System Architecture & Logic

1. Prioritization Logic: The "Tiered Capacity" Model

To satisfy the requirement for **elastic capacity management** while strictly enforcing **hard limits** for standard patients, I implemented a **Tiered Buffer System**. Instead of a simple First-Come-First-Serve queue, the capacity of a slot expands dynamically based on the priority of the incoming request.

Priority Tier	User Types	Capacity Limit	Logic
Standard	ONLINE, WALKIN	**Base Capacity (10)**	Hard limit. If booked count hits 10, these requests are rejected immediately.
Premium	PAID, FOLLOWUP	**Base + Premium Buffer (12)**	These users can "stretch" the slot. If the slot is full (10/10), a Paid user is accepted into the premium buffer.
Critical	EMERGENCY	**Base + Premium + Emergency (15)**	Emergency cases override standard and premium limits, utilizing the maximum emergency reserve.

Why this approach?

This ensures that high-value and critical flows (Emergency/Paid) are never blocked by a flood of standard bookings, effectively reserving capacity without keeping slots empty unnecessarily.

2. Handling Edge Cases & Real-World Variability

The system addresses specific "real-world" operational anomalies mentioned in the assignment context.

A. Concurrency (The Race Condition)

- **Scenario:** In a busy OPD, an online user and a desk receptionist might hit "Book" for the last available slot at the exact same millisecond.
- **Solution:** Implemented **Distributed Mutex Locking** using Redis (SET key value NX EX 2).
- **Mechanism:**
    1. Incoming Request → Acquire Lock for doctor_time_slot.
    2. **If Locked:** Return 429 System Busy (Client retries).
    3. **If Acquired:** Perform Read-Check-Write operation.
    4. **Release:** Unlock key.
- **Result:** Guarantees atomic booking operations, ensuring bookedCount never exceeds the limit due to race conditions.

B. Doctor Delays (Variable Capacity)

- **Scenario:** A doctor is running late, reducing the effective time they can see patients.
- **Solution:** The Slot model includes an isDelayed flag.
- **Logic:** If isDelayed: true, the system dynamically **reduces the effective limit to 80%**.
    - *Standard Limit:* 10 →8.
    - *Impact:* Standard bookings are rejected earlier to prevent overcrowding, while Emergency cases can still bypass this restriction.

C. Dynamic Reallocation (No-Shows)

- **Scenario:** A patient cancels or doesn't show up. The slot must be reused instantly.
- **Solution:** The '/cancel' and '/noshow' endpoints perform an atomic decrement ($inc: { bookedCount: -1 }) on the slot.
- **Result:** The very next booking request (even milliseconds later) will see the available capacity and succeed, fulfilling the "Dynamic Reallocation" requirement.

3. Failure Handling

The system is designed to fail predictably and safely rather than crashing or leaving data in an inconsistent state.

- **Input Validation:** Used **Zod** to strictly validate Enums (e.g., WALKIN vs WALK_IN). Invalid inputs are caught before they touch the database or Redis, returning a clean 400 Bad Request.
- **Lock Contention:** If Redis is down or the lock cannot be acquired, the system fails safe by returning a temporary error rather than allowing a potential overbooking.
- **Deadlock Prevention:** All Redis locks have a **TTL (Time-To-Live)** of 2 seconds. If the node process crashes mid-request, the lock automatically expires, ensuring the slot doesn't remain "frozen" forever.