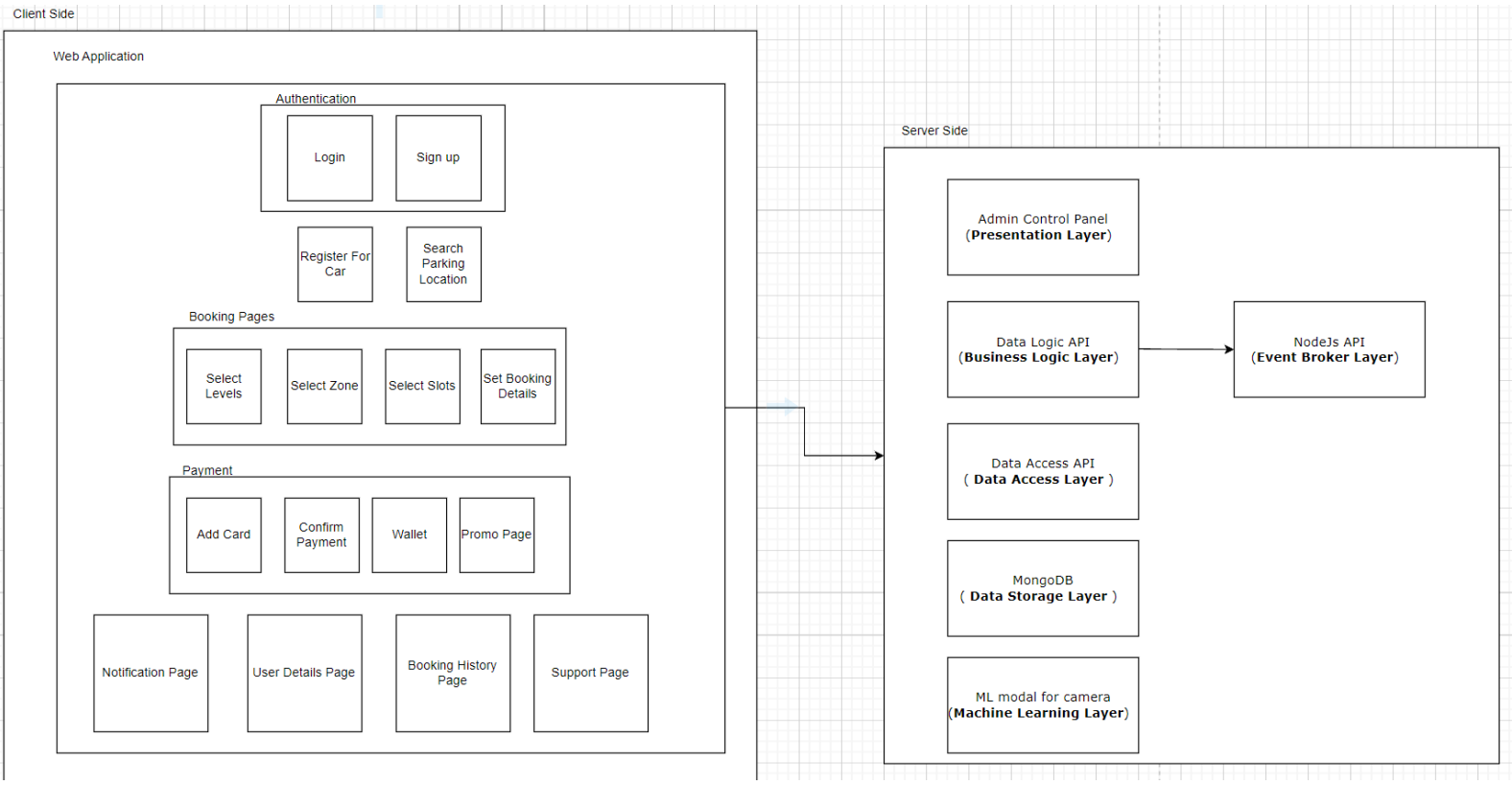
Testable Specification: Conduct usability testing with real users to evaluate the ease of customization and the intuitiveness of the interface. Collect feedback and make necessary adjustments to ensure user satisfaction.

3.5.4 Architectural Design & Pattern

The architectural design of the Smart Parking System incorporates several well-defined patterns to address the project requirements and constraints. These patterns include the Client-Server, Layered, and Event-Driven styles. Below is an overview of the system architecture, including a detailed architectural diagram and the justification for each design decision.

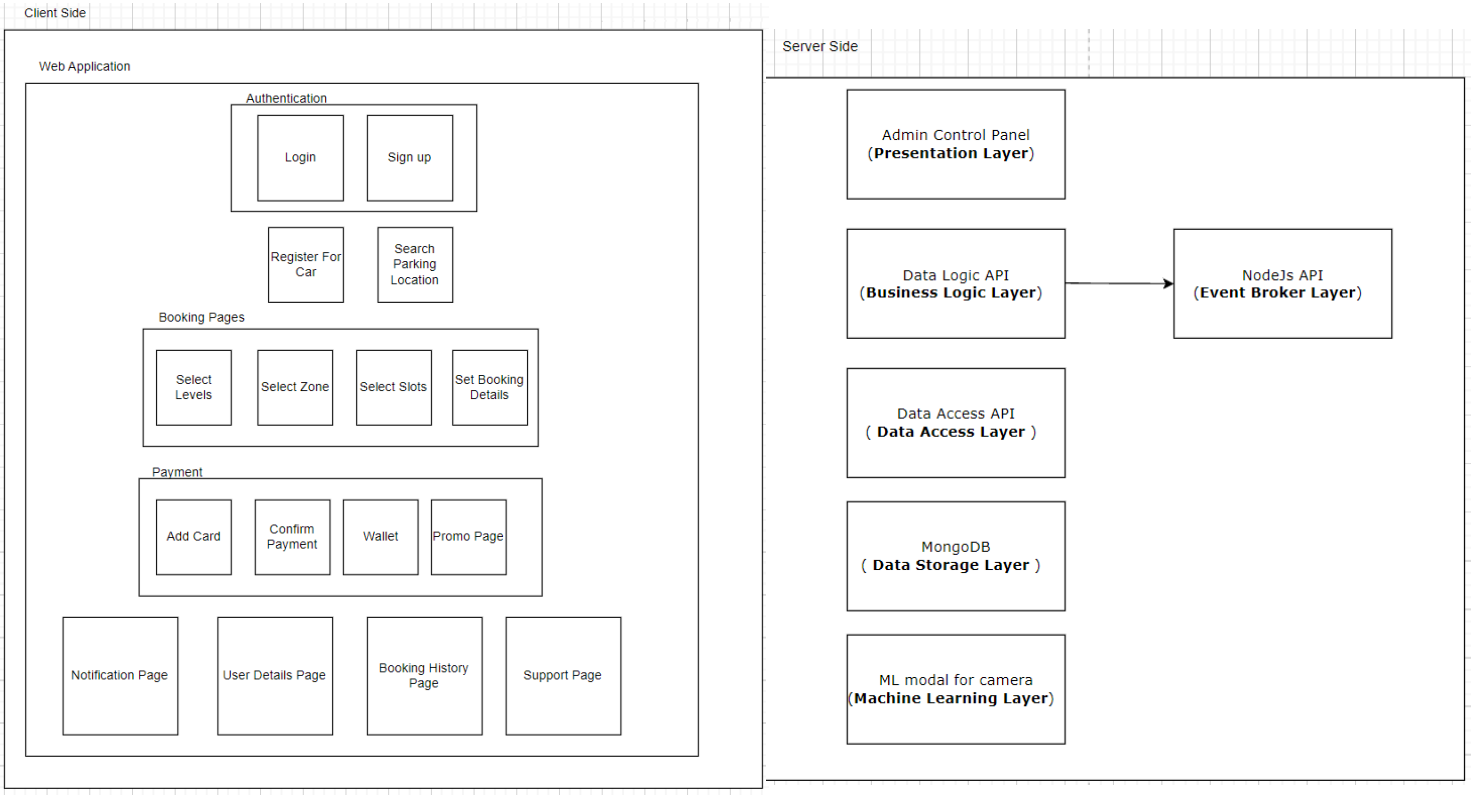
Architectural Diagram

Overview



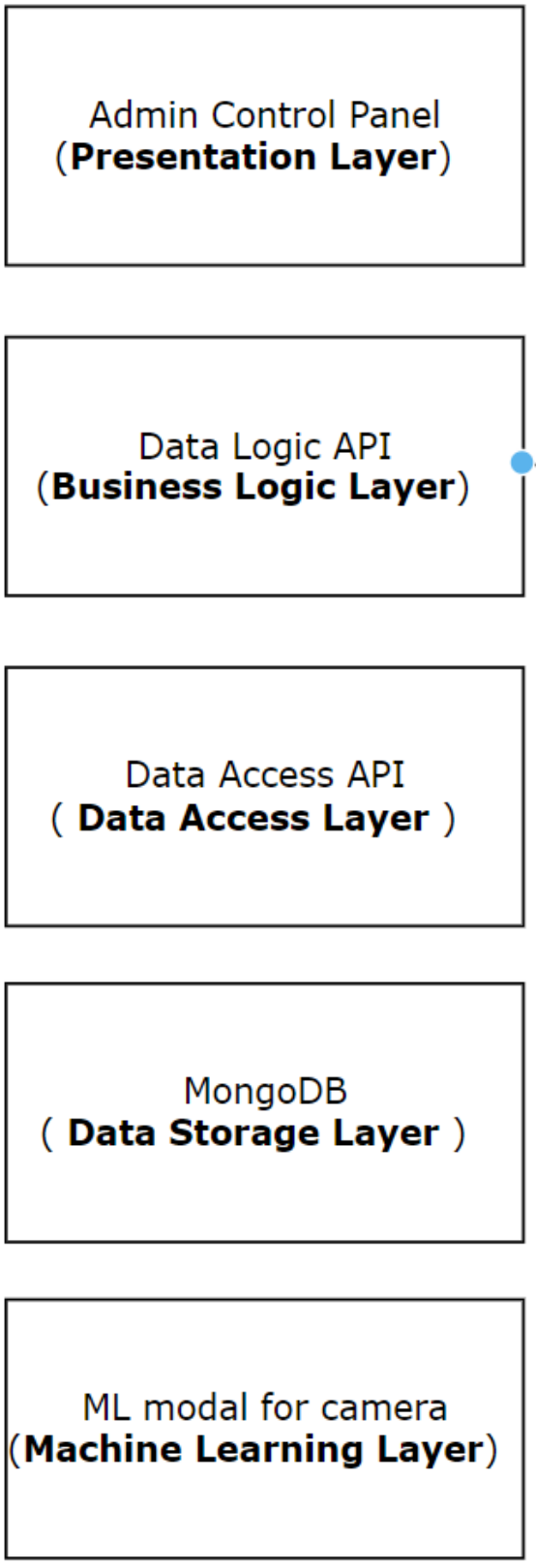
Justification of Design Decisions

1. Client-Server Pattern



- **User Interface (Web & Mobile Clients):** The system uses Flutter and Dart for the web and mobile interfaces, providing a responsive and interactive user experience. The separation of client and server components ensures that changes to the backend do not affect the frontend, promoting easier maintenance and updates.
- **Justification:** This pattern facilitates clear separation of concerns, allowing independent development and scaling of the client and server components. It addresses the requirement for a scalable and maintainable architecture, accommodating a large number of concurrent users.

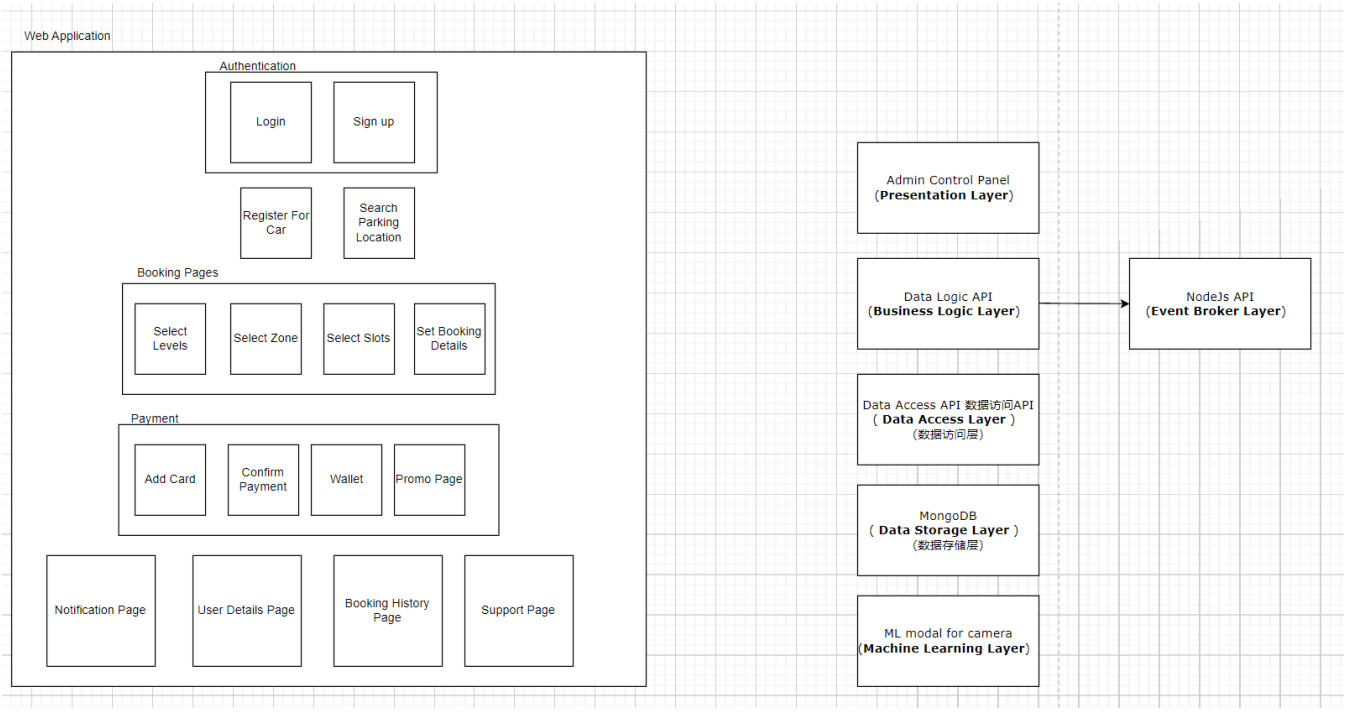
2. **Layered Pattern**



- **Presentation Layer:** The client-side interfaces (both web and mobile) are part of the presentation layer, which interacts directly with the users.

- **Business Logic Layer:** The NodeJS and Express handle the core business logic, including parking slot reservations, user authentication, and payment processing.
- **Data Access Layer:** The NodeJS and Express API manage data interactions, including querying the MongoDB and integrating with external services.
- **Data Storage Layer:** MongoDB stores all persistent data, ensuring high availability and scalability.
- **Machine Learning Layer** This layer involves the use of a machine learning model to analyze camera feeds and recognize if a car is parked in a slot.
- **Justification:** The layered architecture provides modularity, making the system easier to understand, develop, and maintain. Each layer has distinct responsibilities, enhancing reusability and testability. This pattern also supports scalability by allowing independent scaling of each layer. Integrating machine learning enhances the system's ability to accurately detect and monitor parking slot occupancy in real-time, improving the overall efficiency and reliability of the Smart Parking System.

3. **Event-Driven Pattern**



- **Event Broker Layer:** This layer, implemented using NodeJS, manages the event-driven communication between system components. It uses an event broker to handle publish-subscribe interactions, enabling real-time updates and asynchronous processing.
- **Justification:** The event-driven pattern promotes loose coupling between components, enhancing flexibility and scalability. It supports

real-time processing and efficient handling of asynchronous events, critical for the real-time slot availability feature of the Smart Parking System.

Component Descriptions

1. **User Interface (Web & Mobile Clients)**
    - **Technology:** Flutter and Dart
    - **Function:** Provides a responsive and user-friendly interface for accessing the Smart Parking System features.
    - **Justification:** Ensures a seamless user experience across different devices, addressing the availability and usability requirements.
2. **Business Logic Layer**
    - **Technology:** NodeJS and Express
    - **Function:** Implements the core business logic, processing user requests, and interacting with the data access layer.
    - **Justification:** Centralizes business rules and logic, making it easier to manage and update the system's core functionalities.
3. **Data Access Layer**
    - **Technology:** NodeJS and Express
    - **Function:** Manages database operations and interactions with external services.
    - **Justification:** Provides a clear separation between business logic and data management, enhancing maintainability and scalability.
4. **Data Storage Layer**
    - **Technology:** MongoDB
    - **Function:** Stores all persistent data, including user information, parking slot data, and transaction records.
    - **Justification:** Ensures high availability, reliability, and scalability, addressing the system’s data storage requirements.
5. **Event Broker Layer**
    - **Technology:** NodeJS
    - **Function:** Manages event-driven communication, handling real-time updates and asynchronous processing.
    - **Justification:** Enhances the system's ability to process real-time data efficiently and handle asynchronous events, critical for maintaining up-to-date parking slot availability.
6. **Machine Learning Layer**
    - **Technology:** Machine Learning Model
    - **Function:** Analyzes camera feeds to detect parking slot occupancy.
    - **Justification:** Provides accurate and real-time detection of parking slots, improving the system's efficiency and reliability.

3.5.5 Architectural Constraints

In developing the Smart Parking System, several constraints must be considered that will directly impact the architectural design. These constraints include client-specific requirements, technology choices, and deployment considerations. Below is a detailed outline of these constraints:

1. Deployment Constraints

Containerization:

- The system should consider containerization in Docker for easy deployment and management. This constraint impacts how the application is packaged and deployed, promoting the use of container orchestration for managing microservices.

2. Security and Compliance Constraints

Data Encryption:

- All data in transit and at rest must be encrypted. This requires integrating encryption mechanisms throughout the system, impacting performance and design choices for data handling and storage.

User Authentication and Authorization:

- The system must implement robust authentication and authorization mechanisms, including multi-factor authentication (MFA) and role-based access control (RBAC). This affects how user sessions are managed and how sensitive operations are secured.

Privacy Regulations:

- Compliance with data protection regulations (e.g., GDPR) is mandatory. This constraint impacts data handling policies, user consent mechanisms, and data storage practices to ensure legal compliance.

3. Hardware and Environmental Constraints

Sensor Integration:

- The system will utilize cameras to detect vehicle presence in parking spaces. This requires considerations for hardware compatibility, real-time data processing capabilities, and environmental factors such as installation locations and network connectivity.

Network Reliability:

- Given the reliance on real-time data, the system must account for varying network conditions and ensure reliable communication between sensors, the backend, and user interfaces. This impacts the design of the communication protocols and error-handling mechanisms.

4. Budget Constraints

Limited Budget:

- The project has a budget of R5000, which must cover cloud services, third-party services, license fees, mobile data for IoT devices, hardware requirements, and marketing material. This financial constraint necessitates careful planning and prioritization of features and technologies to stay within budget.

3.5.6 Technology Choices

Technology Choice 1: Flutter and Dart

- **Pros:** Cross-platform, responsive, user-friendly interface
- **Cons:** Limited libraries for complex functionalities
- **Justification:** Ensures a seamless user experience across different devices

Technology Choice 2: NodeJS and Express

- **Pros:** High performance, scalability, large community support
- **Cons:** Single-threaded, callback issues
- **Justification:** Centralizes business rules and logic, making it easier to manage and update

Technology Choice 3: MongoDB

- **Pros:** Flexible schema, high availability, scalability
- **Cons:** Complex queries can be slower, requires additional skills
- **Justification:** Ensures high availability, reliability, and scalability