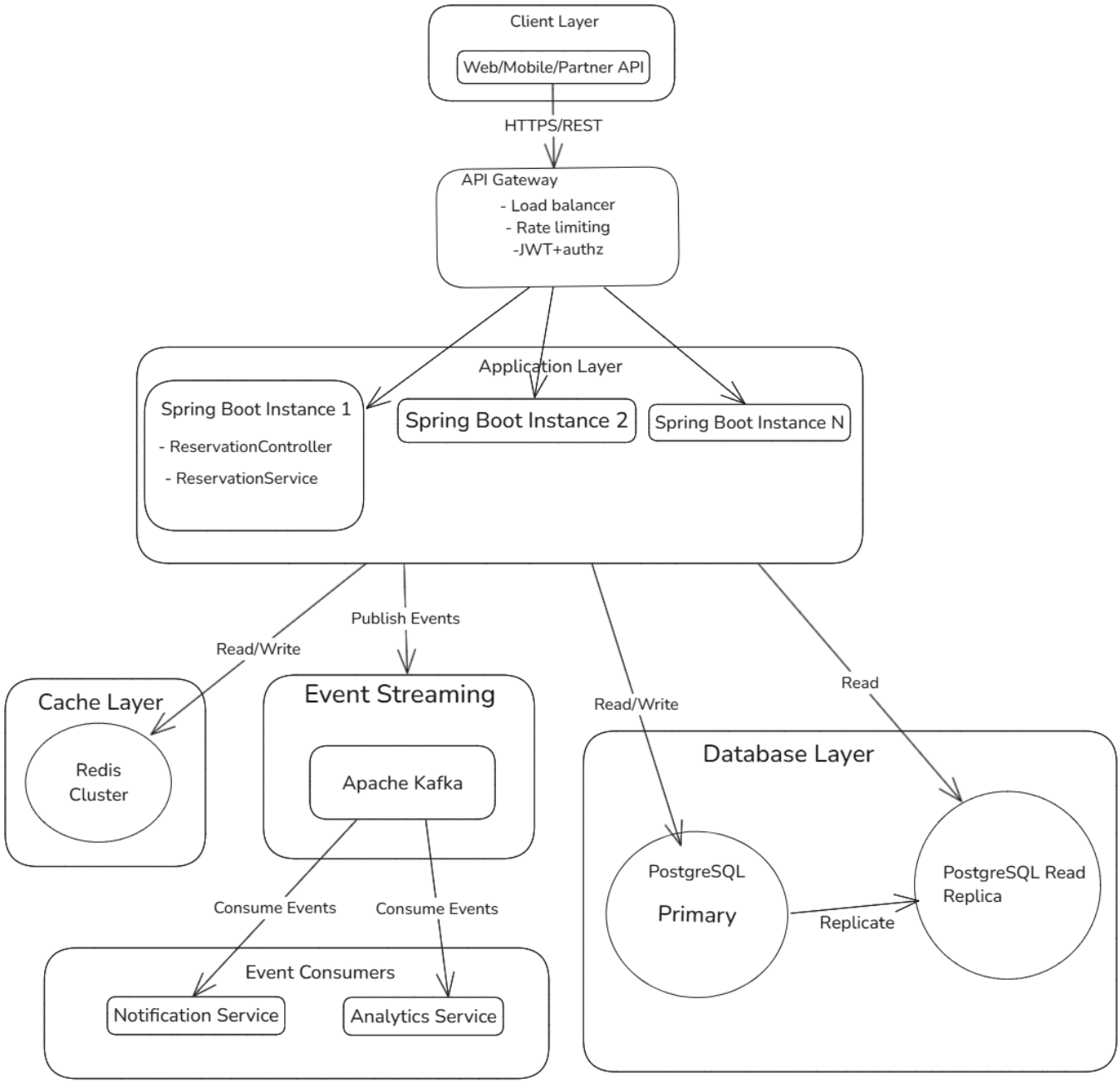
Private Dining Reservation System

1. System Overview

Architecture Principles

- **Double-booking prevention at DB level**: Partial unique index on (room_id, reservation_date, time_slot) enforces this atomically - even buggy application code can't bypass it
- **Stateless design**: No server-side sessions, so multiple instances can handle bookings without coordination
- **Async everything except the booking**: Confirmation emails and audit logs happen in the background and don't block the reservation response

Production Architecture



MVP (Current - ~100 req/sec):

- Single Spring Boot instance
- Spring Events for email notifications
- In-memory cache for restaurant/room catalog
- Single PostgreSQL for all reservation data

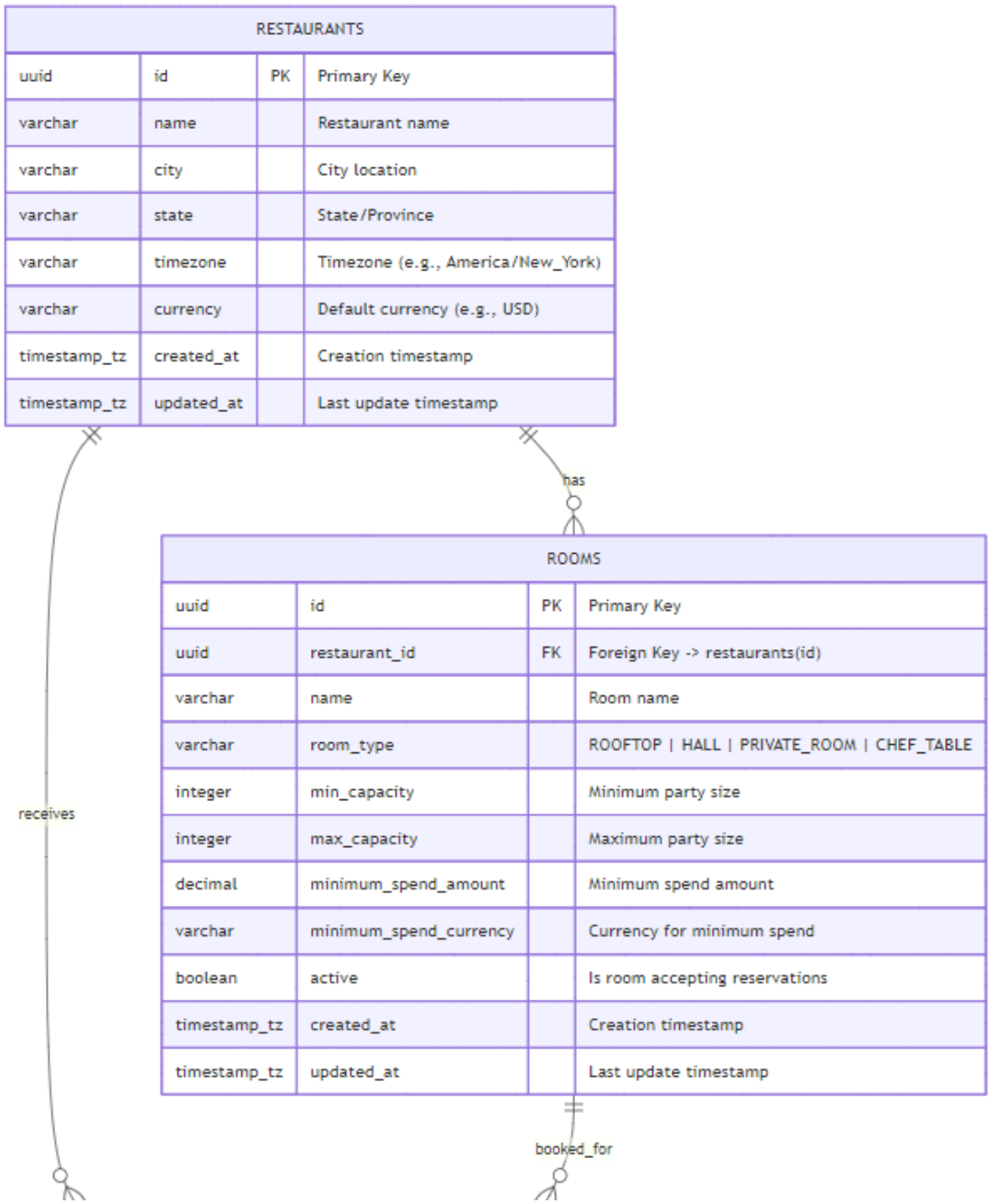
Production (Target - ~1000 req/sec):

- 3+ Spring Boot instances behind load balancer
- Kafka for notification/audit events
- Redis for restaurant/room/availability caching
- PostgreSQL primary (writes) + read replicas (availability queries)

When to Scale: If sustained load > 70 req/sec for 1 week OR expanding to 100+ restaurants

---

2. Database Schema



RESERVATIONS			
uuid	id	PK	Primary Key
uuid	restaurant_id	FK	Foreign Key -> restaurants(id)
uuid	room_id	FK	Foreign Key -> rooms(id)
date	reservation_date		Date of reservation
varchar	time_slot		BREAKFAST | LUNCH | DINNER | LATE_NIGHT
integer	party_size		Number of guests
varchar	diner_name		Customer name
varchar	diner_email		Customer email
varchar	diner_phone		Customer phone
varchar	status		PENDING | CONFIRMED | CANCELLED
varchar	special_requests		Optional special requests
varchar	cancellation_reason		Reason if cancelled
varchar	cancelled_by		Who cancelled (name/email)
timestamp_tz	confirmed_at		Confirmation timestamp
timestamp_tz	cancelled_at		Cancellation timestamp
timestamp_tz	created_at		Creation timestamp
timestamp_tz	updated_at		Last update timestamp
bigint	version		Optimistic locking version

Why PostgreSQL over MongoDB?

I went with PostgreSQL because preventing double-bookings is non-negotiable. ACID guarantees + partial unique indexes enforce this at the database level - even if the application has bugs, the DB won't allow conflicts. The relational model also maps naturally to the domain (restaurants have rooms, rooms have reservations).

MongoDB would give us easier sharding (partition by restaurant_id) and schema flexibility (no migrations when adding fields like "dietary restrictions"). But here's why I'm okay giving that up: the reservation schema is stable - room, date, time_slot aren't changing. And 90% of traffic is availability checks, which scale fine with PostgreSQL read replicas + Redis caching.

Bottom line: preventing double-bookings matters more than write scalability for this use case.

Critical: Partial Unique Index

```
CREATE UNIQUE INDEX uk_room_date_slot_active
  ON reservations (room_id, reservation_date, time_slot)
  WHERE status IN ('PENDING', 'CONFIRMED');
```

Why This Works:

- Prevents double-booking (only 1 active reservation per room/date/slot)

- Allows rebooking (CANCELLED reservations excluded from constraint)
- Cannot be bypassed even with application bugs

Why Not Simple `UNIQUE(room_id, date, slot)`? Would prevent rebooking after cancellation.

Why Not `UNIQUE(room_id, date, slot, status)`? Would allow double-booking (PENDING and CONFIRMED are different values).

---

3. API Design

OpenAPI Specification: system-design/api-spec.json **Interactive Docs:** `http://localhost:8080/swagger-ui.html`

API Design Principles

The API follows standard REST conventions:

- URLs are resource-oriented (`/restaurants/{id}/rooms`, `/reservations/{id}`) not action-based
- Standard HTTP verbs (GET for reads, POST for creates, POST to `/cancel` for cancellations)
- Status codes match intent: 200/201 for success, 404 for not found, **409 for double-booking conflicts**, 503 when unavailable
- GlobalExceptionHandler ensures consistent error responses across all endpoints
- Stateless design - no server sessions, each request is self-contained
- Query params for filtering (`/restaurants?city=Seattle`, availability date ranges)
- URI versioning (`/api/v1/`) since it's simpler than header versioning for load balancer routing
- Per-restaurant rate limits (100-2000 req/min depending on tier) to prevent one busy restaurant from impacting others
- Swagger UI at `/swagger-ui.html` for interactive testing
- Standard pagination (`page`, `size` params) on list endpoints

Key Design Decisions

1. Idempotency The database constraint provides natural idempotency. If a client retries the same reservation request, the partial unique index rejects it with a 409 response that includes the existing reservation ID and a hash of the diner's email. This lets the client verify if it's their own retry (safe to ignore) vs someone else who booked the slot (actual conflict).

2. Versioning Went with URI versioning (`/api/v1/`) instead of header versioning because it's easier to route at the load balancer level and works better with HTTP caching.

3. REST over GraphQL Availability queries are the most common operation (users browsing dates before booking), and they're highly cacheable since the response structure is fixed. GraphQL would add unnecessary complexity when the query pattern is simple: "show me available slots for this room from Dec 20-27." HTTP caching gives us better performance here.

Multi-Tenancy

Data Isolation:

- All queries filtered by `restaurant_id`

- API key authorization: `if (!restaurant.getId().equals(restaurantId)) throw Forbidden`
- Future: PostgreSQL Row-Level Security (RLS) for defense-in-depth

Per-Restaurant Rate Limiting:

- STANDARD: 100 req/min, PREMIUM: 500 req/min, ENTERPRISE: 2000 req/min
- Isolates performance: One restaurant's high traffic doesn't degrade service for others

Feature Flags (LaunchDarkly):

- A/B test new features per restaurant
- Gradual rollout, premium tier features

---

4. Event-Driven Architecture

MVP: Spring Events (in-memory) **Production:** Apache Kafka

Event Types

- `ReservationCreatedEvent` → NotificationListener, AuditListener, AnalyticsListener
- `ReservationCancelledEvent` → NotificationListener, AuditListener

Consistency Model

The reservation write itself is strongly consistent - it's committed to PostgreSQL before returning a response. Everything else (email notifications, audit logs, analytics) happens async via events.

If email sending fails, that's okay - the reservation is already confirmed and the diner can always check `/diners/{email}/reservations` to see it. I prioritized getting the booking locked in over waiting for side effects.

Event Versioning

Using Avro + Schema Registry for when we need to evolve events over time. For example, if we add an "estimatedSpend" field to ReservationCreatedEvent six months from now for revenue tracking, old consumers will ignore it (forward compat) and new consumers will handle its absence in old events (backward compat). Standard migration: deploy the schema, deploy producers, deploy consumers, monitor, then backfill historical events if needed.

---

5. Scalability Plan

Current Limitations

Right now it's a single-instance monolith handling ~100 req/sec, which is fine for the MVP scope.

Horizontal Scaling Strategy

Application Tier:

- Stateless instances behind AWS ALB

- Auto-scale on: CPU > 70%, latency > 150ms, connection pool > 75%

Database Tier:

- Read replicas for availability queries (90% of traffic)
- Primary for reservation writes
- Connection pooling (HikariCP: 20 connections/instance)

Caching (Redis):

- **Restaurant/Room catalog**: 1hr TTL (rarely updated, cache key: `restaurant::{id}`, `room::{id}`)
- **Availability calendar**: 5min TTL (key: `availability::{roomId}::{startDate}::{endDate}`)
  - Stale reads acceptable: If cache shows "available" but slot booked → DB partial unique index returns 409
  - Production: Invalidate on `ReservationCreatedEvent` for freshness
- **Cache hit target**: >70% for availability queries (most users check popular dates)

Event Processing:

- Spring Events (MVP) → Kafka with partition by `reservationId` for ordering

Observability

Critical Metrics:

Metric	Threshold	Action
`reservation_creation_p95_latency`	> 200ms for 5min	Page oncall
`availability_query_p95_latency`	> 100ms for 5min	Auto-scale +2
`error_rate_reservation_create`	> 1% for 2min	Page oncall
`hikaricp_connections_active`	= 20/20 for 1min	Critical alert
`cache_hit_ratio_availability`	< 70%	Review TTL

Distributed Tracing (X-Ray): Helps identify bottlenecks. For example, if 80% of request time is in `postgres_commit`, that's lock contention from concurrent bookings - which is expected and actually proves the locking is working.

Deployment

Blue-Green is my preference for changes touching booking logic - instant rollback if we detect bugs, and the 2x infra cost during the ~10min deployment window is worth it.

Canary deployments work fine for non-critical stuff like UI updates or analytics changes where a slower rollout is acceptable.

Cost (AWS us-east-1)

Phase	Infrastructure	Monthly Cost	Capacity

Phase	Infrastructure	Monthly Cost	Capacity
Phase 1 (MVP)	1 x t3.medium EC2, db.t3.small RDS	$75	~100 req/sec
Phase 2 (Production)	3 x m5.large, db.r5.large Multi-AZ, Read Replica, Redis r5.large, ALB	$956	~1000 req/sec
Phase 3 (Enterprise)	Multi-region, 6 x m5.xlarge, db.r5.2xlarge, 3 Read Replicas, Redis Cluster, MSK Kafka	$4,664	10K+ req/sec

Cost optimization: Reserved Instances save 40%: Phase 2 → $575/month.

Failure Scenarios

Database Primary Down: RDS Multi-AZ failover (~60s), return 503 with retry-after. **Connection Pool Exhausted:** Query timeout 10s, auto-scale instances. **Read Replica Lag > 5s:** Route reads to primary, stale availability acceptable (DB constraint catches conflicts). **Circuit Breaker:** Non-critical services (notification) fail fast, don't block reservation creation.

Non-Functional Requirements

Requirement	Target	Rationale
Latency	p95 < 200ms (create), p95 < 100ms (availability)	Users abandon if booking takes > 3 seconds
Throughput	1000 req/sec (production)	Peak: Friday 6-8pm, 50 restaurants × 20 concurrent users
Availability	99.9% (8.7hr downtime/year)	Acceptable for non-critical business (not payments/healthcare)
Consistency	Zero double-bookings	Revenue loss + customer trust damage
Durability	RDS Multi-AZ (RPO < 5min)	Can tolerate losing last 5min of reservations in disaster

CAP Theorem Decision

I chose **CP** (Consistency + Partition tolerance) over **AP** (Availability + Partition tolerance).

Here's why: double-bookings cause real customer pain - complaints, refunds, reputation damage. During a network partition, I'd rather return a 503 "temporarily unavailable, retry in 60s" than risk accepting conflicting reservations.

The AP alternative (eventual consistency with conflict resolution like "first timestamp wins") would allow double-bookings during the partition window. That's a non-starter for a reservation system. Better to be temporarily unavailable than to accept a booking we can't honor.

Setup: README.md