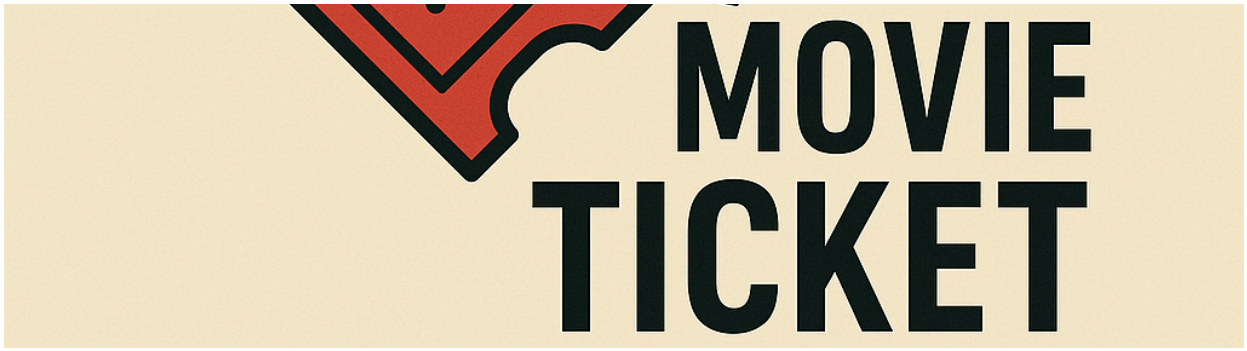
Overview



Status Draft

Tech Manager Chandrashekhar Vishwakarma

Movie Ticket Booking Platform — Technical Design & Architecture Document

Scope: Complete technical design for a B2B + B2C movie ticket booking platform built on AWS with a goal of **99.99% availability**. This document consolidates the design, patterns, data model, APIs, operational runbook, and deployment details. It also aligns with the user-supplied HLD/LLD/ERD and cost/ops notes.

1. Executive summary (Goals & constraints)

Primary goals:

- High availability (99.99%)
- low-latency booking
- consistent seat inventory handling across regions
- multi-country support
- secure payments and PCI compliance

- cost-efficient at scale.

Key non-functional requirements:

- P95 API latency < 200ms, booking confirmation < 3s
- booking success rate > 99%
- cache hit ratio > 90%, horizontal scalability
- multi-region active-active with automatic failover.

- **Traffic assumptions (design targets):** 10M active users, 1M bookings/month, peak concurrent users 100k+, target >1k bookings/sec at peak.

https://github.com/kznchandra/movie-ticket-booking

HLD Architecture

2. High-level architecture (components & AWS mappings)

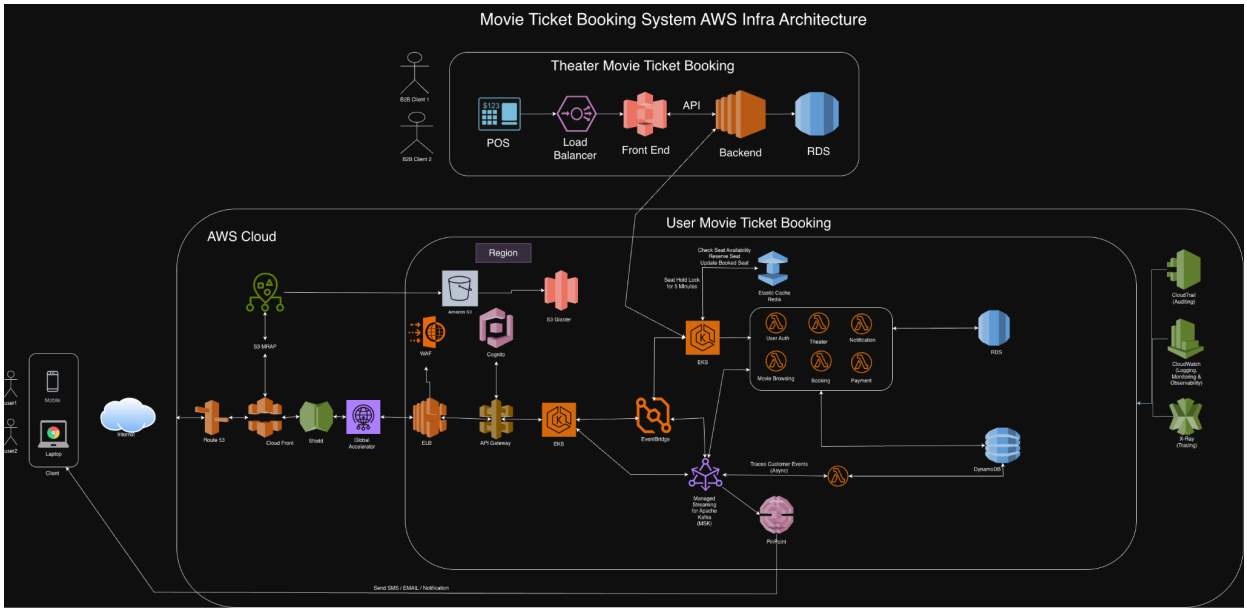
Summary flow (user → services):

1. Client (mobile / web) → Route53 → CloudFront (static + SPA) → AWS Global Accelerator / API Gateway → Regional ELB → EKS (microservices) / Lambda → Data stores (RDS/DynamoDB/Redis) + Event bus (MSK/EventBridge) → Notifications (Pinpoint/SNS/SES) → Observability & Security.

Key AWS services:

- Edge & DNS: **Route 53**, **CloudFront**, **Global Accelerator**
- WAF & DDoS: **AWS WAF**, **AWS Shield Advanced**
- API / Ingress: **API Gateway** (edge or regional), **Elastic Load Balancer (ALB)** fronting **EKS**.
- Compute: **Amazon EKS** for stateless & stateful microservices; **AWS Lambda** for asynchronous workers, webhooks, small tasks.
- Caching & locks: **Amazon ElastiCache for Redis (Cluster Mode)** — seat locks, session caching, leaderboard/metadata.
- Primary RDBMS: **Amazon RDS (Postgres / Aurora Postgres)** — transactional booking data, theatre metadata (multi-AZ).
- NoSQL: **Amazon DynamoDB** — high-scale lookup data (e.g., user session index, booking details, idempotency store).
- Streaming & events: **Amazon MSK (Kafka)** for event-driven flows + **Amazon EventBridge** for cross-account/region events.
- Messaging/notifications: **Amazon Pinpoint**

- Storage: **Amazon S3** (static assets, DL archives, backups), **Glacier** for archival
- Observability: **CloudWatch**, **X-Ray**, **CloudTrail**, optional APM (Datadog/NewRelic).
- Security: **AWS KMS** for keys, **Cognito** for user auth (or custom OIDC), **AWS Secrets Manager** for credentials, **IAM** policies and SCPs.
- Logging: Centralized logs via **CloudWatch** and an ELK / OpenSearch stack if required.



LLD and Work Flow

3. Detailed design

3.1 Deployment topology — Multi-region active-active

- Deploy full stack into at least 2 active regions per major geography (e.g., ap-south-1 & ap-northeast-1, us-east-1 & us-west-2, eu-west-1 & eu-central-1).
- **Route53** uses **Latency + Geolocation** policies to send users to nearest healthy region. Health checks and failover policies configured per endpoint.
- **Global Accelerator** for a single static anycast IP and optimal routing to the nearest edge and regional endpoint.
- Data residency: some data (PII) is region-bound; design each region to accept local writes when necessary and replicate/synchronize where allowed.

3.2 Microservice boundaries

- **Auth Service** (Cognito or custom) — user sign-up, MFA, tokens (OIDC).
- **Theatre Management (B2B)** — theatre onboarding, screen/seat mapping, schedule management.
- **Movie Browsing** — catalog, search (Elasticsearch/OpenSearch), metadata.
- **Booking Service** — booking lifecycle (initiate, hold, confirm, cancel, refund). **Core transactional service**.
- **Payment Service** — integrates with PCI-compliant payment gateway (Stripe/Adyen etc.) — tokenize card data, not store CVV.
- **Notification Service** — email, SMS, push via Pinpoint/SNS/SES.
- **Reservation Locker** — seat locking & TTL using Redis (fast).

- **Reporting / Analytics** — streaming consumer that writes to data lake (S3/Glue/Redshift) or OLAP.

- **Billing & Invoicing** — partner settlements for B2B.

- **Admin Panel** & **B2B Portal** — separate UI & APIs for theatre partners.

3.3 Booking flow (HLD -> LLD)

Synchronous + Event-driven hybrid:

1. **Browse & Select Seats:** Client requests seat map; service retrieves seat availability aggregated in Redis (fast). If cache miss, read DB and populate cache.

2. **Initiate Booking / Reserve Seats (fast path):**

    - Client calls POST /bookings/initiate with showId, seatIds, userId.

    - Booking Service attempts to **acquire locks** on seatIds via Redis **SETNX** or Lua script (atomic): lock key = lock:showId:seatId with TTL (e.g., 5 minutes).

    - If all locks acquired, create a *provisional booking record* with status PENDING_PAYMENT in RDS (transaction), and store idempotency token.

    - Return booking reference + payment token to client; start countdown TTL.

3. **Payment:** Client performs payment through Payment Service (tokenized). Payment Service returns success/failure and emits event to MSK and EventBridge.

4. **Confirm Booking:** On payment-success event:

    - Booking Service reconfirms locks; marks booking CONFIRMED in RDS, persists final seat occupancy, releases locks, writes event to MSK for analytics.

    - Notify user via Notification Service.
    - Reserve data synchronized to DynamoDB/Cache to support fast reads.

5. **Payment Failure / Timeout:**
    - If payment fails or timeouts: Booking Service cancels PENDING_PAYMENT, releases seat locks (Redis), updates DB; notify user. The seat becomes available.

6. **Idempotency & Retry:** All write endpoints accept idempotency tokens and are idempotent. Payment webhooks and asynchronous event consumers must handle duplicate events.

Seat lock implementation details:

- Use Redis cluster with strong consistency in each region.
- Use Lua script to atomically check and set multiple seat locks in a single call (avoid race).
- Use TTL <= maximum payment timeout (e.g., 5–15 minutes). TTL expiry automatically frees seats.

3.4 Concurrency & consistency patterns

- **Booking transaction** uses *pessimistic locking* at seat-level (via Redis) to prevent double allocation.
- **Eventual consistency** for secondary views (analytics, search index) via Kafka/MSK consumers.
- **Strong consistency** for final booking state kept in RDS/Aurora with ACID transactions.
- **Read scaling**: read replicas and read-only caches (ElastiCache + CDN).

- **Concurrent booking across regions:** region-local seat locking, cross-region replication of booking events to central store. For global shows, use a single region ownership or central seat master to avoid split-brain for the same show. Prefer sharding shows to region owners per theatre (single source of truth).

3.5 Data stores & schema

- **Primary transactional DB:** Aurora PostgreSQL (Multi-AZ, read replicas). Partition bookings by region/country or by theatre (LIST partitioning) for scale and locality. Example table snippet (from your doc):

```
CREATE TABLE bookings (
 id BIGSERIAL PRIMARY KEY,
 booking_reference VARCHAR(50) UNIQUE,
 user_id UUID,
 theatre_id UUID,
 show_id UUID,
 seats JSONB,
 amount_cents BIGINT,
 currency CHAR(3),
 status VARCHAR(20),
 created_at TIMESTAMP,
 updated_at TIMESTAMP
) PARTITION BY LIST (theatre_region);
```

Index Created By all Region Patrion Query

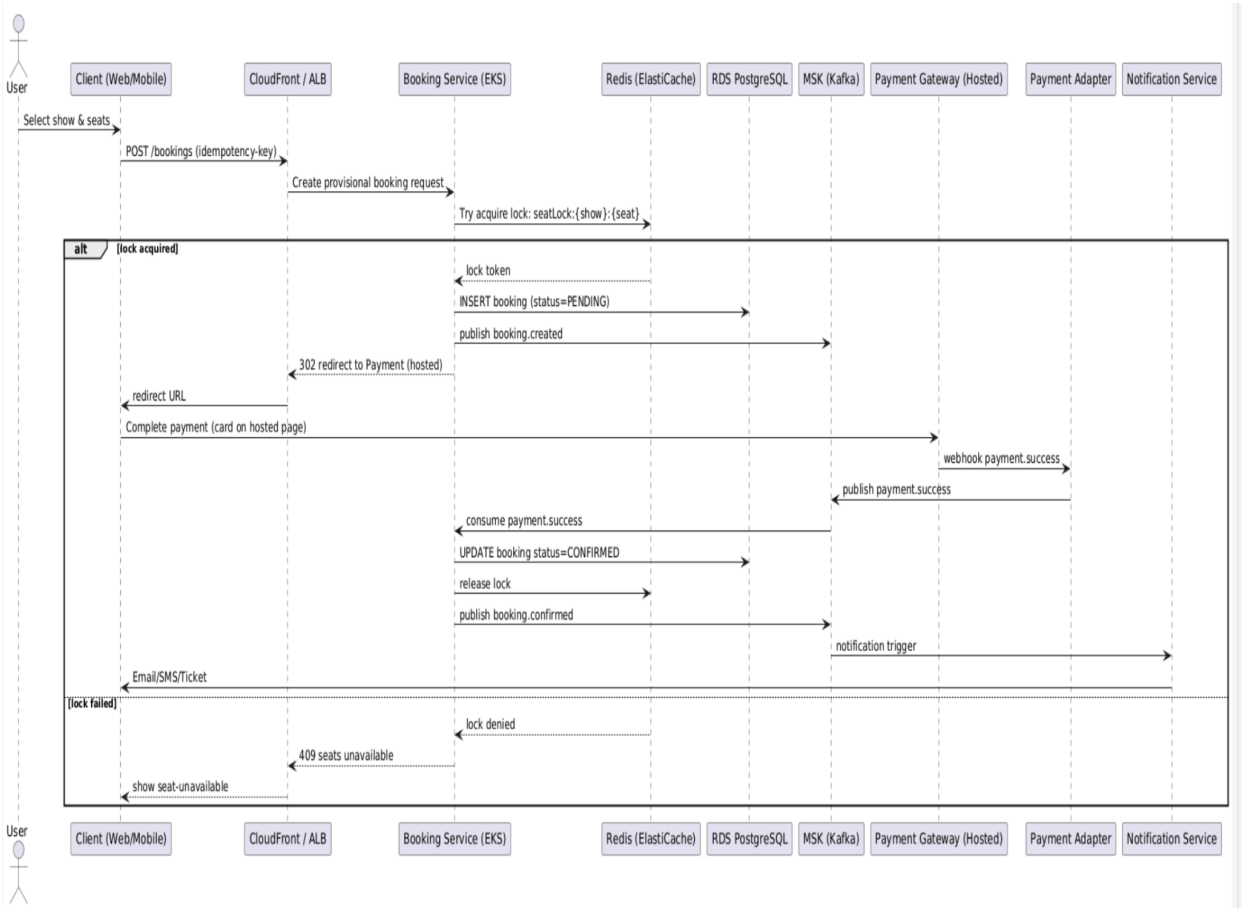
```
CREATE TABLE bookings_ind_nr PARTITION OF bookings FOR VALUES IN
('DEL-EST', 'DEL-WST');
CREATE TABLE bookings_eu PARTITION OF bookings FOR VALUES IN
('GB','FR','DE');
CREATE TABLE bookings_asia PARTITION OF bookings FOR VALUES IN ('CH',
'SG','JP');
```

- **DynamoDB:** idempotency token store, quick lookup items, session index for mobile connectivity; global tables if needed cross-region.
- **Redis (ElastiCache):** seat locks, seat-availability cache (show inventory), session cache, rate-limit counters.
- **OpenSearch / Elasticsearch:** movie & theatre search & filter.
- **S3:** static content, backups, uploads, DL exports.
- **Data lake:** S3 + Glue for analytics.

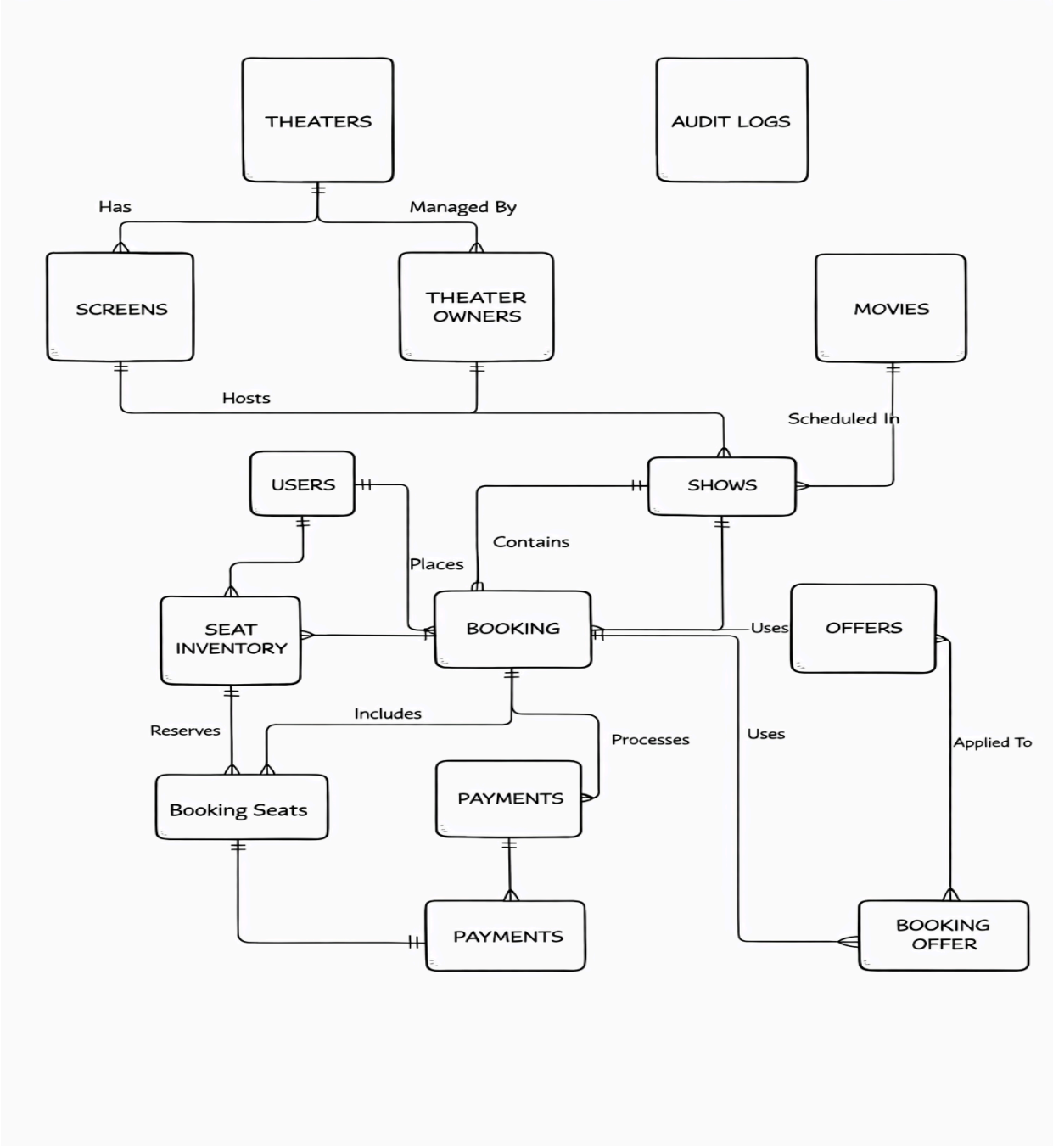
3.6 API contract (summary)

- POST /v1/bookings/initiate — body: {userId, showId, seatIds[], idempotencyKey} → returns bookingReference, expiresAt, paymentUrl.
- POST /v1/bookings/confirm — body: {bookingReference, paymentId} → returns booked details.
- GET /v1/shows/{showId}/seats — returns seat map with status (available/held/booked).
- POST /v1/theatres (B2B) — onboard theatre; requires admin auth.
- POST /v1/payments/webhook — receives payment provider webhook (idempotent).

All APIs must support **idempotencyKey**, throttling headers, tracing headers (X-Trace-Id).


User
Client (Web/Mobile)
CloudFront / ALB
Booking Service (EKS)
Redis (ElastiCache)
RDS PostgreSQL
MSK (Kafka)
Payment Gateway (Hosted)
Payment Adapter
Notification Service
Select show & seats
POST /bookings (idempotency-key)
Create provisional booking request
Try acquire lock: seatLock:{show}:{seat}
alt
[lock acquired]
lock token
INSERT booking (status=PENDING)
publish booking.created
302 redirect to Payment (hosted)
redirect URL
Complete payment (card on hosted page)
webhook payment.success
publish payment.success
consume payment.success
UPDATE booking status=CONFIRMED
release lock
publish booking.confirmed
notification trigger
Email/SMS/Ticket
[lock failed]
lock denied
409 seats unavailable
show seat-unavailable
User
Client (Web/Mobile)
CloudFront / ALB
Booking Service (EKS)
Redis (ElastiCache)
RDS PostgreSQL
MSK (Kafka)
Payment Gateway (Hosted)
Payment Adapter
Notification Service

ER Diagram & Database Schema Design




```
CREATE TABLE theater_owners (
                                owner_id BIGSERIAL PRIMARY KEY,
                                name VARCHAR(100) NOT NULL,
                                email VARCHAR(150) UNIQUE NOT NULL,
                                phone VARCHAR(20),
                                created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
```

```
);

CREATE TABLE theaters (
                        theater_id BIGSERIAL PRIMARY KEY,
                        owner_id BIGINT NOT NULL,
                        name VARCHAR(150) NOT NULL,
                        city VARCHAR(100) NOT NULL,
                        country VARCHAR(50) NOT NULL,
                        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                        FOREIGN KEY (owner_id) REFERENCES
theater_owners(owner_id)
);

CREATE TABLE screens (
                       screen_id BIGSERIAL PRIMARY KEY,
                       theater_id BIGINT NOT NULL,
                       screen_name VARCHAR(50) NOT NULL,
                       total_seats INT NOT NULL,
                       created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                       FOREIGN KEY (theater_id) REFERENCES
theaters(theater_id)
);

CREATE TABLE movies (
                      movie_id BIGSERIAL PRIMARY KEY,
                      title VARCHAR(200) NOT NULL,
                      language VARCHAR(50),
                      duration_minutes INT NOT NULL,
                      rating VARCHAR(10),
                      release_date DATE,
                      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE shows (
                     show_id BIGSERIAL PRIMARY KEY,
                     movie_id BIGINT NOT NULL,
                     screen_id BIGINT NOT NULL,
                     show_start_time TIMESTAMP NOT NULL,
                     show_end_time TIMESTAMP NOT NULL,
```

```
                    base_price NUMERIC(10,2) NOT NULL,
                    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    FOREIGN KEY (movie_id) REFERENCES movies(movie_id),
                    FOREIGN KEY (screen_id) REFERENCES
screens(screen_id),
                    UNIQUE (screen_id, show_start_time)
);

CREATE TABLE users (
                    user_id BIGSERIAL PRIMARY KEY,
                    name VARCHAR(100),
                    email VARCHAR(150) UNIQUE NOT NULL,
                    phone VARCHAR(20),
                    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE seat_inventory (
                            seat_id BIGSERIAL PRIMARY KEY,
                            screen_id BIGINT NOT NULL,
                            seat_number VARCHAR(10) NOT NULL,
                            seat_type VARCHAR(20), -- REGULAR, PREMIUM,
RECLINER
                            created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP,
                            FOREIGN KEY (screen_id) REFERENCES
screens(screen_id),
                            UNIQUE (screen_id, seat_number)
);

CREATE TABLE bookings (
                        booking_id BIGSERIAL PRIMARY KEY,
                        booking_reference VARCHAR(50) UNIQUE NOT NULL,
                        user_id BIGINT NOT NULL,
                        show_id BIGINT NOT NULL,
                        status VARCHAR(30) NOT NULL,
   -- INITIATED, CONFIRMED, CANCELLED, EXPIRED
                        total_amount NUMERIC(10,2),
                        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
                        FOREIGN KEY (user_id) REFERENCES users(user_id),
                        FOREIGN KEY (show_id) REFERENCES shows(show_id)
);

CREATE TABLE booking_seats (
                            booking_seat_id BIGSERIAL PRIMARY KEY,
                            booking_id BIGINT NOT NULL,
                            seat_id BIGINT NOT NULL,
                            price NUMERIC(10,2) NOT NULL,
                            FOREIGN KEY (booking_id) REFERENCES
bookings(booking_id),
                            FOREIGN KEY (seat_id) REFERENCES
seat_inventory(seat_id),
                            UNIQUE (seat_id, booking_id)
);

CREATE TABLE payments (
                          payment_id BIGSERIAL PRIMARY KEY,
                          booking_id BIGINT NOT NULL,
                          payment_gateway VARCHAR(50),
                          transaction_reference VARCHAR(100) UNIQUE,
                          amount NUMERIC(10,2) NOT NULL,
                          payment_status VARCHAR(30),
   -- SUCCESS, FAILED, PENDING, REFUNDED
                          created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                          FOREIGN KEY (booking_id) REFERENCES
bookings(booking_id)
);

CREATE TABLE offers (
                        offer_id BIGSERIAL PRIMARY KEY,
                        offer_code VARCHAR(50) UNIQUE NOT NULL,
                        discount_type VARCHAR(20), -- PERCENTAGE, FLAT
                        discount_value NUMERIC(10,2),
                        valid_from DATE,
                        valid_to DATE,
                        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```sql
CREATE TABLE booking_offers (
                                booking_offer_id BIGSERIAL PRIMARY KEY,
                                booking_id BIGINT NOT NULL,
                                offer_id BIGINT NOT NULL,
                                discount_amount NUMERIC(10,2),
                                FOREIGN KEY (booking_id) REFERENCES
bookings(booking_id),
                                FOREIGN KEY (offer_id) REFERENCES
offers(offer_id),
                                UNIQUE (booking_id, offer_id)
);

CREATE TABLE audit_logs (
                            audit_id BIGSERIAL PRIMARY KEY,
                            entity_type VARCHAR(50),
                            entity_id BIGINT,
                            action VARCHAR(50),
                            performed_by VARCHAR(100),
                            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_show_time ON shows(show_start_time);
CREATE INDEX idx_booking_user ON bookings(user_id);
CREATE INDEX idx_booking_status ON bookings(status);
CREATE INDEX idx_payment_booking ON payments(booking_id);
```

Reliability & 99.99% Availability strategy

4. Reliability & 99.99% Availability strategy

4.1 Multi-region active-active

- Deploy to >=2 active regions per geography. Use **Route53** and **Global Accelerator** for traffic steering and failover.

- Stateful components (RDS/Aurora) are replicated via cross-region read replicas and asynchronous replication for DR. Consider **Aurora Global Database** for low-latency cross-region read and fast recovery.

4.2 Availability patterns

- **Autoscaling** for EKS worker nodes (Karpenter / Cluster Autoscaler) and HPA for pods.

- **Use multi-AZ** for every stateful layer (RDS Multi-AZ, ElastiCache across AZs).

- **Circuit breakers** and bulkhead patterns in services to isolate failures.

- **Chaos Engineering** and regular failure drills.

4.3 Failover timelines (example)

- Health check detects region failure (10s).

- Route53 failover shift (TTL dependent): use low TTL (30s) + Global Accelerator reduces global time to failover.

- Warmed replicas in DR region (autoscaling) spin up within minutes; use pre-warmed capacity for critical paths.

4.4 Disaster recovery

- **Backups:** continuous automated snapshots (RDS), S3 lifecycle to Glacier.
- **RTO** & **RPO** targets: RTO < 15 minutes for core services, RPO < few minutes using cross-region replication and event streaming.
- **Runbook:** documented playbooks for DB failover, DNS failover, and manual rollback.

Scalability & performance optimizations

5. Scalability & performance optimizations

5.1 Caching

- Multi-layer caching: CloudFront (frontend), API Gateway caching, ElastiCache Redis, application-level caches.
- TTLs: inventory short TTLs (30s), show metadata 5–15 mins.

5.2 DB & query optimizations

- Partition bookings by region/theatre. Use composite indexes on (show_id, seat_id).
- Read replicas for read-heavy traffic, write masters per shard.
- Use prepared statements, batching, and connection pooling (HikariCP tuned as in doc).

5.3 Async processing

- Use MSK for events: booking.initiated, booking.confirmed, payment.success, booking.cancelled.
- Consumers: notification, settlement, analytics, search-index updates — decoupled for resilience.

5.4 Cost-optimizations

- Reserved instances & Savings Plans as baseline.

- Use spot instances for batch/analytics.
- Right-size DB instance classes and cache sizes.
- Archive old bookings to S3/Glacier.

5. Scalability & performance optimizations

5.1 Caching

- Multi-layer caching: CloudFront (frontend), API Gateway caching, ElastiCache Redis, application-level caches.
- TTLs: inventory short TTLs (30s), show metadata 5–15 mins

5.2 DB & query optimizations

- Partition bookings by region/theatre. Use composite indexes on (show_id, seat_id).
- Read replicas for read-heavy traffic, write masters per shard.
- Use prepared statements, batching, and connection pooling (HikariCP tuned as in doc).

5.3 Async processing

- Use MSK for events: booking.initiated, booking.confirmed, payment.success, booking.cancelled.
- Consumers: notification, settlement, analytics, search-index updates — decoupled for resilience.

5.4 Cost-optimizations

- Reserved instances & Savings Plans as baseline.

- Use spot instances for batch/analytics.
- Right-size DB instance classes and cache sizes.
- Archive old bookings to S3/Glacier.

Security & compliance

6. Security & compliance

6.1 PCI & payment

- Use third-party PCI-compliant payment providers. Tokenize card data; do not store CVV nor raw PAN.
- Quarterly scans & annual audit.

6.2 Network & data security

- Use **TLS 1.3** for all traffic. mTLS for service-to-service if feasible.
- IAM least privilege; use roles for services and KMS for encryption.
- PII: encrypt at rest (AES-256 via KMS), store EU data only within EU regions to meet GDPR.

6.3 DDoS protection & WAF

- AWS Shield Advanced + WAF rules to block common vectors.
- Rate limiting per IP and per user (API Gateway + app-layer counters in Redis).

6.4 Monitoring & observability

- Tracing: AWS X-Ray + distributed tracing headers.
- Metrics: CloudWatch custom metrics (booking success, lock latencies, cache hit ratio, DB Q time).

- Logging: centralized logs with retention & alerting. Set alerts for error spikes and availability drops.

- Security monitoring: CloudTrail for audit, GuardDuty for threats.

Operations, SLOs, Runbooks & Alerts

7. Operations, SLOs, Runbooks & Alerts

7.1 SLO / SLA targets

- Availability: 99.99% (monthly).
- Booking success: > 99%.
- P95 latency: < 200ms.

7.2 Key alerts & thresholds

- **Critical:** API error rate >1% for 5m, RDS connection exhausted, payment gateway down, overall availability <99.9%.
- **Warning:** P95 > 500ms, CPU > 80% for 10 min, Cache hit < 80%.
- **Info:** slow queries, unusual traffic patterns.

7.3 Runbooks (examples)

- **Payment gateway outage:** divert users to retry queue + show appropriate UI message; notify SRE; page on-call.
- **DB failover:** promote cross-region replica using Aurora Global DB failover playbook; run consistency checks.
- **Cache cluster failure:** failover to replicas; if full Redis cluster down, fallback to DB reads with throttling.

Observability & testing

8. Observability & testing

- Canary deployments + feature flags for new features.
- Synthetic monitoring (booking flows) executed from multiple geographies.
- Load testing: run pre-release load tests to validate >1k bookings/sec.
- Chaos experiments: periodic pod/region failure tests.

Cost estimate

9. Cost estimate

Monthly baseline (10M users, 1M bookings): **~$50k–$60k** (compute, DB, cache, storage, networking). With reserved instances & savings: **~$35k/month**. (See your uploaded cost breakdown for details.)

Low Level Artifacts

10. Low-level artifacts (LLD pointers & code snippets)

10.1 Redis seat-lock Lua (pseudo)

```
-- Attempt to lock multiple seats atomically
-- KEYS: lock keys for each seat
-- ARGV[1] = lock value (bookingRef)
-- ARGV[2] = ttl seconds
for i=1,#KEYS do
 if redis.call('exists', KEYS[i]) == 1 then
    return 0
 end
end
for i=1,#KEYS do
 redis.call('set', KEYS[i], ARGV[1], 'EX', ARGV[2])
end
return 1
```

- Release script checks ownership before deleting lock.

10.2 Booking table partition example

```
CREATE TABLE bookings (
 id BIGSERIAL PRIMARY KEY,
 booking_reference VARCHAR(50) UNIQUE,
 theatre_region VARCHAR(10),
 user_id UUID,
 show_id UUID,
 seats JSONB,
 status VARCHAR(20),
 created_at TIMESTAMP DEFAULT now()
```

```
) PARTITION BY LIST (theatre_region);
```

- Release script checks ownership before deleting lock.

Sequence diagrams Booking success & failure

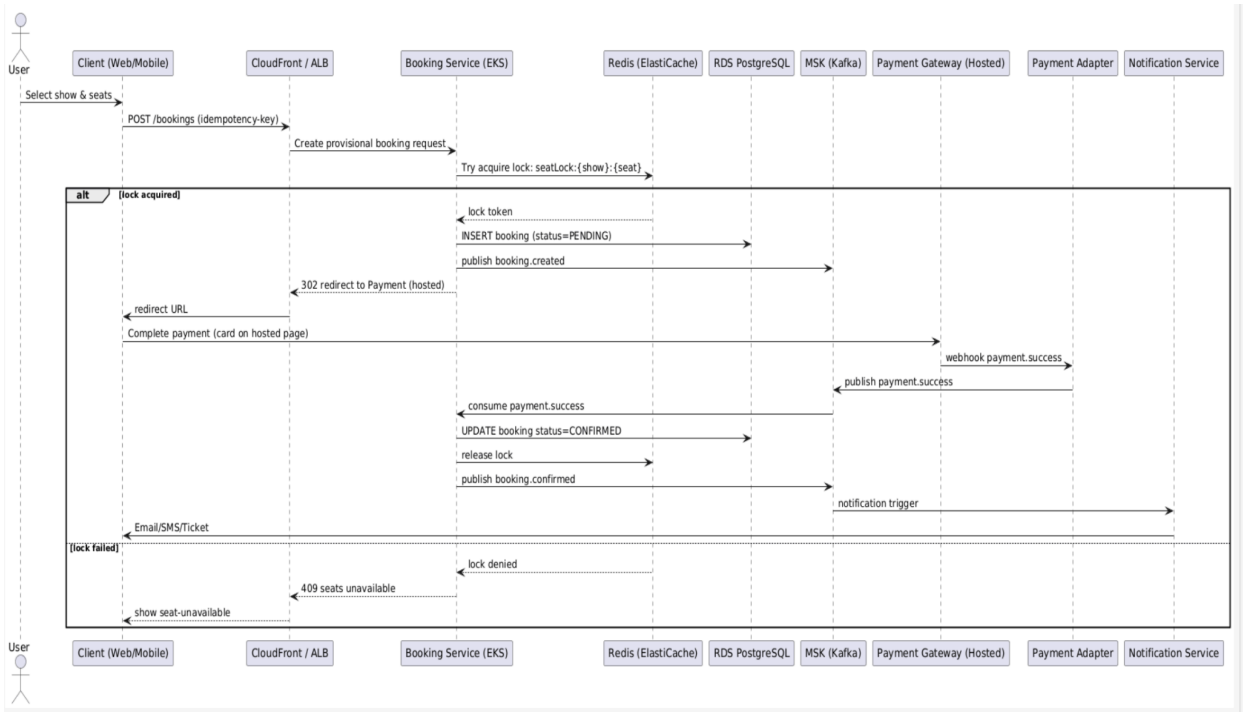
11. Sequence diagrams (textual) — Booking success & failure

Success path

1. Client → BookingService /initiate (idempotencyKey)
2. BookingService → Redis (lock seats via Lua)
3. BookingService → RDS create PENDING_PAYMENT booking
4. BookingService → Payment gateway (create payment session)
5. Client completes payment → Payment gateway → webhook → PaymentService
6. PaymentService emits payment.success to MSK/EventBridge
7. BookingService consumes payment.success → mark booking CONFIRMED in RDS → release seat lock
8. BookingService → NotificationService → send ticket
9. BookingService → produces booking.confirmed event for analytics

Timeout/failure

- After TTL or payment.failed, BookingService cancels booking and releases locks; notifications sent.


User
Client (Web/Mobile)
CloudFront / ALB
Booking Service (EKS)
Redis (ElastiCache)
RDS PostgreSQL
MSK (Kafka)
Payment Gateway (Hosted)
Payment Adapter
Notification Service
Select show & seats
POST /bookings (idempotency-key)
Create provisional booking request
Try acquire lock: seatLock:{show}:{seat}
alt
[lock acquired]
lock token
INSERT booking (status=PENDING)
publish booking.created
302 redirect to Payment (hosted)
redirect URL
Complete payment (card on hosted page)
webhook payment.success
publish payment.success
consume payment.success
UPDATE booking status=CONFIRMED
release lock
publish booking.confirmed
notification trigger
Email/SMS/Ticket
[lock failed]
lock denied
409 seats unavailable
show seat-unavailable
User
Client (Web/Mobile)
CloudFront / ALB
Booking Service (EKS)
Redis (ElastiCache)
RDS PostgreSQL
MSK (Kafka)
Payment Gateway (Hosted)
Payment Adapter
Notification Service

Testing & validation

12. Testing & validation

- **Unit & Integration tests** for seat locking, transactions, and idempotency.
- **End-to-end** tests for booking scenarios (concurrent holds).
- **Load tests** to validate expected TPS and latency.
- **Security tests** including penetration testing and PCI compliance scans.

Migration & rollout plan

13. Migration & rollout plan

1. Canary release in a single region with low traffic.
2. Validate metrics & error budgets.
3. Gradually increase traffic (10% → 50% → 100%) using Global Accelerator routing weights.
4. Perform DR & failover runbooks during maintenance windows.

Next Steps / Deliverables

14. Next steps / deliverables

- Produce final **infrastructure Terraform/Terragrunt** templates for EKS, RDS (Aurora Global), MSK, ElastiCache, API Gateway, Route53.
- Provide EKS Helm charts & Kubernetes manifests (service meshes optional).
- Implement core microservices (Booking, Payment, Notification) with proper observability hooks.
- Prepare runbooks and SRE playbooks (DB failover, region failover).
- Schedule a full load test & chaos test before production cutover.

Appendix

15. Appendix (references from provided document)

Partitioning strategy, Redis configuration, metrics & alert thresholds, and estimated costs are taken and adapted from the uploaded design and cost notes.