

Healthcare Appointment Scheduling System

Healthcare Appointment Scheduling System allows patients to book, manage, and cancel medical appointments while enabling doctors to control availability and schedules. The system provides real-time updates, automated notifications, and secure communication between patients and healthcare providers.

Why Start with Monolithic Architecture?

1-Simplicity and Speed

- Faster Development: Single codebase, no service boundaries to manage
- Easier Deployment: One application to deploy, test, and monitor
- Lower Complexity: No inter-service communication, no distributed system challenges
- Quick Time-to-Market: Can deliver MVP faster with fewer moving parts

2-Resource Efficiency

- Small Team: Initial development team can work efficiently on single codebase
- Lower Infrastructure Costs: Single server/container initially sufficient
- Simpler Testing: End-to-end testing easier in monolithic system
- Reduced Overhead: No API Gateway, service discovery, or orchestration needed initially

3-Uncertain Requirements

- Early Stage: System requirements may evolve rapidly
- Flexibility: Easier to refactor and change in single codebase
- Learning Phase: Understand domain and business logic before splitting
- Proof of Concept: Validate system concept before investing in microservices complexity

4-Smaller Scale

- Initial Users: System starts with hundreds/thousands of users, not millions
- Single Database: Simpler data management, no distributed transactions
- Vertical Scaling: Can scale up (more CPU/RAM) before needing horizontal scaling
- Cost-Effective: Avoid over-engineering for initial scale

Why Evolve to Microservices?

Evolution Trigger: System Growth

1-Scalability Challenges

- Traffic Growth: System now handles 10,000+ concurrent users (requirement)
- Bottleneck Identification: Certain features (e.g., Scheduling) need more resources
- Independent Scaling: Need to scale Scheduling service separately from Notification service
- Performance Requirements: <2 seconds response time requires optimized services

2-Complexity Management

- Codebase Size: Monolithic codebase becomes too large to manage
- Team Growth: Multiple teams need to work independently
- Deployment Conflicts: Different features need different deployment schedules
- Technology Diversity: Different services may need different tech stacks

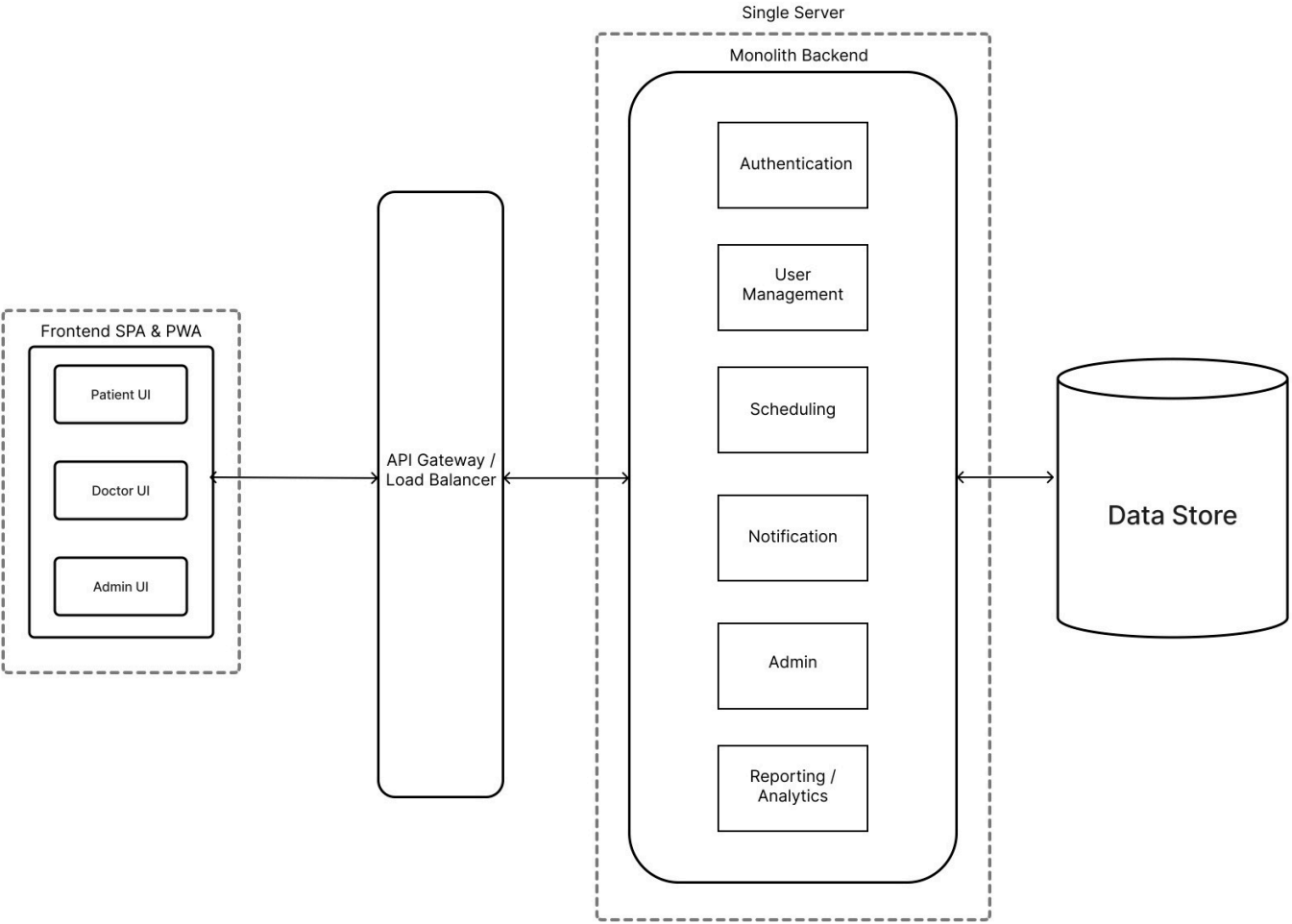
3-Business Requirements

- Modularity Requirement: System must be split into independent modules (Patient, Doctor, Scheduling, Notification)
- Future Integration: Need to integrate with EHR systems as separate service
- 24/7 Availability: Microservices provide better fault isolation
- Maintainability: Easier to maintain and update individual services

4-Technical Benefits

- Fault Isolation: Failure in one service doesn't bring down entire system
 - Technology Flexibility: Each service can use optimal technology
 - Independent Deployment: Deploy services independently without affecting others
 - Better Testing: Smaller, focused services are easier to test
-

Monolithic Architecture



Component Explanations

1. Frontend Layer (SPA with PWA)

- **Patient UI:** Interface for patients to search doctors, book appointments, manage appointments
- **Doctor UI:** Interface for doctors to manage schedules, view appointments
- **Admin UI:** Administrative interface for system management
- **Communication:** REST API calls to monolithic backend

2. Monolithic Application

a. Presentation Layer

- **Purpose:** Handles HTTP requests, routing, request validation
- **Responsibilities:**
 - Receives requests from frontend
 - Validates input data
 - Routes requests to appropriate business logic
 - Formats responses

b. Business Logic Layer

- **Purpose:** Contains all business rules and domain logic
- **Responsibilities:**
 - Patient registration and management
 - Doctor profile and schedule management
 - Appointment booking logic (prevents double-booking)
 - Notification scheduling
 - Search functionality

c. Data Access Layer

- **Purpose:** Handles database operations
- **Responsibilities:**
 - Database queries
 - Data persistence

3. Single Database

- **Purpose:** Stores all system data
-

Component Interactions in Monolith

Appointment Booking Flow:

1. **User Action:** Patient selects time slot, clicks "Book"
2. **Presentation Layer:** Receives booking request
3. **Business Logic:**
 - Scheduling Service checks availability
 - Prevents double-booking (transaction)
 - Creates appointment record
 - Triggers Notification Service
4. **Data Access:** Writes to Appointments table
5. **Notification:** Notification Service schedules reminder
6. **Response:** Confirmation sent to frontend

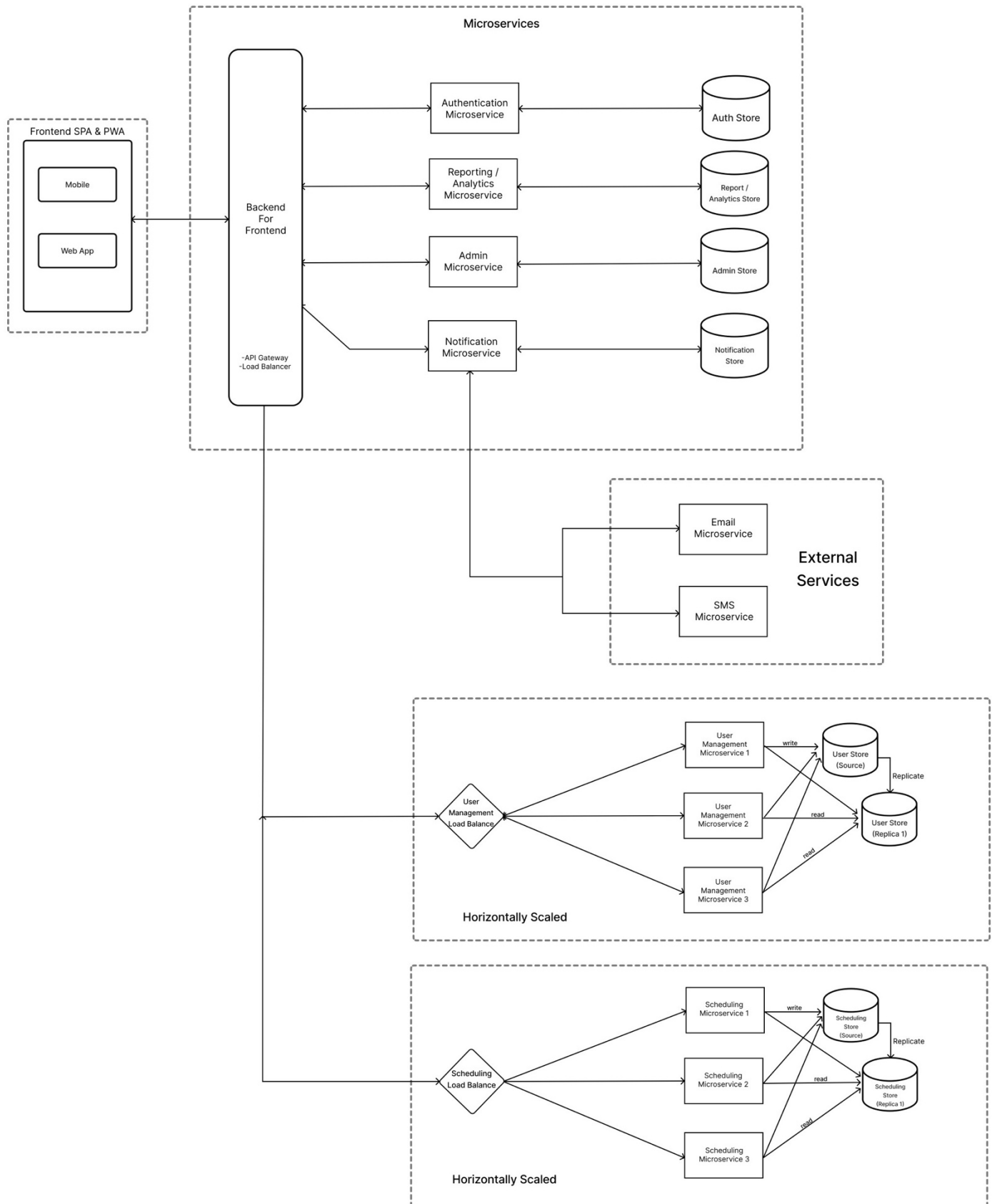
Advantages:

- Simple request flow
- Fast in-process communication
- Easy debugging (single process)

Disadvantages:

- All modules share resources
 - Cannot scale modules independently
 - Single point of failure
 - Tight coupling between modules
-

Microservices Architecture



Component Explanations

1. Frontend Layer (SPA with PWA)

- **Same as Monolith:** Patient, Doctor, Admin UIs
- **Communication:** Single API Gateway endpoint (BFF pattern)

2. Load Balancer

- **Purpose:** Distributes incoming requests across API Gateway instances
- **Responsibilities:**
 - Request distribution
 - Health checking
- **Benefits:** High availability, handles 10,000+ concurrent users

3. API Gateway (Backend for Front-End - BFF)

- **Purpose:** Single entry point for all frontend requests
- **Benefits:**
 - Frontend doesn't need to know about multiple services
 - Centralized security
 - Request/response optimization

4. Microservices

a. User Management Microservice

- **Purpose:** Manages all patient-related and doctor-related operations
- **Responsibilities:**
 - Patient registration
 - Patient profile management
 - Patient appointment history
 - Patient authentication
 - Doctor profile management
 - Doctor search functionality
 - Schedule management
 - Availability updates
- **Database:** User Store (dedicated)

c. Scheduling Service

- **Purpose:** Core appointment booking logic
- **Responsibilities:**
 - Appointment creation
 - Time slot management
 - Double-booking prevention
 - Appointment rescheduling/cancellation
- **Database:** Scheduling Store (dedicated)

d. Notification Service

- **Purpose:** Handles all notifications
- **Responsibilities:**
 - Email reminder scheduling
 - SMS reminder scheduling
 - Notification queue management
 - Delivery tracking
- **Database:** Notification Store (dedicated)
- **External Services:** Integrates with Email/SMS Microservices

e. Authentication Service

- **Purpose:** Manages user authentication and authorization
- **Responsibilities:**
 - User login and logout
 - Token generation and validation (JWT)
 - Password management and encryption
 - Session management
 - Role-based access control (RBAC) enforcement
 - User registration authentication
- **Database:** Auth Store (dedicated)

f. Reporting and Analytics Service

- **Purpose:** Generates reports and provides analytics insights
- **Responsibilities:**
 - Appointment statistics and reports
 - Doctor performance analytics
 - Patient engagement metrics
 - System usage reports
 - Revenue and billing reports
 - Custom report generation
 - Data aggregation and analysis
- **Database:** Analytics Store (dedicated)

g. Admin Service

- **Purpose:** Manages administrative operations and system configuration
- **Responsibilities:**
 - System configuration management
 - User management (create, update, delete users)
 - Role and permission management
 - System monitoring and health checks
 - System maintenance operations
- **Database:** Admin Store (dedicated)

5. Databases (Per Service)

- **User Store**
- **Scheduling Store**
- **Notification Store**
- **Reporting and Analytics Store**
- **Admin Store**
- **Auth Store**

Benefits:

- Data isolation
- Independent scaling
- Technology flexibility
- Better security

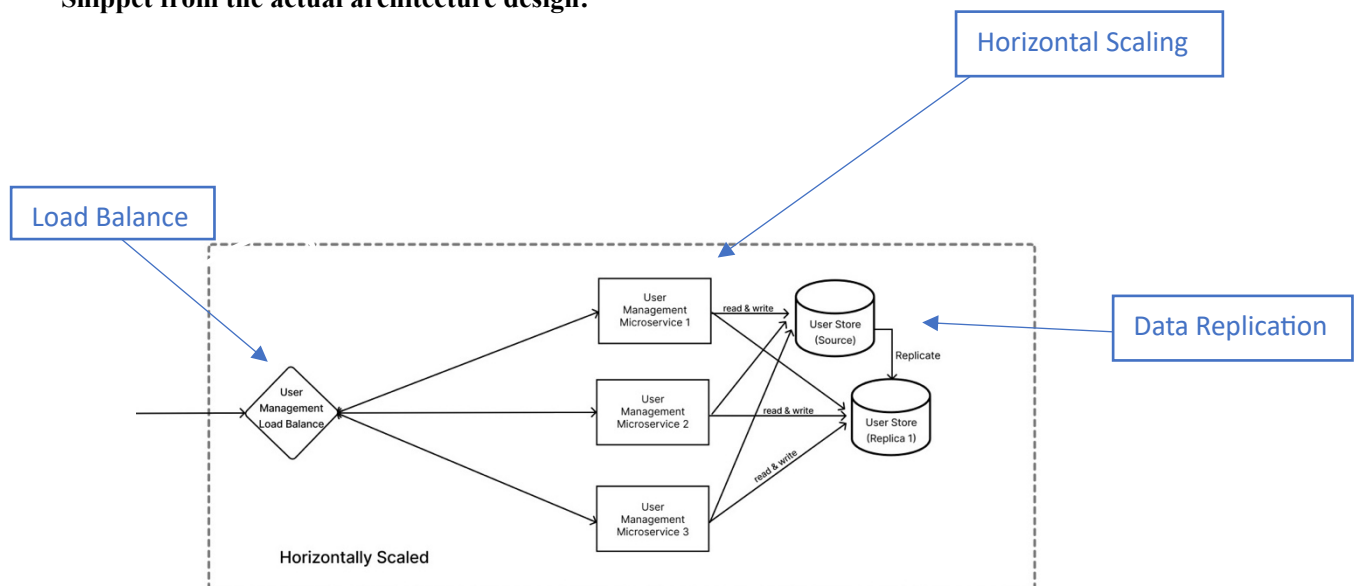
Horizontal Scaling For Microservices

Applied For User Management Microservice and Scheduling Microservices as both of them can experience heavy load due to thousands of users using the application. Vertical reaches its limits, and further splitting the service into more microservices is not practical. To handle the increased demand, the system adopts horizontal scaling, by using load balancer, which enables horizontal scaling by acting as a reverse proxy. This improves performance, stability, and overall system reliability.

Horizontal Scaling For Databases

Applied for User Management Microservice and Scheduling Databases using source–replica model.

Snippet from the actual architecture design:



Component Interactions in Microservices

Request Flow (Patient Search):

1. **Frontend:** Patient searches for doctor
2. **Load Balancer:** Distributes request to API Gateway
3. **API Gateway:** Routes to Doctor Service
4. **Doctor Service:** Queries Doctor DB
5. **Response:** Returns through API Gateway to frontend

Advantages:

- Independent scaling per service
- Fault isolation
- Better performance optimization

Disadvantages:

- More complex deployment
- More infrastructure to manage

Backend Architecture Choice: Microservices Architecture

Justification:

- **Scalability:** Each service scales independently (handles 10,000+ concurrent users)
 - **Modularity:** Aligns with requirement for independent modules (Patient, Doctor, Scheduling, Notification)
 - **Future Evolution:** Supports future EHR integration as separate service
 - **Performance:** Services can be optimized individually (<2s response requirement)
 - **Maintainability:** Independent deployment and testing of each service
 - **Flexibility:** Services can use different technologies as needed
-

Part 4: Frontend Architecture Choice

SPA vs SPA with PWA

Option 1: Single Page Application (SPA)

Option 2: SPA with PWA (Progressive Web App)

Comparison: SPA vs SPA with PWA

Feature	SPA	SPA with PWA
Offline Support	No	Yes (cached content)
Installable	No	Yes (home screen)
Push Notifications	No	Yes
App-like Experience	Limited	Full
Development Complexity	Simple	More complex
Browser Support	All modern	Modern browsers
Initial Load	Slower	Slower (more assets)
Caching	Basic	Advanced
Mobile Experience	Good	Excellent

After Research the chosen style: Single Page Application (SPA) enhanced with Progressive Web App (PWA) features

Justification

An SPA provides a fast, seamless user experience for data exploration and dashboards, with instant navigation and no full-page reloads — essential for analytics workflows. Adding PWA capabilities enables offline caching, installability, and faster reloading, improving reliability for users and reducing server load. The SPA + PWA combination supports responsive UI, fluid navigation, and high user engagement across browsers and devices.

Why This Approach Fits UI/UX and System Flow

UI/UX Benefits:

1. **Seamless Experience**
 - **No Page Reloads:** SPA provides smooth navigation
 - **Mobile-Optimized:** Touch-friendly, responsive design
 2. **App-Like Feel (PWA)**
 - **Installation:** Users can install on home screen
 - **Full Screen:** Runs without browser chrome
 3. **Offline Access (PWA)**
 - **View Appointments:** Users can view cached appointments offline
 - **View Schedules:** Doctors can view schedules without internet
 4. **Push Notifications (PWA)**
 - **Real-time Updates:** Instant notifications for schedule changes
 - **User Convenience:** Notifications even when app closed
-