Scalable Event Ticketing & Seat Allocation — Complete System Design Document

---

Executive Summary

A production-ready design for a scalable event ticketing platform that supports flash-sales, prevents oversells, and provides low-latency checkout. The system guarantees **strong consistency** for seat commits while allowing **eventual consistency** for listings and analytics. This document includes requirements, architecture, APIs, data model, capacity planning, operational runbooks, security, testing, SLOs, and cost guidance.

---

1. Requirements Pack

1.1 Stakeholders & Prioritization

- **Buyers (P0):** Fast browsing, seat map, holds, checkout, refunds.
- **Event Organizers (P0):** Create/manage events, inventory, reporting.
- **Payments/Finance (P0):** Idempotent charges, reconciliation, audit logs.
- **Support/Ops (P1):** Observability, admin tools, incident playbooks.
- **Marketing (P2):** Promo management, analytics.

1.2 Functional Requirements (selected)

- FR1 — Browse/search events with filters and cached listings.
- FR2 — View interactive seating map with availability overlays.
- FR3 — Reserve seats with TTL (configurable, default 5m).
- FR4 — Commit reservation after successful payment (atomic seat commit).
- FR5 — Cancel/Refund flows and ticket delivery.
- FR6 — Idempotent payment and commit endpoints.
- FR7 — Audit trail for finance and reconciliation.

1.3 Non-functional Requirements

- **Latency:** p99 checkout (end-to-end) < 2s under normal load; p99 < 4s during flash sale.
- **Availability:** 99.95% for purchase flow.
- **Scalability:** Support bursts to tens of thousands req/s in peak windows.
- **Durability/Consistency:** No oversells; committed tickets durable.
- **Security:** PCI compliance for card handling; encryption in transit & at rest.

1.4 Constraints & Assumptions

- Event listings can be eventually consistent.
- Seat commit must be linearizable.
- Payment gateway provides idempotency support.

- Multi-region support desired; single selling currency per event recommended.

---

2. High-level Architecture

2.1 System Components

- **Clients:** Web, Mobile, Kiosk
- **Edge:** CDN (static assets), WAF
- **API Gateway:** Rate-limiting, auth, routing
- **Services:** Auth, Event Service, Seat Service, Order Service, Payment Service, Notification Service, Analytics Service
- **Infra:** Redis (cache), Seat DB (strong-consistency DB), Orders DB, Kafka/PubSub, Object Storage (assets)
- **Third parties:** Payment gateway (Stripe-like), Email/SMS providers

2.2 Data Flow (overview)

- Reads: Client → CDN → API Gateway → Cache → Service
- Reserve: Client → API Gateway → Seat Service → Seat DB (transaction) → Cache update → Order Service
- Commit: Payment success → Payment Service → Order Service → Seat Service commit → DB finalize → Notifications

---

3. Sequence Diagrams (text)

3.1 Reserve Seats (fast path)

```
User -> API GW -> Seat Service: reserveSeats(eventId, seatIds, userId)
Seat Service -> Redis: check availability
If Redis says available -> attempt DB transaction:
  DB: SELECT ... FOR UPDATE -> set status=HELD, hold_id, expires_at
Seat Service -> Redis: set held status with TTL
Seat Service -> Order Service: create provisional order (HOLD)
Return reservationId, expiresAt -> User
```

3.2 Commit Reservation

```
User -> Payment Service: pay(orderId, idempotencyKey)
Payment Service -> Payment Gateway: charge (idempotencyKey)
Gateway -> Payment Service: success
Payment Service -> Order Service: commit(orderId)
Order Service -> Seat Service: commitSeats(reservationId)
Seat Service -> DB: set status=COMMITTED, owner_order_id
```

```
Seat Service -> Kafka: publish seat_committed event
Order Service -> Notification Service: send tickets
```

---

4. API Contracts

4.1 Reserve Seats

`POST /v1/events/{eventId}/reservations` Body: `{ userId, seatIds: ["A1"], ttlSeconds: 300 }` Responses: - 200: `{ reservationId, expiresAt, heldSeats }` - 409: seat unavailable - 429: rate limited

4.2 Commit Reservation

`POST /v1/reservations/{reservationId}/commit` Body: `{ paymentId, idempotencyKey }` Responses: - 200: `{ orderId, tickets }` - 410: reservation expired - 409: seats unavailable (rare)

4.3 Cancel Reservation

`POST /v1/reservations/{reservationId}/cancel`

---

5. Data Model

5.1 Key Tables (DDL sketches)

Seats

```
CREATE TABLE seats (
  event_id UUID,
  seat_id VARCHAR,
  section VARCHAR,
  row VARCHAR,
  number INT,
  status VARCHAR CHECK (status IN ('AVAILABLE','HELD','COMMITTED')),
  hold_reservation_id UUID NULL,
  hold_expires_at TIMESTAMP NULL,
  owner_order_id UUID NULL,
  version BIGINT DEFAULT 0,
  PRIMARY KEY (event_id, seat_id)
);
```

Reservations

```
reservation_id UUID PRIMARY KEY,
user_id UUID,
event_id UUID,
seat_ids JSONB,
status VARCHAR CHECK(status IN ('HOLD','EXPIRED','COMMITTED')),
created_at,
expires_at
```

Orders

```
order_id UUID PRIMARY KEY,
reservation_id UUID,
user_id UUID,
amount_cents INT,
currency VARCHAR,
status VARCHAR,
payment_id VARCHAR,
created_at
```

---

6. Consistency & Concurrency Strategies

6.1 Seat Strong Consistency Options

- **Single-writer per partition (recommended):** Partition seats by eventId or section; assign a leader process for hot events to serialize writes.
- **Transactional DB (CockroachDB/Spanner):** Provide distributed transactions and linearizability across partitions.
- **Optimistic concurrency with CAS:** Use version field and CAS updates with retry loop.

6.2 Cache Strategy

- **Cache-aside** with short TTLs (5–15s) for seat availability.
- **Pub/Sub invalidation:** seat state changes publish events; cache consumers update/invalidate keys.
- **Fast-path reads from cache; authoritative writes to DB.**

---

7. Capacity Planning & Sizing (detailed)

7.1 Traffic Assumptions (example)

- Event seats: 50k
- Concurrent visitors: 1M
- Read-heavy ratio: 95% reads / 5% writes

- Peak commit attempts: 100k over initial 5 minutes → 333 req/s commit
- Peak total API requests: 16k–20k req/s

7.2 Sizing (example)

- API servers: 50 servers @400 req/s → baseline; 3x for HA and AZ spread → 150 instances.
- Redis: 8 nodes across shards to handle ~13k ops/s with redundancy.
- Seat DB: partitioned; provision for 1k writes/s with low tail latency; use provisioned IOPS and replicas.
- Kafka: 7 brokers, 50 partitions for hot topics; replication factor 3.

Note: Replace numbers with your measured traffic for accurate sizing.

---

8. Rate Limiting, Throttling & Fairness

- **API Gateway rate limits:** per-user (token bucket), per-IP, and global spikes cap.
- **Reservation quotas:** max holds per user per minute (e.g., 5/min).
- **Graceful throttling:** return informative 429 with retry-after; backoff guidance in client UI.

---

9. Resiliency & Failure Handling

9.1 Common Failures & Mitigations

- **Payment success, commit fails:** mark order `PAYMENT_SUCCESS_PENDING_COMMIT`, attempt retries, if impossible initiate refund and notify support.
- **Cache inconsistency:** pub/sub invalidation + short TTLs; reconcile via background sweep job.
- **DB hot-partition:** detect hot `eventId`; spin up dedicated shard or throttle non-essential traffic.

9.2 Retries & Circuit Breakers

- Exponential backoff for external calls (payment, email). Circuit breaker around payment gateway and DB.
- Idempotency keys to make retries safe.

---

10. Observability & SRE Practices

10.1 Metrics

- Request rates, p50/p90/p99 latencies per endpoint.
- Reservation holds created/expired, commit success rate.
- Payment failure rate, external call latencies.
- Queue lag and consumer throughput.

10.2 Tracing & Logging

- Distributed tracing (OpenTelemetry). Correlate traceId across services.
- Structured logs (JSON) with audit fields for order/seat state changes.

10.3 Alerts & Runbooks

- Example alerts: payment-failure-rate >1% (5m), seat DB p99 write latency >200ms (3m), queue lag >30s.
- Runbooks: hot-partition mitigation, payment partial success, reservation leak cleanup.

---

11. Security & Compliance

- **PCI:** Do not store card PANs; use tokenization. Scope minimized by keeping payment flows in PCI-certified environment.
- **Auth:** OIDC/OAuth2 for users; RBAC for admin/organizer consoles.
- **Secrets:** Cloud KMS/Secret Manager; rotate regularly.
- **Encryption:** TLS 1.2+ in transit; AES-256 at rest.
- **Audit logs:** Immutable audit trail for payments and seat commits.

---

12. Testing & Launch Strategy

- **Load testing:** k6/Locust with realistic flash-sale traffic (95% reads). Ramp and soak tests.
- **Chaos engineering:** Simulate AZ failures, DB leader failover, and payment gateway slowdowns.
- **Canary deploys:** Validate small percentage traffic before global rollout.
- **Staging dry-run:** Run a real dry-run with synthetic users prior to live flash sale.

---

13. Operational Runbooks (summaries)

13.1 Hot Partition Procedure

1. Identify `eventId` causing high load.
2. Add dedicated cache shard and route reads.
3. Throttle write-heavy operations for non-paying users.
4. Spin up dedicated DB shard or isolate event on a higher-capacity cluster.

13.2 Payment Partial Success

1. Tag order as `PAYMENT_UNKNOWN`.
2. Call payment gateway reconciliation API.
3. If charge captured but seats unavailable: attempt commit retry; else initiate refund and notify user & support.

13.3 Reservation Leak Recovery

1. Periodic job scans holds past expiry and releases seats.
2. Alert if leaked holds rate > threshold and investigate root cause.

---

14. Monitoring Thresholds & SLOs

- **SLOs:** 99.95% availability for purchase flow; p99 checkout latency < 2s.
- **Error budget:** track monthly; alert on burn >50%.
- **Key alerts:** payment-failure-rate, seat DB p99 latency, reservation expiry spike.

---

15. Cost Estimate (example)

- App servers (150) — $30K/month
- Redis — $4K/month
- Seat DB cluster — $20K/month
- Kafka (managed) — $5K/month
- CDN & bandwidth — $3K/month
- Monitoring & logs — $2K/month
- **Total (approx):** $64K/month

---

16. Data Retention & Privacy

- **Financial records:** retain per legal requirements (e.g., 7 years).
- **PII:** retention policy and erasure workflow (anonymize while keeping financial auditability).
- **Access controls:** restrict access to PII and payments data; log access.

---

17. Edge Cases & Trade-offs

- **Partial-commit handling:** prefer full-commit-or-fail to simplify invariants.
- **Promo race conditions:** use atomic DB claim or strong counters.
- **Global scaling trade-off:** strong global consistency (Spanner) vs cost and complexity; pick based on business needs.

---

18. Next Steps & Deliverables

- Replace example capacity numbers with your real traffic profile for precise sizing.
- I can: add architecture PNG diagrams, export PDF, produce PPTX slides, or generate runbook markdown files.

End of Document