Event Ticketing Platform

Solution Architecture Assessment

February 2026

Problem & Scenarios

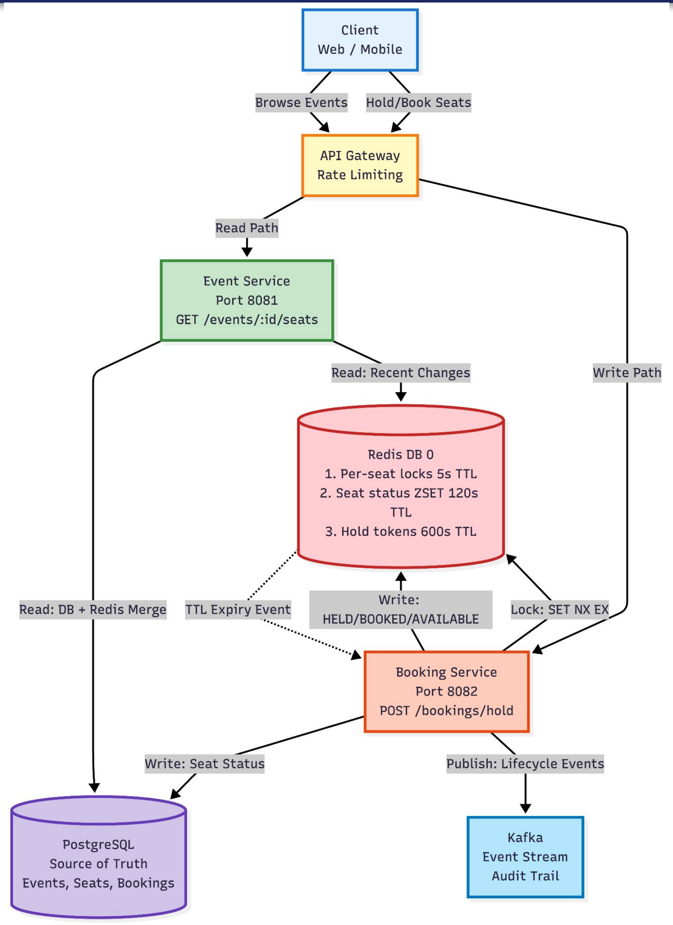
Category	Scenario	Strategic Rationale
Write Path	Seat Hold (e.g 10-Minute TTL)	Addresses high-concurrency locking, distributed coordination, expiry handling, and event-driven workflows
Read Path	Event Discovery with Caching	Optimizes search latency, availability reads, and cache consistency strategies

Challenge	Business Risk / Impact
Double Booking	Revenue loss and reputational damage due to concurrent seat allocation conflicts
Abandoned Holds	Inventory lock leading to reduced conversion and revenue leakage
Lack of Audit Trail	Operational risk in dispute resolution and regulatory traceability

Success Criteria (North Star Outcomes)

Zero double bookings | Automated seat release via deterministic TTL enforcement | End-to-end booking lifecycle auditability

Architectural Overview



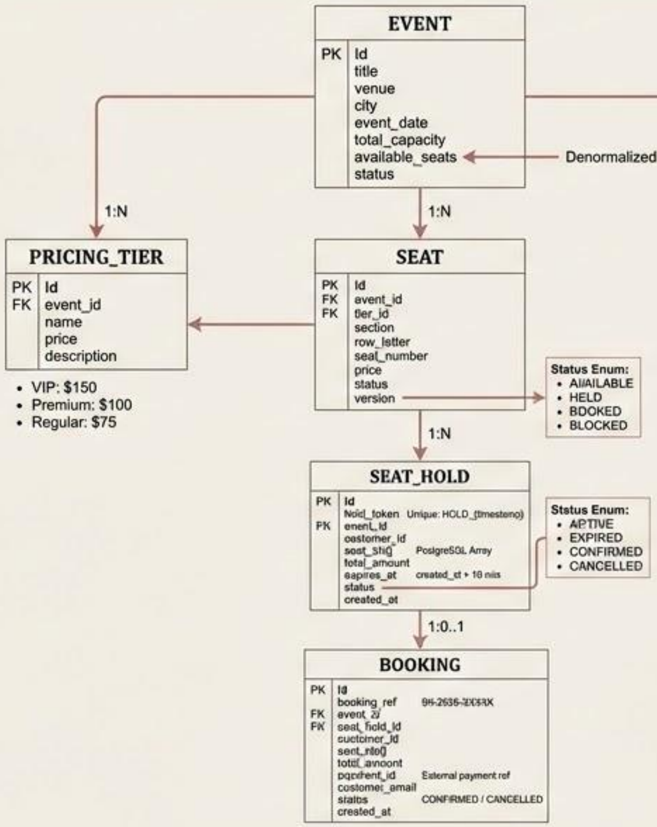
Architecture Style
Microservices-based, event-driven system separating Event and Booking domains for scalability and fault isolation.

Concurrency Control
Redis distributed locks + ACID DB transactions ensure seat integrity under high contention.

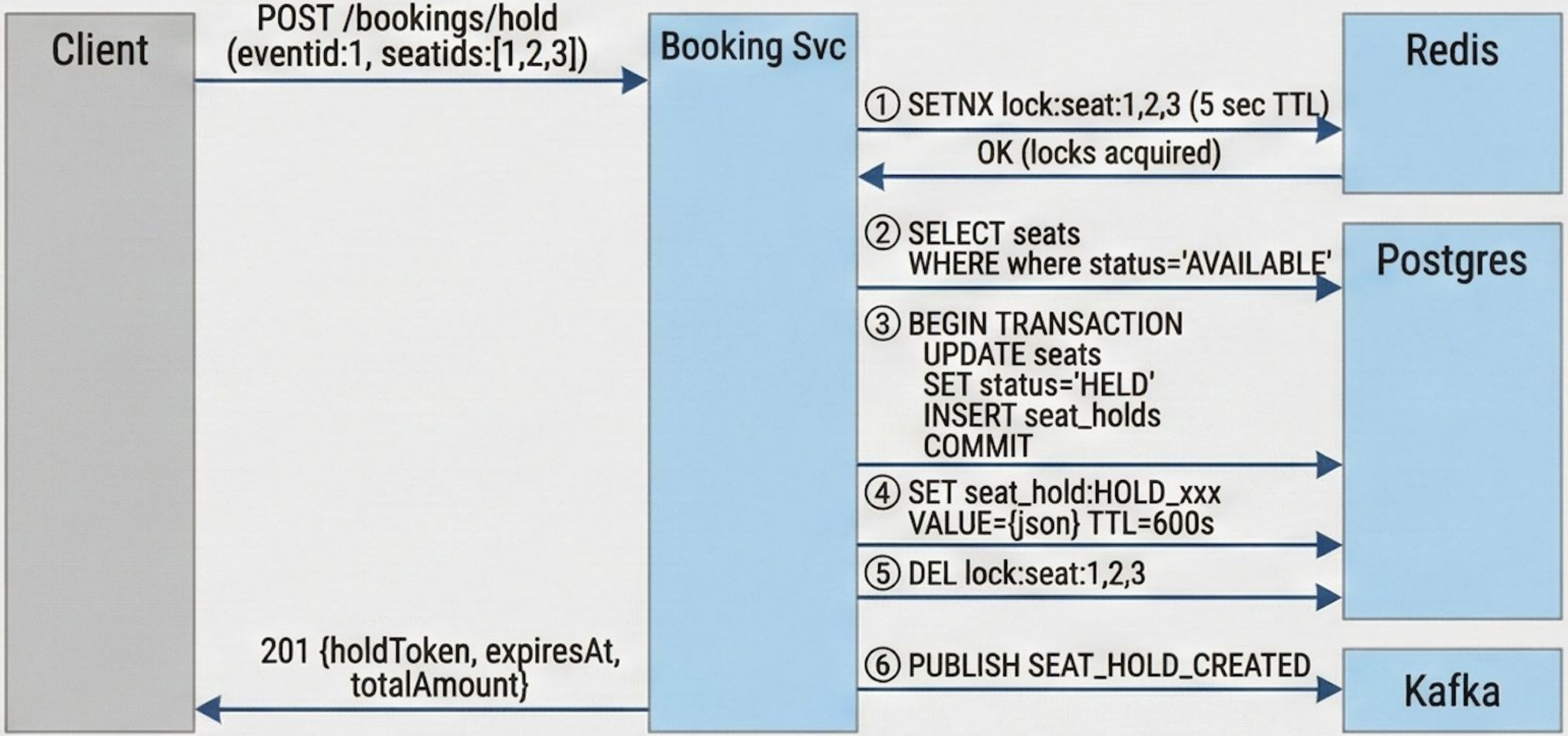
Event-Driven Workflow
Kafka publishes lifecycle events:
`SEAT-HOLD-CREATED`, `SEAT-HOLD-CONFIRMED`, `SEAT-HOLD-CANCELLED`, `SEAT-HOLD-EXPIRED`, `BOOKING-CONFIRMED`

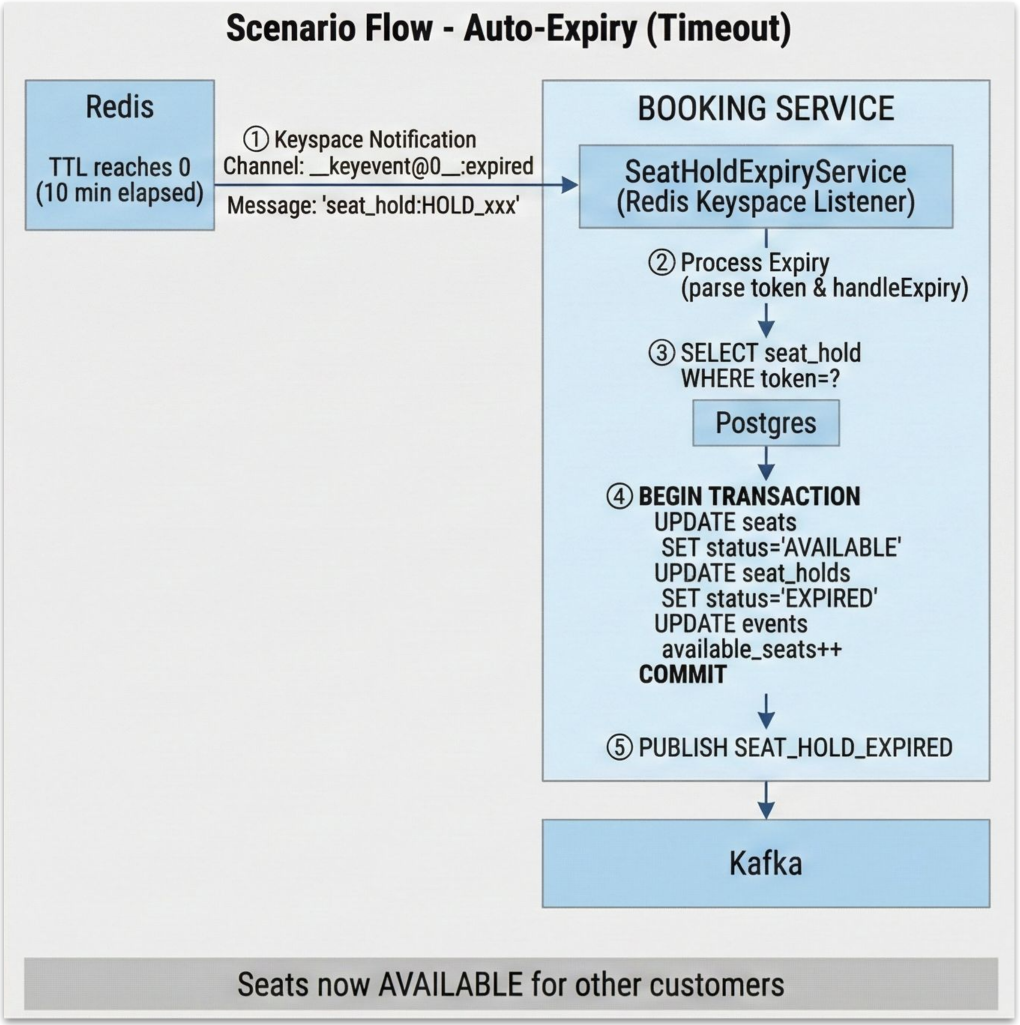
Resilience & Recovery
TTL-based auto release, idempotent APIs, and service isolation prevent deadlocks and duplicate bookings.

Data Entity Design



Scenario Flow - Seat Hold (Write)



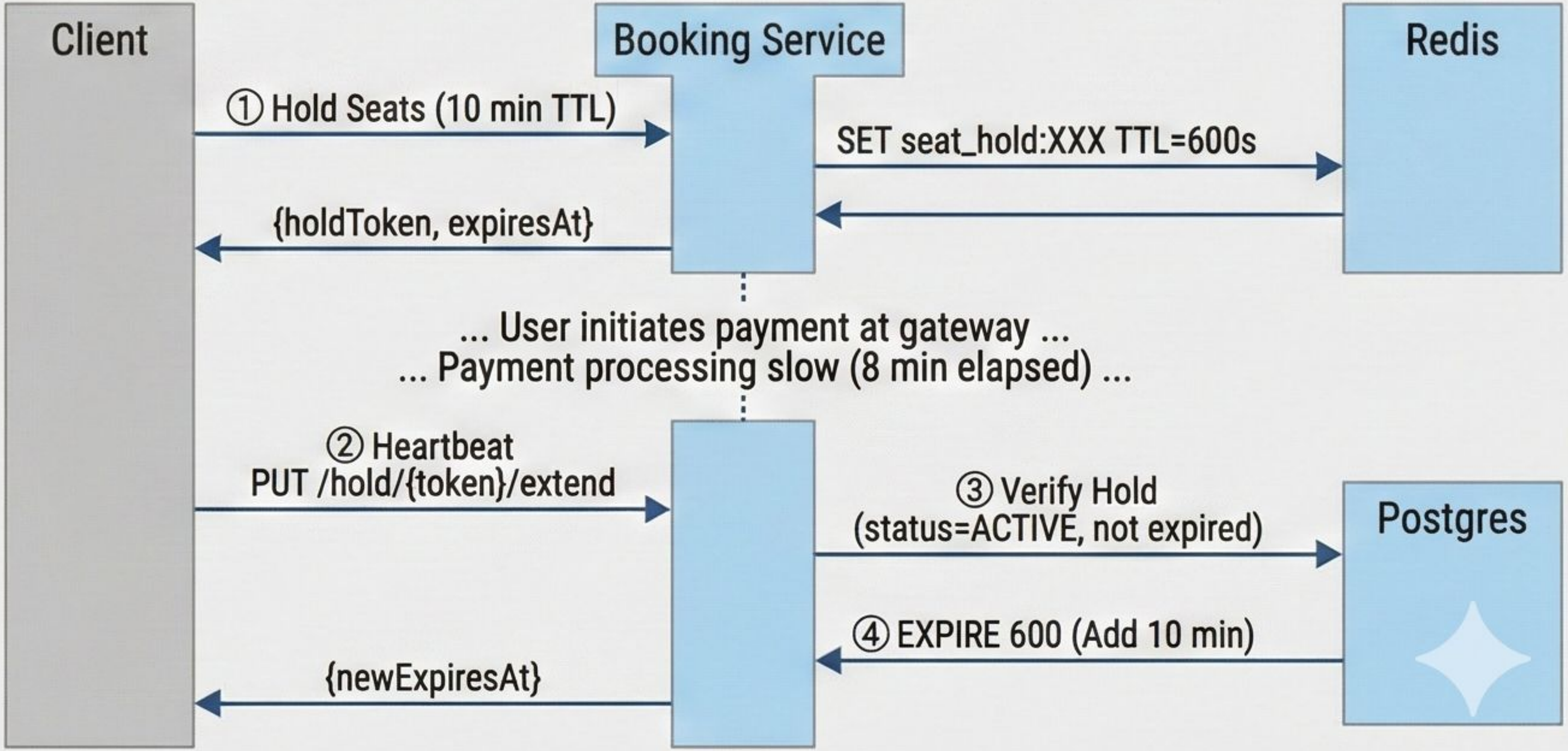

Scenario Flow - Auto-Expiry (Timeout)
Redis
TTL reaches 0 (10 min elapsed)
① Keyspace Notification
Channel: __keyevent@0__:expired
Message: 'seat_hold:HOLD_xxx'
BOOKING SERVICE
SeatHoldExpiryService (Redis Keyspace Listener)
② Process Expiry (parse token & handleExpiry)
③ SELECT seat_hold WHERE token=?
Postgres
④ BEGIN TRANSACTION
UPDATE seats SET status='AVAILABLE'
UPDATE seat_holds SET status='EXPIRED'
UPDATE events available_seats++
COMMIT
⑤ PUBLISH SEAT_HOLD_EXPIRED
Kafka
Seats now AVAILABLE for other customers

Assumption :

Booking not completed within Hold-Window

Redis key pattern: `seat:{eventId}:{seatId}:HELD`

Sequence 2: Heartbeat Extension for Slow Payment Gateways


Client
Booking Service
Redis
① Hold Seats (10 min TTL)
SET seat_hold:XXX TTL=600s
{holdToken, expiresAt}
... User initiates payment at gateway ...
... Payment processing slow (8 min elapsed) ...
② Heartbeat
PUT /hold/{token}/extend
③ Verify Hold
(status=ACTIVE, not expired)
Postgres
④ EXPIRE 600 (Add 10 min)
{newExpiresAt}

Core Architectural Decision

Database as Source of Truth

All booking confirmations validated via **conditional DB update** Prevents double booking under race conditions Redis used only for coordination, not durability Strong consistency enforced at booking boundary

Deterministic Expiry & Self-Healing

Seat holds enforced with TTL

Delayed expiry validation via background consumer

Periodic reconciliation job (Redis ↔ DB drift correction)

System remains consistent even after crashes

Distributed Locks

`SET NX EX` with unique lock token (UUID)

Lua-based safe unlock (ownership verification)

Short lock TTL (e.g., 30s) to avoid deadlocks

Designed to handle high-contention flash-sale scenarios

Idempotent by Design State Machine

The holdToken acts as a natural, single-use idempotency key for the confirm flow. A retry against an already-confirmed hold is rejected by the isActive() pre-check and the WHERE status='HELD' guard. domain state itself prevents duplicate effects

Key Architectural Trade-offs

PostgreSQL for Transactional Data

Strong ACID guarantees for bookings
Trade-off: Scaling requires partitioningΩ

Redis for Distributed State

Fast locks, TTL-based holds, Real-Time data
Trade-off: Additional infrastructure complexity

Pessimistic Locking + Distributed Locks

Prevent double booking at multiple layers, supports Redis unavailability
Trade-off: Retry logic, potential contention

Microservices Architecture

Independent scaling, fault isolation
Trade-off: Increased operational complexity

Event-Driven Async Processing

Decouple notifications, analytics
Trade-off: Eventual consistency

API Design - Complete User Journey

`GET /api/events/search` Browse events by city, date, category	200 OK with paginated results
`GET /api/events/{id}/seats` View seat layout and availability	200 OK with seat map
`POST /api/bookings/hold-seats` Hold selected seats (10 min TTL)	200 OK with hold ID
`POST /api/bookings/{hold-token}/confirm` Create booking from hold	201 Created with booking ref
`GET /api/bookings/{id}` Get booking details	200 OK with full details
`POST /api/bookings/hold/{hold-token}?customerid={castID}` Cancel hold & return to AVAILABLE pool	200 OK with clear hold

Failure Scenario

1. Service Crash After Lock Acquisition

- Scenario: Lock acquired in Redis, but DB write fails/crashes.
- Risk: Seat locked in Redis, not in DB (state drift).

Mitigation:

1. Persist DB immediately after lock (fail-fast).
2. On DB failure → Safe unlock using Lua script:
   if redis.call("GET", KEYS[1]) == ARGV[1] then
3.   return redis.call("DEL", KEYS[1])
4. end
5. Reconciliation job (every 5 min):
   - Find Redis locks without DB records → Delete lock.
   - Find DB HELD records without Redis locks → Recreate/release.

2. Redis Crash / Restart

- Scenario: All locks lost after Redis restart.
- Risk: DB and Redis state drift (DB=HELD, no lock).

Mitigation:

1. DB is source of truth; Redis is derived/cache.
2. Periodic reconciliation job:
   - DB=HELD & no Redis lock → Recreate lock or mark expired.
   - Redis lock & DB=AVAILABLE → Delete stale lock.
3. Use Redis persistence (AOF/RDB) for faster recovery.
4. Fallback: Use DB row locks if Redis is down (degraded mode).

Failure Scenario continued..

3. Lock Expiry During Payment Processing

Scenario: The distributed lock (e.g., Redis lock) expires before the full payment transaction is finalized.

Mitigation:

- Enforce mandatory database conditional confirmation:-

```
UPDATE seats SET status='BOOKED' WHERE id IN (1,2,3) AND status='HELD'
```

- Optionally implement a Time-To-Live (TTL) heartbeat mechanism to extend the lock duration for protracted payment processes.
- **The booking attempt must be rejected if the conditional database update fails**

Testing Strategy and Quality Assurance

```
┌──────────────────────────────┐
│     API / Contract Tests     │  ← End-to-End
│    (Postman, REST Assured)   │    Behavior
└──────────────────────────────┘
      ┌──────────────────────────┐
      │    Integration Tests     │  ← Component
      │    (@SpringBootTest)     │    Integration
      └──────────────────────────┘
            ┌─────────────────────────┐
            │       Unit Tests        │  ← Business
            │   (JUnit 5, Mockito)    │    Logic
            └─────────────────────────┘
```

Thank You