How Airbnb Works

> ***Target Audience**: Fullstack engineers (TS/Go, PostgreSQL, Redis)*
> ***Focus**: Booking system, search/availability, payments, trust/safety*
> ***Scale Context**: ~7M listings, ~150M users, ~400M bookings/year*

---

1. Problem Definition (What This System Must Achieve)

Airbnb is a two-sided marketplace connecting property hosts with travelers seeking accommodations.

Core functional requirements:

- **Host side**: List properties (photos, amenities, pricing, availability calendar)
- **Guest side**: Search properties (location, dates, filters), book stays, pay securely
- **Messaging**: Host-guest communication
- **Reviews**: Bidirectional ratings after checkout
- **Pricing**: Dynamic pricing, currency conversion, taxes, fees
- **Calendar management**: Availability, booking conflicts, cancellations

Non-functional requirements:

- **Latency**: Search results < 500ms, booking confirmation < 1s
- **Scale**: 7M listings, 150M users, 1M+ searches/minute peak
- **Consistency**: No double-booking (strong consistency for reservations)
- **Availability**: 99.95%+ uptime (bookings are revenue-critical)
- **Accuracy**: Search results must respect availability calendars
- **Transaction safety**: Payment holds, refunds, dispute resolution
- **Compliance**: Tax calculations per jurisdiction, legal requirements

What makes this hard:

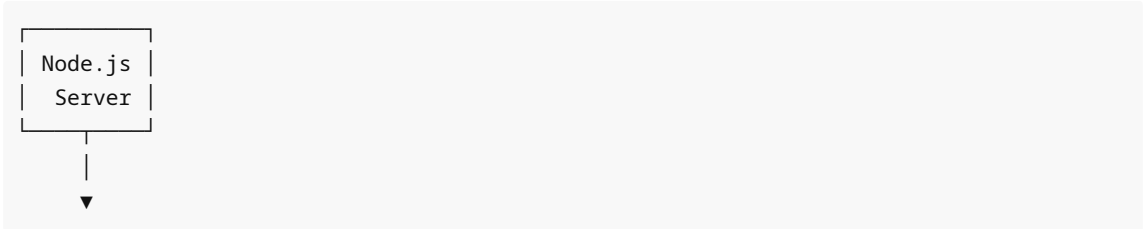
- **Inventory management**: Prevent double-booking across timezones
- **Complex search**: Geo-spatial + date-range + price + amenities + instant book
- **Calendar math**: Handle check-in/out dates, minimum stays, blocked dates
- **Payment orchestration**: Hold funds, payout to host, handle cancellations
- **Trust & safety**: Verify hosts/guests, detect fraud, enforce policies
- **Currency handling**: 180+ currencies, exchange rates, price display
- **Peak load**: Holiday weekends = 10x normal search traffic

Unlike hotel booking (inventory managed by hotel chain), Airbnb has **distributed inventory** controlled by millions of individual hosts — no central source of truth for availability.

---

2. Naive Design (And Why It Fails)

The Simple Version



```
┌──────────┐
│Postgres  │
│  (all)   │
└──────────┘
```

Schema:

```
listings: { id, host_id, title, address, price_per_night, lat, lng, ... }
bookings: { id, listing_id, guest_id, check_in, check_out, status }
calendars: { listing_id, date, available }

// Search query
SELECT * FROM listings
WHERE ST_Distance(location, user_location) < 50km
AND price_per_night BETWEEN $100 AND $200
```

Booking flow (naive):

```
async function bookListing(listingId: string, checkIn: Date, checkOut: Date) {
  // Check availability
  const blocked = await db.query(
    'SELECT * FROM calendars WHERE listing_id = $1 AND date >= $2 AND date < $3 AND
available = false',
    [listingId, checkIn, checkOut]
  );

  if (blocked.length > 0) {
    throw new Error('Dates not available');
  }

  // Create booking
  await db.query(
    'INSERT INTO bookings (listing_id, guest_id, check_in, check_out, status) VALUES
($1, $2, $3, $4, $5)',
    [listingId, guestId, checkIn, checkOut, 'confirmed']
  );

  // Block calendar
  await db.query(
    'UPDATE calendars SET available = false WHERE listing_id = $1 AND date >= $2 AND
date < $3',
    [listingId, checkIn, checkOut]
  );
}
```

Why This Breaks

1. Double-booking race condition:

```
Time  Guest A                Guest B
t0    Check availability ✓   Check availability ✓
```

```
t1    Create booking →       Create booking →
t2    Both bookings succeed (DISASTER)
```

- Two guests book same dates simultaneously
- Both see availability, both create bookings
- Host has overlapping reservations
- **Result**: Trust erosion, angry customers, refunds

2. Search query performance collapses:

- Geo-spatial query (`ST_Distance`) is slow without proper indexing
- Date-range availability check requires scanning `calendars` table
- 7M listings × 365 days = 2.5B calendar rows
- Query takes 10+ seconds
- **Result**: Search unusable during peak hours

3. Calendar table explodes:

- Each listing has 365 days/year × 2 years future = 730 rows
- 7M listings × 730 = 5.1 billion rows
- Inserts/updates are slow
- Storage cost: $50K+/month
- **Result**: Database becomes write-bottleneck

4. No pricing flexibility:

- Fixed `price_per_night` in listings table
- Can't handle weekend premiums, seasonal pricing
- Can't handle length-of-stay discounts
- **Result**: Hosts leave platform for competitors

5. Payment holds don't exist:

- Naive design charges guest immediately
- What if booking canceled?
- What if dispute?
- No escrow → fraud risk
- **Result**: Chargebacks, financial loss

6. Search filter explosion:

- User filters: "pet-friendly, wifi, pool, parking, instant book, superhosts only"
- Query becomes: `WHERE ... AND pet_friendly = true AND wifi = true AND pool = true ...`
- No indexes can satisfy all combinations
- **Result**: Full table scans, timeouts

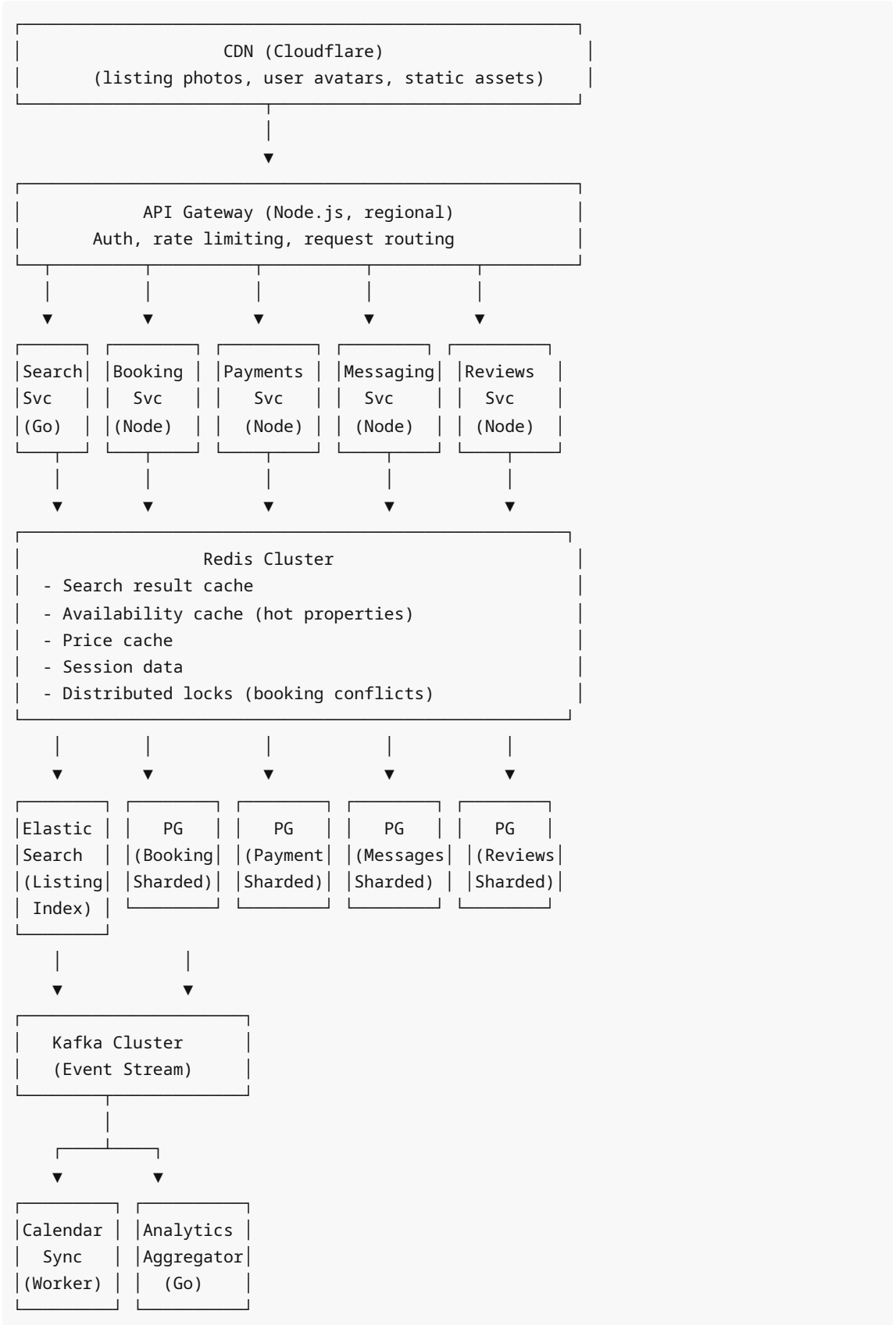
7. Cross-timezone booking bugs:

- Guest in Tokyo books for "March 15 check-in"
- Host in New York sees different date due to timezone conversion
- Check-in time confusion
- **Result**: Guest arrives, host hasn't prepared, bad experience

---

3. High-Level Architecture

Component Overview



Service Boundaries

Search Service (Go):

- Query Elasticsearch for listings
- Apply filters (price range, amenities, instant book)
- Geo-spatial search
- Merge availability data from cache/DB
- Go chosen for: High concurrency, JSON parsing efficiency

Booking Service (Node.js):

- Create reservations (with distributed locking)
- Calendar availability checks
- Booking modifications and cancellations
- Manage booking state machine

Payments Service (Node.js):

- Stripe integration
- Payment holds (authorize, not capture)
- Payout orchestration to hosts
- Refund processing
- Currency conversion

Messaging Service (Node.js):

- Host-guest messaging
- Real-time delivery (WebSockets)
- Message persistence
- Notification triggers

Reviews Service (Node.js):

- Bidirectional reviews (guest → host, host → guest)
- Rating aggregation
- Review visibility rules (both parties must review)

---

4. Core Data Model

PostgreSQL Schema (Sharded)

Listings Database (sharded by listing_id):

```
interface Listing {
  id: string; // UUID
  host_id: string;

  // Property details
  title: string;
  description: string;
  property_type: 'apartment' | 'house' | 'villa' | 'room';

  // Location
```

```
  address: string;
  city: string;
  state: string;
  country: string;
  postal_code: string;
  lat: number;
  lng: number;

  // Capacity
  max_guests: number;
  bedrooms: number;
  beds: number;
  bathrooms: number;

  // Amenities (JSON array)
  amenities: string[]; // ['wifi', 'kitchen', 'parking', 'pool', 'pet_friendly']

  // Pricing (base price, overrides handled separately)
  base_price_per_night: number;
  currency: string;
  cleaning_fee: number;

  // Booking rules
  min_nights: number;
  max_nights: number;
  instant_book: boolean;
  cancellation_policy: 'flexible' | 'moderate' | 'strict';

  // Status
  status: 'draft' | 'active' | 'inactive' | 'suspended';

  created_at: timestamp;
  updated_at: timestamp;
}
// Index: (host_id, status)
// Index: (city, status) for analytics
// PostGIS index: (location) for geo queries

interface ListingPhoto {
  id: string;
  listing_id: string;
  url: string; // CDN URL
  display_order: number;
  caption: string | null;
}
// Index: (listing_id, display_order)
```

Bookings Database (sharded by booking_id):

```
interface Booking {
  id: string; // UUID
```

```
  listing_id: string;
  host_id: string;
  guest_id: string;

  // Dates (stored in UTC)
  check_in_date: date; // YYYY-MM-DD
  check_out_date: date;
  check_in_time: string; // "15:00" (listing's local time)
  check_out_time: string; // "11:00"

  // Guests
  num_guests: number;
  num_adults: number;
  num_children: number;
  num_infants: number;

  // Pricing breakdown
  nights: number;
  price_per_night: number;
  total_accommodation: number;
  cleaning_fee: number;
  service_fee: number;
  taxes: number;
  total_amount: number;
  currency: string;

  // Payment
  payment_intent_id: string; // Stripe payment intent
  payment_status: 'pending' | 'authorized' | 'captured' | 'refunded';
  payout_status: 'pending' | 'scheduled' | 'paid';

  // Status
  status: 'pending' | 'confirmed' | 'checked_in' | 'checked_out' | 'cancelled';
  cancelled_by: 'guest' | 'host' | 'system' | null;
  cancelled_at: timestamp | null;
  cancellation_reason: string | null;

  created_at: timestamp;
  updated_at: timestamp;
}
// Index: (listing_id, check_in_date) for availability queries
// Index: (guest_id, status, check_in_date) for guest bookings
// Index: (host_id, status, check_in_date) for host bookings
// Index: (status, check_in_date) for upcoming bookings

// Separate table for calendar management
interface ListingCalendar {
  listing_id: string;
  date: date; // Single date
  status: 'available' | 'blocked' | 'booked';
  booking_id: string | null; // If booked
  price_override: number | null; // Host can set custom price for specific dates
```

```
  notes: string | null; // Host notes (e.g., "maintenance")
}
// Composite PK: (listing_id, date)
// Index: (listing_id, date, status)
```

Users Database (sharded by user_id):

```
interface User {
  id: string; // UUID
  email: string; // unique
  password_hash: string;

  // Profile
  first_name: string;
  last_name: string;
  phone: string;
  birth_date: date;
  profile_photo_url: string;

  // Verification
  email_verified: boolean;
  phone_verified: boolean;
  identity_verified: boolean; // Government ID

  // Roles
  is_host: boolean;
  is_superhost: boolean;

  // Trust score (internal, not shown to users)
  trust_score: number; // 0-100

  // Preferences
  preferred_currency: string;
  preferred_language: string;

  created_at: timestamp;
}
// Index: (email), (phone)

interface UserVerification {
  user_id: string;
  type: 'email' | 'phone' | 'identity' | 'payment';
  verified: boolean;
  verified_at: timestamp | null;
  verification_data: jsonb; // Encrypted sensitive data
}
// Composite PK: (user_id, type)
```

Reviews Database (sharded by booking_id):

```
interface Review {
  id: string;
  booking_id: string;
  reviewer_id: string; // Guest or host who wrote this
  reviewee_id: string; // Host or guest being reviewed

  // Ratings (1-5 stars)
  overall_rating: number;
  cleanliness_rating: number | null; // Guest reviews host
  communication_rating: number | null;
  accuracy_rating: number | null;
  location_rating: number | null;
  value_rating: number | null;

  // Review text
  public_review: string; // Visible to everyone
  private_feedback: string | null; // Only to Airbnb

  // Visibility (both parties must review before revealing)
  visible: boolean;

  created_at: timestamp;
  published_at: timestamp | null;
}
// Index: (listing_id, visible, created_at) for listing reviews
// Index: (reviewee_id, visible) for user reviews
```

Elasticsearch Index (Listings)

Structure:

```
{
  "listing_id": "uuid",
  "host_id": "uuid",
  "title": "Cozy apartment in downtown",
  "city": "San Francisco",
  "state": "California",
  "country": "United States",
  "location": {
    "lat": 37.7749,
    "lon": -122.4194
  },
  "property_type": "apartment",
  "max_guests": 4,
  "bedrooms": 2,
  "bathrooms": 1.5,
  "amenities": ["wifi", "kitchen", "parking", "ac"],
  "base_price_per_night": 150,
  "currency": "USD",
  "instant_book": true,
  "min_nights": 2,
```

```
  "avg_rating": 4.8,
  "num_reviews": 47,
  "is_superhost": true,
  "photos": ["url1", "url2"],
  "status": "active"
}
```

Why Elasticsearch:

- Full-text search on title/description
- Geo-spatial queries (distance from user location)
- Complex filter combinations (amenities, price range, instant book)
- Fast aggregations (price distribution, popular amenities)
- Relevance scoring (boost superhosts, highly-rated properties)

Redis Cache Structure

```
// Listing metadata cache
`listing:${listingId}` → JSON { id, title, hostId, basePrice, amenities, ... }

// Availability cache (hot properties)
`availability:${listingId}:${month}` → Bitmap (1 = available, 0 = blocked)
  // Example: March 2026 = 31 bits

// Search results cache (geo + filters)
`search:${lat}:${lng}:${filters_hash}` → JSON array of listing IDs

// Price cache (with overrides)
`price:${listingId}:${date}` → number

// Distributed lock for booking
`lock:booking:${listingId}:${checkIn}:${checkOut}` → "locked" (TTL: 30 seconds)

// User session
`session:${token}` → JSON { userId, email, ... }
```

Consistency Guarantees

Strongly consistent:

- Booking creation (no double-booking)
- Payment authorization
- Calendar blocking

Eventually consistent:

- Search index (listing updates take 1-2 seconds to appear)
- Review ratings aggregation
- Host/guest profile stats

Real-time:

- Availability queries (must reflect latest bookings)

- Price calculations (taxes, fees)

---

5. Core Workflows

Workflow 1: Guest Searches for Listings

Step-by-step:

1. **Client** sends search request

```
GET /api/v1/search?location=San Francisco&checkIn=2026-07-01&checkOut=2026-07-05&guests=2&minPrice=100&maxPrice=300&amenities=wifi,parking&instantBook=true
```

2. **Search Service** (Go) processes request:

```
func (s *SearchService) SearchListings(req SearchRequest) ([]Listing, error) {
    // 1. Geocode location (if not lat/lng provided)
    coords, err := s.geocode(req.Location)
    if err != nil {
        return nil, err
    }

    // 2. Check cache first
    cacheKey := s.generateCacheKey(req)
    cached, _ := s.redis.Get(ctx, cacheKey).Result()
    if cached != "" {
        var results []string
        json.Unmarshal([]byte(cached), &results)
        return s.hydrateListings(results), nil
    }

    // 3. Query Elasticsearch
    esQuery := s.buildElasticsearchQuery(coords, req)
    results, err := s.es.Search(esQuery)
    if err != nil {
        return nil, err
    }

    // 4. Filter by availability (critical step)
    availableListings := s.filterByAvailability(results, req.CheckIn,
req.CheckOut)

    // 5. Cache results (5 minutes)
    s.redis.Set(ctx, cacheKey, json.Marshal(availableListings), 5*time.Minute)

    return availableListings, nil
}

func (s *SearchService) buildElasticsearchQuery(coords Coords, req
SearchRequest) map[string]interface{} {
    query := map[string]interface{}{
```

```
        "bool": map[string]interface{}{
            "must": []interface{}{
                // Geo-distance filter
                map[string]interface{}{
                    "geo_distance": map[string]interface{}{
                        "distance": "50km",
                        "location": map[string]interface{}{
                            "lat": coords.Lat,
                            "lon": coords.Lng,
                        },
                    },
                },
                // Status filter
                map[string]interface{}{
                    "term": map[string]interface{}{"status": "active"},
                },
                // Guest capacity
                map[string]interface{}{
                    "range": map[string]interface{}{
                        "max_guests": map[string]interface{}{"gte":
req.Guests},
                    },
                },
            },
            "filter": []interface{}{},
        },
    }

    // Price range
    if req.MinPrice > 0 || req.MaxPrice > 0 {
        priceFilter := map[string]interface{}{
            "range": map[string]interface{}{
                "base_price_per_night": map[string]interface{}{},
            },
        }
        if req.MinPrice > 0 {
            priceFilter["range"].(map[string]interface{})
["base_price_per_night"].(map[string]interface{})["gte"] = req.MinPrice
        }
        if req.MaxPrice > 0 {
            priceFilter["range"].(map[string]interface{})
["base_price_per_night"].(map[string]interface{})["lte"] = req.MaxPrice
        }
        query["bool"].(map[string]interface{})["filter"] = append(
            query["bool"].(map[string]interface{})["filter"].([]interface{}),
            priceFilter,
        )
    }

    // Amenities filter (all must be present)
    if len(req.Amenities) > 0 {
        for _, amenity := range req.Amenities {
```

```
            query["bool"].(map[string]interface{})["filter"] = append(
                query["bool"].(map[string]interface{})["filter"].
([]interface{}),
                map[string]interface{}{
                    "term": map[string]interface{}{"amenities": amenity},
                },
            )
        }
    }

    // Instant book filter
    if req.InstantBook {
        query["bool"].(map[string]interface{})["filter"] = append(
            query["bool"].(map[string]interface{})["filter"].([]interface{}),
            map[string]interface{}{
                "term": map[string]interface{}{"instant_book": true},
            },
        )
    }

    return query
}

func (s *SearchService) filterByAvailability(listings []Listing, checkIn,
checkOut time.Time) []Listing {
    var available []Listing

    // Batch check availability for all listings
    for _, listing := range listings {
        if s.isAvailable(listing.Id, checkIn, checkOut) {
            available = append(available, listing)
        }
    }

    return available
}

func (s *SearchService) isAvailable(listingId string, checkIn, checkOut
time.Time) bool {
    // Check Redis bitmap cache first (fast path)
    month := checkIn.Format("2006-01")
    cacheKey := fmt.Sprintf("availability:%s:%s", listingId, month)

    // Get bitmap for the month
    bitmap, err := s.redis.Get(ctx, cacheKey).Result()
    if err == nil {
        // Check each day in range
        for d := checkIn; d.Before(checkOut); d = d.AddDate(0, 0, 1) {
            dayOfMonth := d.Day() - 1 // 0-indexed
            if !isBitSet(bitmap, dayOfMonth) {
                return false // Day not available
            }
```

```
        }
        return true
    }

    // Cache miss: Query database
    blocked := s.db.Query(
        `SELECT COUNT(*) FROM listing_calendar
         WHERE listing_id = $1
         AND date >= $2 AND date < $3
         AND status != 'available'`,
        listingId, checkIn, checkOut,
    )

    return blocked == 0
}
```

3. **Client** receives results

```
Response:
{
  "results": [
    {
      "id": "listing-123",
      "title": "Cozy 2BR apartment",
      "city": "San Francisco",
      "pricePerNight": 180,
      "totalPrice": 720, // 4 nights
      "photos": ["url1", "url2"],
      "rating": 4.8,
      "numReviews": 47,
      "isSuperhost": true,
      "instantBook": true
    },
    // ... more results
  ],
  "total": 234,
  "page": 1
}
```

Performance targets:

- Elasticsearch query: 50-100ms
- Availability filtering: 50-100ms (cached) or 200-300ms (DB)
- Total: 200-500ms

Workflow 2: Guest Books a Listing

Challenge: Prevent double-booking with strong consistency

Step-by-step:

1. **Client** sends booking request

```
POST /api/v1/bookings
{
  "listingId": "listing-123",
  "checkIn": "2026-07-01",
  "checkOut": "2026-07-05",
  "guests": 2,
  "paymentMethodId": "pm_abc123"
}
```

2. **Booking Service** (Node.js) - **Distributed Lock Pattern**:

```
async function createBooking(req: BookingRequest): Promise<Booking> {
  const lockKey =
`lock:booking:${req.listingId}:${req.checkIn}:${req.checkOut}`;

  // 1. Acquire distributed lock (prevent concurrent bookings)
  const lockAcquired = await redis.set(lockKey, 'locked', {
    NX: true, // Only set if doesn't exist
    EX: 30    // 30 second expiry
  });

  if (!lockAcquired) {
    throw new Error('Someone else is booking this property. Please try
again.');
  }

  try {
    // 2. Check availability (double-check after lock)
    const available = await checkAvailability(req.listingId, req.checkIn,
req.checkOut);
    if (!available) {
      throw new Error('Dates no longer available');
    }

    // 3. Calculate pricing
    const pricing = await calculatePricing(req);

    // 4. Authorize payment (hold funds, don't capture yet)
    const paymentIntent = await stripe.paymentIntents.create({
      amount: pricing.totalAmount * 100, // cents
      currency: pricing.currency,
      payment_method: req.paymentMethodId,
      capture_method: 'manual', // Authorize only
      metadata: {
        listingId: req.listingId,
        guestId: req.guestId
      }
    });

    // 5. Create booking in database (transaction)
    const booking = await db.transaction(async (tx) => {
```

```
      // Insert booking
      const bookingResult = await tx.query(
        `INSERT INTO bookings (
          id, listing_id, host_id, guest_id,
          check_in_date, check_out_date,
          num_guests, total_amount, currency,
          payment_intent_id, payment_status, status,
          created_at
        ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, NOW())
        RETURNING *`,
        [
          generateUUID(), req.listingId, listing.hostId, req.guestId,
          req.checkIn, req.checkOut,
          req.guests, pricing.totalAmount, pricing.currency,
          paymentIntent.id, 'authorized', 'confirmed',
        ]
      );

      const booking = bookingResult.rows[0];

      // Block calendar dates
      const dates = getDateRange(req.checkIn, req.checkOut);
      for (const date of dates) {
        await tx.query(
          `INSERT INTO listing_calendar (listing_id, date, status, booking_id)
           VALUES ($1, $2, 'booked', $3)
           ON CONFLICT (listing_id, date)
           DO UPDATE SET status = 'booked', booking_id = $3`,
          [req.listingId, date, booking.id]
        );
      }

      return booking;
    });

    // 6. Publish event (async)
    await kafka.publish('booking.created', {
      bookingId: booking.id,
      listingId: req.listingId,
      hostId: listing.hostId,
      guestId: req.guestId,
      checkIn: req.checkIn,
      checkOut: req.checkOut
    });

    // 7. Invalidate availability cache
    await invalidateAvailabilityCache(req.listingId, req.checkIn,
req.checkOut);

    return booking;
  } finally {
    // 8. Release lock
```

```
    await redis.del(lockKey);
  }
}

async function checkAvailability(listingId: string, checkIn: Date, checkOut:
Date): Promise<boolean> {
  // Check for any conflicting bookings
  const conflicts = await db.query(
    `SELECT id FROM bookings
     WHERE listing_id = $1
     AND status NOT IN ('cancelled')
     AND (
       (check_in_date < $3 AND check_out_date > $2) OR  -- Overlap
       (check_in_date >= $2 AND check_in_date < $3)      -- Check-in within
range
     )
     FOR UPDATE`,  // Lock rows to prevent concurrent modifications
    [listingId, checkIn, checkOut]
  );

  return conflicts.rows.length === 0;
}

async function calculatePricing(req: BookingRequest):
Promise<PricingBreakdown> {
  const listing = await getListing(req.listingId);

  // Calculate nights
  const nights = daysBetween(req.checkIn, req.checkOut);

  // Get price per night (may vary by date)
  let accommodationTotal = 0;
  for (let d = new Date(req.checkIn); d < req.checkOut; d.setDate(d.getDate()
+ 1)) {
    const priceForDate = await getPriceForDate(req.listingId, d);
    accommodationTotal += priceForDate;
  }

  // Add fees
  const cleaningFee = listing.cleaningFee;
  const serviceFee = accommodationTotal * 0.14; // 14% Airbnb fee

  // Calculate taxes (varies by jurisdiction)
  const taxes = await calculateTaxes(listing.country, listing.state,
accommodationTotal);

  const total = accommodationTotal + cleaningFee + serviceFee + taxes;

  return {
    nights,
    pricePerNight: accommodationTotal / nights,
    accommodationTotal,
```

```
    cleaningFee,
    serviceFee,
    taxes,
    totalAmount: total,
    currency: listing.currency
  };
}
```

3. **Background worker** notifies host:

```
// Kafka consumer
async function handleBookingCreated(event: BookingCreatedEvent) {
  // Send email to host
  await emailService.send(event.hostId, {
    template: 'new_booking',
    data: {
      guestName: guest.firstName,
      checkIn: event.checkIn,
      checkOut: event.checkOut,
      totalPayout: calculateHostPayout(event.totalAmount)
    }
  });

  // Send push notification
  await pushService.send(event.hostId, {
    title: 'New booking!',
    body: `${guest.firstName} booked your property for ${event.checkIn}`
  });
}
```

4. **Client** receives confirmation

```
Response:
{
  "bookingId": "booking-456",
  "status": "confirmed",
  "confirmationCode": "HMBNXK",
  "listing": {...},
  "checkIn": "2026-07-01",
  "checkOut": "2026-07-05",
  "pricing": {
    "accommodation": 720,
    "cleaningFee": 50,
    "serviceFee": 100,
    "taxes": 75,
    "total": 945,
    "currency": "USD"
  },
  "paymentStatus": "authorized"
}
```

Failure handling:

- If payment authorization fails → no booking created, return error
- If database transaction fails → payment authorization is voided
- If Kafka publish fails → booking still created (notification can be retried)

Why this works:

- Distributed lock prevents concurrent bookings
- Transaction ensures atomicity (booking + calendar block)
- `FOR UPDATE` lock in availability check prevents phantom reads
- Payment authorization (not capture) allows cancellation

Workflow 3: Host Updates Availability Calendar

Step-by-step:

1. **Host** blocks dates (e.g., personal use)

```
POST /api/v1/listings/listing-123/calendar/block
{
  "startDate": "2026-08-15",
  "endDate": "2026-08-20",
  "notes": "Family vacation"
}
```

2. **Booking Service** updates calendar:

```
async function blockDates(listingId: string, startDate: Date, endDate: Date,
notes: string) {
  // Check for existing bookings
  const existingBookings = await db.query(
    `SELECT id FROM bookings
     WHERE listing_id = $1
     AND status NOT IN ('cancelled')
     AND check_in_date < $3 AND check_out_date > $2`,
    [listingId, startDate, endDate]
  );

  if (existingBookings.rows.length > 0) {
    throw new Error('Cannot block dates with existing bookings');
  }

  // Block in calendar
  const dates = getDateRange(startDate, endDate);
  for (const date of dates) {
    await db.query(
      `INSERT INTO listing_calendar (listing_id, date, status, notes)
       VALUES ($1, $2, 'blocked', $3)
       ON CONFLICT (listing_id, date)
       DO UPDATE SET status = 'blocked', notes = $3`,
      [listingId, date, notes]
```

```
    );
  }

  // Invalidate cache
  await invalidateAvailabilityCache(listingId, startDate, endDate);

  // Update Elasticsearch (async)
  // Doesn't need immediate consistency
  await kafka.publish('listing.calendar.updated', { listingId, startDate,
endDate });
}
```

---

6. API Design

REST Endpoints

Authentication:

```
POST /api/v1/auth/signup
POST /api/v1/auth/login
POST /api/v1/auth/logout
POST /api/v1/auth/verify-email
```

Search:

```
GET /api/v1/search?location=&checkIn=&checkOut=&guests=&filters=
GET /api/v1/listings/:id
GET /api/v1/listings/:id/availability?month=2026-07
GET /api/v1/listings/:id/reviews?limit=20
```

Bookings:

```
POST   /api/v1/bookings
GET    /api/v1/bookings/:id
PUT    /api/v1/bookings/:id/cancel
GET    /api/v1/users/:userId/bookings (guest trips)
GET    /api/v1/hosts/:userId/bookings (host reservations)
```

Listings (Host):

```
POST   /api/v1/listings
PUT    /api/v1/listings/:id
DELETE /api/v1/listings/:id
POST   /api/v1/listings/:id/photos
POST   /api/v1/listings/:id/calendar/block
POST   /api/v1/listings/:id/calendar/set-price
```

Messaging:

```
GET    /api/v1/conversations
POST   /api/v1/conversations/:id/messages
GET    /api/v1/conversations/:id/messages?limit=50
WebSocket: wss://messaging.airbnb.com/ws
```

Reviews:

```
POST /api/v1/bookings/:id/reviews
GET  /api/v1/listings/:id/reviews?limit=20
GET  /api/v1/users/:id/reviews
```

Idempotency

Booking creation:

```
POST /api/v1/bookings
Headers: {
  "Idempotency-Key": "uuid-generated-by-client"
}

// Server-side
async function createBooking(req: BookingRequest, idempotencyKey: string) {
  // Check if already processed
  const existing = await redis.get(`idempotency:${idempotencyKey}`);
  if (existing) {
    return JSON.parse(existing); // Return cached result
  }

  // Process booking
  const booking = await processBooking(req);

  // Cache result for 24 hours
  await redis.setex(`idempotency:${idempotencyKey}`, 86400,
JSON.stringify(booking));

  return booking;
}
```

---

7. Calendar & Availability Algorithm

Bitmap Encoding for Performance

Problem: Checking 30 days of availability = 30 DB queries (slow)

Solution: Redis bitmap (1 bit per day)

```
// Encode availability for a month
async function buildAvailabilityBitmap(listingId: string, year: number, month:
number) {
```

```
  const daysInMonth = new Date(year, month + 1, 0).getDate();
  const bitmap = new Array(daysInMonth).fill(1); // Default available

  // Query blocked/booked dates for this month
  const blockedDates = await db.query(
    `SELECT date FROM listing_calendar
     WHERE listing_id = $1
     AND date >= $2 AND date < $3
     AND status != 'available'`,
    [listingId, `${year}-${month}-01`, `${year}-${month + 1}-01`]
  );

  for (const row of blockedDates) {
    const day = new Date(row.date).getDate() - 1; // 0-indexed
    bitmap[day] = 0; // Mark unavailable
  }

  // Convert to string
  const bitmapStr = bitmap.join('');

  // Store in Redis
  const key = `availability:${listingId}:${year}-${month}`;
  await redis.set(key, bitmapStr, { EX: 3600 }); // 1 hour TTL

  return bitmapStr;
}

// Check if date range is available
async function isAvailable(listingId: string, checkIn: Date, checkOut: Date):
Promise<boolean> {
  const months = getMonthsInRange(checkIn, checkOut);

  for (const month of months) {
    const key = `availability:${listingId}:${month}`;
    let bitmap = await redis.get(key);

    if (!bitmap) {
      // Cache miss: build it
      bitmap = await buildAvailabilityBitmap(listingId, month.year, month.month);
    }

    // Check each day in this month
    for (let d = new Date(Math.max(checkIn, month.start)); d < Math.min(checkOut,
month.end); d.setDate(d.getDate() + 1)) {
      const dayOfMonth = d.getDate() - 1;
      if (bitmap[dayOfMonth] === '0') {
        return false; // Day not available
      }
    }
  }
```

```
  return true;
}
```

Performance:

- Without bitmap: 30 days × 1ms DB query = 30ms
- With bitmap: 1 Redis lookup = 1ms (30x faster)

---

8. Consistency, Ordering & Concurrency

Double-Booking Prevention

The Critical Guarantee: No two guests can book overlapping dates for the same property.

Approaches Considered:

1. Client-side validation only (WRONG):

```
// Check availability, then book
if (await isAvailable()) {
  await book(); // Race condition!
}
```

- Two clients check simultaneously, both see "available"
- Both proceed to book → double-booking

2. Database unique constraint (INSUFFICIENT):

```
CREATE UNIQUE INDEX ON bookings (listing_id, check_in_date);
```

- Only prevents bookings starting on same date
- Doesn't prevent overlap (check-in during another booking)

3. Row-level locking with `FOR UPDATE` (BETTER):

```
async function createBooking(req: BookingRequest) {
  await db.transaction(async (tx) => {
    // Lock all rows that could conflict
    const conflicts = await tx.query(
      `SELECT id FROM bookings
       WHERE listing_id = $1
       AND status NOT IN ('cancelled')
       AND (check_in_date < $3 AND check_out_date > $2)
       FOR UPDATE`, // Acquire row locks
      [req.listingId, req.checkIn, req.checkOut]
    );

    if (conflicts.rows.length > 0) {
      throw new Error('Dates not available');
    }

    // Create booking
```

```
    await tx.query('INSERT INTO bookings ...');
  });
}
```

- Locks existing bookings that overlap
- Prevents concurrent modifications
- **Problem**: Doesn't lock "gaps" (no rows to lock if no existing booking)

4. Distributed lock + row lock (PRODUCTION SOLUTION):

```
async function createBooking(req: BookingRequest): Promise<Booking> {
  const lockKey = `lock:booking:${req.listingId}:${req.checkIn}:${req.checkOut}`;

  // Acquire distributed lock
  const locked = await acquireLock(lockKey, 30_000); // 30 second timeout
  if (!locked) {
    throw new Error('Another booking in progress. Try again.');
  }

  try {
    return await db.transaction(async (tx) => {
      // Check availability with row locks
      const conflicts = await tx.query(
        `SELECT id FROM bookings
         WHERE listing_id = $1
         AND status NOT IN ('cancelled')
         AND (check_in_date < $3 AND check_out_date > $2)
         FOR UPDATE`,
        [req.listingId, req.checkIn, req.checkOut]
      );

      if (conflicts.rows.length > 0) {
        throw new Error('Dates not available');
      }

      // Also check calendar blocks
      const calendarBlocks = await tx.query(
        `SELECT date FROM listing_calendar
         WHERE listing_id = $1
         AND date >= $2 AND date < $3
         AND status = 'blocked'
         FOR UPDATE`,
        [req.listingId, req.checkIn, req.checkOut]
      );

      if (calendarBlocks.rows.length > 0) {
        throw new Error('Some dates are blocked by host');
      }

      // Create booking (atomic with calendar update)
      const booking = await tx.query('INSERT INTO bookings ...');
```

```
      // Block calendar
      const dates = getDateRange(req.checkIn, req.checkOut);
      for (const date of dates) {
        await tx.query(
          `INSERT INTO listing_calendar (listing_id, date, status, booking_id)
           VALUES ($1, $2, 'booked', $3)
           ON CONFLICT (listing_id, date)
           DO UPDATE SET status = 'booked', booking_id = $3
           WHERE listing_calendar.status != 'booked'`, // Don't overwrite existing
bookings
          [req.listingId, date, booking.id]
        );
      }

      return booking;
    });
  } finally {
    await releaseLock(lockKey);
  }
}

// Distributed lock implementation (Redis)
async function acquireLock(key: string, ttlMs: number): Promise<boolean> {
  const token = generateUUID(); // Unique token for this lock holder

  const acquired = await redis.set(key, token, {
    NX: true,           // Only set if doesn't exist
    PX: ttlMs           // TTL in milliseconds
  });

  if (!acquired) {
    return false;
  }

  // Store token for later release
  lockTokens.set(key, token);
  return true;
}

async function releaseLock(key: string): Promise<void> {
  const token = lockTokens.get(key);
  if (!token) return;

  // Only delete if we still own the lock (prevent deleting someone else's lock)
  const script = `
    if redis.call("get", KEYS[1]) == ARGV[1] then
      return redis.call("del", KEYS[1])
    else
      return 0
    end
  `;
```

```
  await redis.eval(script, 1, key, token);
  lockTokens.delete(key);
}
```

Why this works:

- Distributed lock prevents concurrent booking attempts on same dates
- Row locks prevent modifications to existing bookings
- Calendar updates are atomic with booking creation
- Lock timeout (30s) prevents deadlocks if server crashes

Calendar Updates (Host Side)

Scenario: Host blocks dates while guest is searching.

Timeline:

```
t0: Guest searches, sees July 1-5 available
t1: Host blocks July 3-4
t2: Guest clicks "Book" for July 1-5
```

Solution: Re-check availability after acquiring lock (defensive programming)

```
async function createBooking(req: BookingRequest) {
  // ... acquire lock ...

  // Re-check availability (after lock)
  const available = await checkAvailabilityWithLocks(req);
  if (!available) {
    throw new Error('Dates no longer available. Please search again.');
  }

  // ... proceed with booking ...
}
```

Trade-off:

- Guest sees "available" in search results (stale data)
- Booking fails at checkout if host blocked in meantime
- Acceptable: Host changes are rare, guest retries quickly

Price Consistency

Problem: Price shown in search vs price at booking time may differ.

Causes:

- Host updated price
- Seasonal pricing rule changed
- Currency exchange rate updated

Solution: Lock price at search time (with expiry)

```
// When guest views listing details
async function getListingDetails(listingId: string, checkIn: Date, checkOut: Date) {
  const pricing = await calculatePricing(listingId, checkIn, checkOut);

  // Cache price quote for 15 minutes
  const quoteId = generateUUID();
  await redis.setex(`quote:${quoteId}`, 900, JSON.stringify(pricing));

  return {
    listing: {...},
    pricing,
    quoteId // Return to client
  };
}

// When guest books
async function createBooking(req: BookingRequest & { quoteId?: string }) {
  let pricing: PricingBreakdown;

  if (req.quoteId) {
    // Try to use cached quote
    const cached = await redis.get(`quote:${req.quoteId}`);
    if (cached) {
      pricing = JSON.parse(cached);
    }
  }

  if (!pricing) {
    // Quote expired or not provided: recalculate
    pricing = await calculatePricing(req.listingId, req.checkIn, req.checkOut);

    // If price changed significantly, warn user
    if (req.expectedPrice && Math.abs(pricing.total - req.expectedPrice) >
req.expectedPrice * 0.05) {
      throw new Error('Price has changed. Please review the new price before
booking.');
    }
  }

  // ... proceed with booking using locked pricing ...
}
```

Trade-off:

- Guest sees consistent price for 15 minutes
- After expiry, price may change (user must re-confirm)
- Prevents "bait and switch" complaints

Message Ordering

Problem: Host-guest messages must appear in correct order.

Challenge: Messages sent from different servers, stored in sharded database.

Solution: Use timestamp + sequence number

```
interface Message {
  id: string; // UUID
  conversationId: string;
  senderId: string;
  recipientId: string;
  content: string;
  timestamp: number; // Unix timestamp (ms)
  sequence: number;  // Auto-increment per conversation
  createdAt: Date;
}

// Sequence generation (PostgreSQL)
CREATE SEQUENCE conversation_message_seq_123 START 1;

// Insert message
async function sendMessage(msg: MessageInput) {
  const result = await db.query(
    `INSERT INTO messages (
      id, conversation_id, sender_id, recipient_id, content,
      timestamp, sequence, created_at
    ) VALUES (
      $1, $2, $3, $4, $5,
      $6, nextval('conversation_message_seq_${msg.conversationId}'), NOW()
    ) RETURNING *`,
    [generateUUID(), msg.conversationId, msg.senderId, msg.recipientId, msg.content,
Date.now()]
  );

  return result.rows[0];
}

// Retrieve messages (ordered)
async function getMessages(conversationId: string, limit: number) {
  return db.query(
    `SELECT * FROM messages
     WHERE conversation_id = $1
     ORDER BY sequence ASC
     LIMIT $2`,
    [conversationId, limit]
  );
}
```

Why this works:

- Sequence numbers are strictly increasing per conversation
- Even if timestamps are slightly off (clock skew), sequence ensures order
- Client sorts by sequence, not timestamp

Review Visibility (Double-Blind Reviews)

Requirement: Both host and guest must submit reviews before either review becomes visible.

Implementation:

```
interface Review {
  id: string;
  bookingId: string;
  reviewerId: string;
  revieweeId: string;
  rating: number;
  text: string;
  visible: boolean; // Initially false
  createdAt: Date;
}

async function submitReview(review: ReviewInput) {
  await db.transaction(async (tx) => {
    // Insert review
    await tx.query(
      `INSERT INTO reviews (
        id, booking_id, reviewer_id, reviewee_id, rating, text, visible, created_at
      ) VALUES ($1, $2, $3, $4, $5, $6, false, NOW())`,
      [generateUUID(), review.bookingId, review.reviewerId, review.revieweeId,
review.rating, review.text]
    );

    // Check if both parties have reviewed
    const reviewCount = await tx.query(
      `SELECT COUNT(*) as count FROM reviews WHERE booking_id = $1`,
      [review.bookingId]
    );

    if (reviewCount.rows[0].count === 2) {
      // Both reviews submitted: make both visible
      await tx.query(
        `UPDATE reviews SET visible = true WHERE booking_id = $1`,
        [review.bookingId]
      );
    }
  });
}
```

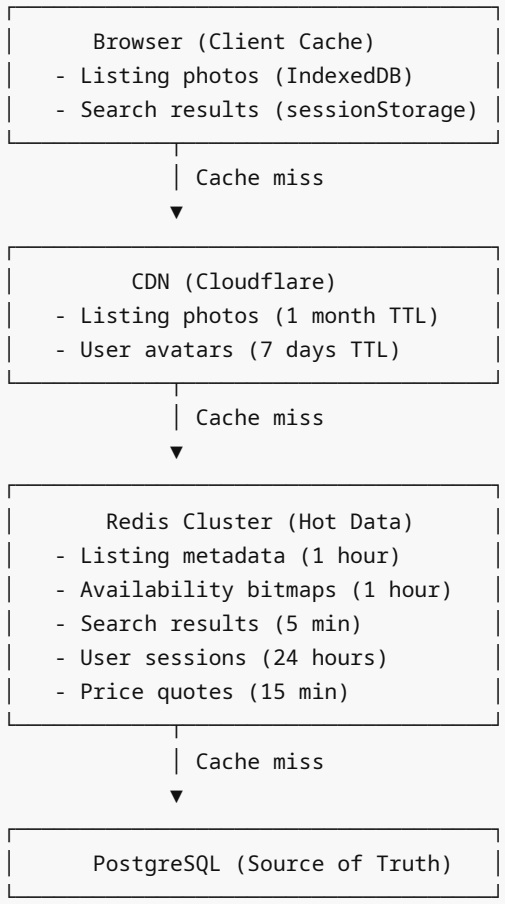
Consequence:

- Guest submits review → sees "waiting for host to review"
- Host submits review → both reviews immediately visible
- Prevents bias (can't see other's review before writing yours)

---

9. Caching Strategy

Multi-Layer Cache Architecture



What to Cache (and Why)

1. Listing Metadata (HIGH hit rate)

```
// Cache structure
`listing:${listingId}` → JSON {
  id, title, hostId, propertyType, maxGuests,
  bedrooms, bathrooms, amenities, basePrice, currency,
  instantBook, minNights, avgRating, numReviews
}

// TTL: 1 hour
// Invalidation: On listing update by host

async function getListing(listingId: string): Promise<Listing> {
  const cached = await redis.get(`listing:${listingId}`);
  if (cached) {
    listingCacheHits.inc();
    return JSON.parse(cached);
  }
```

```
  listingCacheMisses.inc();
  const listing = await db.query('SELECT * FROM listings WHERE id = $1',
[listingId]);
  await redis.setex(`listing:${listingId}`, 3600, JSON.stringify(listing));

  return listing;
}

// Invalidate on update
async function updateListing(listingId: string, updates: Partial<Listing>) {
  await db.query('UPDATE listings SET ... WHERE id = $1', [listingId]);
  await redis.del(`listing:${listingId}`);

  // Also update Elasticsearch (async)
  await kafka.publish('listing.updated', { listingId, updates });
}
```

Why cache listings:

- Read-heavy (100:1 read:write ratio)
- Listing details viewed multiple times per booking (search → details → review → book)
- Size: ~2KB per listing × 7M listings = 14GB (fits in memory)

---

2. Availability Bitmaps (CRITICAL for performance)

```
// Cache structure
`availability:${listingId}:${year-month}` → "1111011111..." (bit string)

// TTL: 1 hour
// Invalidation: On booking creation or calendar update

async function isAvailable(listingId: string, checkIn: Date, checkOut: Date):
Promise<boolean> {
  const months = getMonthsInRange(checkIn, checkOut);

  for (const month of months) {
    const cacheKey = `availability:${listingId}:${month}`;
    let bitmap = await redis.get(cacheKey);

    if (!bitmap) {
      // Cache miss: build from DB
      bitmap = await buildAvailabilityBitmap(listingId, month.year, month.month);
      await redis.setex(cacheKey, 3600, bitmap);
    }

    // Check each day in range
    for (let d = new Date(checkIn); d < checkOut; d.setDate(d.getDate() + 1)) {
      if (d.getMonth() === month.month) {
        const day = d.getDate() - 1;
        if (bitmap[day] === '0') {
          return false;
```

```
        }
      }
    }
  }

  return true;
}

// Invalidate on booking or block
async function invalidateAvailabilityCache(listingId: string, startDate: Date,
endDate: Date) {
  const months = getMonthsInRange(startDate, endDate);
  for (const month of months) {
    await redis.del(`availability:${listingId}:${month}`);
  }
}
```

Why cache availability:

- Queried on every search (1M+ searches/minute)
- Without cache: 30 DB queries per listing × 20 results = 600 queries per search
- With cache: 2-3 Redis lookups per search
- **100x performance improvement**

---

3. Search Results (SHORT TTL)

```
// Cache structure
`search:${lat}:${lng}:${checkIn}:${checkOut}:${filters_hash}` → JSON array of
listing IDs

// TTL: 5 minutes
// Invalidation: Time-based only (no active invalidation)

async function searchListings(req: SearchRequest): Promise<Listing[]> {
  const cacheKey = generateSearchCacheKey(req);
  const cached = await redis.get(cacheKey);

  if (cached) {
    searchCacheHits.inc();
    const listingIds = JSON.parse(cached);
    return await hydrateListings(listingIds);
  }

  searchCacheMisses.inc();

  // Execute search
  const results = await executeElasticsearchQuery(req);
  const availableResults = await filterByAvailability(results, req.checkIn,
req.checkOut);

  // Cache listing IDs only (not full data)
```

```
  const listingIds = availableResults.map(l => l.id);
  await redis.setex(cacheKey, 300, JSON.stringify(listingIds));

  return availableResults;
}

function generateSearchCacheKey(req: SearchRequest): string {
  const filters = {
    minPrice: req.minPrice,
    maxPrice: req.maxPrice,
    amenities: req.amenities?.sort().join(','),
    instantBook: req.instantBook,
    propertyType: req.propertyType
  };

  const filtersHash = hashObject(filters);
  return
`search:${req.lat}:${req.lng}:${req.checkIn}:${req.checkOut}:${filtersHash}`;
}
```

Why short TTL:

- Bookings change availability constantly
- 5 minutes is acceptable staleness (guest will see error at booking if actually unavailable)
- Reduces Elasticsearch load by 80%+

---

4. User Sessions (IMPORTANT for auth)

```
// Cache structure
`session:${token}` → JSON { userId, email, roles, ... }

// TTL: 24 hours (sliding window)
// Invalidation: On logout or password change

async function getSession(token: string): Promise<Session | null> {
  const cached = await redis.get(`session:${token}`);
  if (!cached) {
    return null; // Expired or invalid
  }

  const session = JSON.parse(cached);

  // Extend TTL on each request (sliding window)
  await redis.expire(`session:${token}`, 86400);

  return session;
}

async function logout(token: string) {
  await redis.del(`session:${token}`);
}
```

Why cache sessions:

- Checked on every API request
- Database lookup would add 5-10ms per request
- Redis lookup: <1ms

---

5. Price Quotes (BUSINESS logic)

```
// Cache structure
`quote:${quoteId}` → JSON { listingId, checkIn, checkOut, pricing: {...} }

// TTL: 15 minutes
// Invalidation: Time-based only

async function createPriceQuote(listingId: string, checkIn: Date, checkOut: Date):
Promise<PriceQuote> {
  const pricing = await calculatePricing(listingId, checkIn, checkOut);

  const quoteId = generateUUID();
  const quote = { listingId, checkIn, checkOut, pricing };

  await redis.setex(`quote:${quoteId}`, 900, JSON.stringify(quote));

  return { quoteId, ...quote };
}
```

Why 15 minutes:

- Guest needs time to review listing, read reviews, decide
- Too short → annoys user with "price changed" errors
- Too long → host can't adjust prices quickly

Cache Invalidation Patterns

1. Write-Through (Listings):

```
async function updateListing(listingId: string, updates: Partial<Listing>) {
  // Write to DB
  await db.query('UPDATE listings SET ... WHERE id = $1', [listingId, updates]);

  // Delete cache (next read will repopulate)
  await redis.del(`listing:${listingId}`);
}
```

2. Write-Behind (Counters - view count, bookmark count):

```
async function incrementViewCount(listingId: string) {
  // Increment in Redis immediately
  await redis.incr(`views:${listingId}`);

  // Flush to DB periodically (background worker)
```

```
  // Worker runs every 5 minutes, batches updates
}

// Background worker
setInterval(async () => {
  const keys = await redis.keys('views:*');
  for (const key of keys) {
    const listingId = key.split(':')[1];
    const count = await redis.get(key);

    await db.query('UPDATE listings SET view_count = view_count + $1 WHERE id = $2',
[count, listingId]);
    await redis.del(key);
  }
}, 300_000); // 5 minutes
```

3. Cache-Aside (Search results):

```
// Check cache first
const cached = await redis.get(cacheKey);
if (cached) return cached;

// Cache miss: fetch from source
const data = await fetchFromSource();

// Populate cache
await redis.set(cacheKey, data, { EX: ttl });

return data;
```

Cache Stampede Prevention

Problem: Hot listing's cache expires, 1000 concurrent requests hit DB.

Solution: Locking (singleflight pattern)

```
const inflightRequests = new Map<string, Promise<any>>();

async function getListingWithStampedeProtection(listingId: string): Promise<Listing>
{
  const cacheKey = `listing:${listingId}`;

  // Check cache
  const cached = await redis.get(cacheKey);
  if (cached) {
    return JSON.parse(cached);
  }

  // Check if another request is already fetching this listing
  if (inflightRequests.has(listingId)) {
    // Wait for in-flight request to complete
```

```
    return await inflightRequests.get(listingId);
  }

  // Start new request
  const promise = (async () => {
    try {
      const listing = await db.query('SELECT * FROM listings WHERE id = $1',
[listingId]);
      await redis.setex(cacheKey, 3600, JSON.stringify(listing));
      return listing;
    } finally {
      inflightRequests.delete(listingId);
    }
  })();

  inflightRequests.set(listingId, promise);
  return await promise;
}
```

Result: Only 1 DB query even if 1000 concurrent requests.

---

10. Scaling Strategy

Database Sharding

Sharding Key Selection:

Listings: Shard by `listing_id`

- All listing data (photos, amenities, calendar) co-located
- Queries typically fetch single listing
- Host queries (`host_id`) require scatter-gather (acceptable, low frequency)

Bookings: Shard by `booking_id` or `listing_id` ?

- **Option 1**: By `booking_id`
  - Pro: Booking lookups are fast
  - Con: "All bookings for listing X" requires scatter-gather
- **Option 2**: By `listing_id`
  - Pro: Availability checks are fast (single shard)
  - Con: "All bookings for guest X" requires scatter-gather
- **Choice**: Shard by `listing_id` (availability is more critical)

Users: Shard by `user_id`

- User profile, reviews, messages all co-located

Sharding Implementation (PostgreSQL):

```
// 16 shards (sufficient for 7M listings)
const NUM_SHARDS = 16;

function getShardForListing(listingId: string): number {
```

```
  const hash = hashCode(listingId);
  return Math.abs(hash) % NUM_SHARDS;
}

function getShardConnection(shardId: number): dbConnection {
  return dbConnections[shardId];
}

// Query single listing (single shard)
async function getBookingsForListing(listingId: string): Promise<Booking[]> {
  const shard = getShardForListing(listingId);
  const conn = getShardConnection(shard);

  return await conn.query(
    'SELECT * FROM bookings WHERE listing_id = $1 ORDER BY check_in_date',
    [listingId]
  );
}

// Query all bookings for guest (scatter-gather)
async function getBookingsForGuest(guestId: string): Promise<Booking[]> {
  const promises = [];

  for (let i = 0; i < NUM_SHARDS; i++) {
    const conn = getShardConnection(i);
    promises.push(
      conn.query('SELECT * FROM bookings WHERE guest_id = $1', [guestId])
    );
  }

  const results = await Promise.all(promises);
  const allBookings = results.flatMap(r => r.rows);

  // Sort by check-in date
  return allBookings.sort((a, b) => a.checkInDate - b.checkInDate);
}
```

Read Replicas

Configuration:

- **Primary** (writes only): 1 per shard
- **Replicas** (reads): 3 per shard (distributed across regions)

Read routing:

```
enum ReadConsistency {
  STRONG = 'strong',     // Read from primary
  EVENTUAL = 'eventual'   // Read from replica
}

async function getBooking(bookingId: string, consistency: ReadConsistency =
```

```
ReadConsistency.EVENTUAL) {
  const shard = getShardForBooking(bookingId);

  if (consistency === ReadConsistency.STRONG) {
    // Read from primary
    const conn = getPrimaryConnection(shard);
    return await conn.query('SELECT * FROM bookings WHERE id = $1', [bookingId]);
  } else {
    // Read from replica (random selection for load balancing)
    const conn = getReplicaConnection(shard);
    return await conn.query('SELECT * FROM bookings WHERE id = $1', [bookingId]);
  }
}

// Use cases for strong consistency
async function checkAvailabilityForBooking(listingId: string, checkIn: Date,
checkOut: Date) {
  // MUST read from primary to avoid double-booking
  return await getBookings(listingId, ReadConsistency.STRONG);
}

// Use cases for eventual consistency
async function getGuestBookingHistory(guestId: string) {
  // Slight staleness is acceptable for history view
  return await getBookingsForGuest(guestId, ReadConsistency.EVENTUAL);
}
```

Replication Lag Handling:

```
// After booking creation, store in Redis temporarily
async function createBooking(req: BookingRequest) {
  const booking = await db.query('INSERT INTO bookings ...'); // Write to primary

  // Cache booking for 5 seconds (covers replication lag)
  await redis.setex(`booking:${booking.id}`, 5, JSON.stringify(booking));

  return booking;
}

async function getBooking(bookingId: string) {
  // Check cache first (for recently created bookings)
  const cached = await redis.get(`booking:${bookingId}`);
  if (cached) {
    return JSON.parse(cached);
  }

  // Read from replica (replication lag should have passed)
  return await db.replica.query('SELECT * FROM bookings WHERE id = $1',
[bookingId]);
}
```

Elasticsearch Scaling

Index Strategy:

- **Single index** for all listings (7M documents)
- 5 primary shards, 1 replica per shard = 10 shards total
- Each shard: ~1.4M documents, ~5GB

Scaling Limits:

- Current: 10 shards, 3 nodes
- At 50M listings: 30 shards, 10 nodes
- At 500M listings: Re-shard or move to multi-index strategy

Query Optimization:

```
// Use filters (cached) instead of queries (scored)
{
  "query": {
    "bool": {
      "must": [
        {
          "geo_distance": {
            "distance": "50km",
            "location": { "lat": 37.7749, "lon": -122.4194 }
          }
        }
      ],
      "filter": [ // Filters are cached, faster than queries
        { "term": { "status": "active" } },
        { "range": { "base_price_per_night": { "gte": 100, "lte": 300 } } },
        { "terms": { "amenities": ["wifi", "parking"] } }
      ]
    }
  }
}
```

Refresh Interval:

- Default: 1 second (too frequent for our use case)
- Set to 30 seconds (reduces indexing load)
- Trade-off: Listing updates take 30s to appear in search (acceptable)

```
PUT /listings/_settings
{
  "index": {
    "refresh_interval": "30s"
  }
}
```

Redis Cluster

Configuration:

- 6 nodes: 3 primaries + 3 replicas
- Data is automatically sharded across primaries using hash slots

Key Distribution:

```
// Redis Cluster uses CRC16 hash of key to determine slot
// 16,384 slots total, evenly distributed across 3 primaries

// Good key design (distributes evenly)
`listing:${listingId}` // listingId is UUID, uniform distribution

// Bad key design (hotspot)
`search:San Francisco:*` // All San Francisco searches on same shard
```

Avoiding Hotspots:

```
// Problem: Popular city searches all hit same Redis node
`search:San Francisco:...` → Shard 1 (overloaded)
`search:New York:...`      → Shard 1 (overloaded)

// Solution: Add random suffix to distribute
const shard = Math.floor(Math.random() * 10); // 0-9
const cacheKey = `search:${city}:${shard}:${filtersHash}`;

// Now same search is stored 10 times, different shards
// Reduces cache hit rate, but prevents hotspot
```

Memory Management:

- Set `maxmemory` policy: `allkeys-lru` (evict least recently used)
- Monitor eviction rate (should be <5%)
- If evictions spike → add more nodes

CDN (Cloudflare)

Static Asset Delivery:

- Listing photos: ~10 photos per listing × 500KB/photo = 5MB per listing
- 7M listings × 5MB = 35TB total
- CDN reduces origin traffic by 90%+

Cache Headers:

```
// Listing photos (immutable)
app.get('/photos/:photoId', (req, res) => {
  const photoUrl = getPhotoUrl(req.params.photoId);

  res.setHeader('Cache-Control', 'public, max-age=2592000, immutable'); // 30 days
  res.setHeader('CDN-Cache-Control', 'max-age=31536000'); // 1 year on CDN
```

```
  res.redirect(photoUrl);
});

// User avatars (may change)
app.get('/avatars/:userId', (req, res) => {
  const avatarUrl = getAvatarUrl(req.params.userId);

  res.setHeader('Cache-Control', 'public, max-age=86400'); // 1 day
  res.setHeader('CDN-Cache-Control', 'max-age=604800'); // 7 days on CDN

  res.redirect(avatarUrl);
});
```

CDN Purging:

```
// When host uploads new photo, purge CDN cache
async function uploadListingPhoto(listingId: string, photo: Buffer) {
  const photoId = generateUUID();

  // Upload to S3
  await s3.upload(`photos/${photoId}.jpg`, photo);

  // Purge CDN cache for this listing's photos
  await cloudflare.purgeCache([
    `https://cdn.airbnb.com/photos/${photoId}.jpg`
  ]);

  return photoId;
}
```

Rate Limiting

Strategy: Token bucket per user + IP

```
// Redis structure
`ratelimit:user:${userId}` → number (tokens remaining)
`ratelimit:ip:${ipAddress}` → number (tokens remaining)

async function checkRateLimit(userId: string, ipAddress: string): Promise<boolean> {
  const userKey = `ratelimit:user:${userId}`;
  const ipKey = `ratelimit:ip:${ipAddress}`;

  // User rate limit: 100 requests/minute
  const userTokens = await redis.incr(userKey);
  if (userTokens === 1) {
    await redis.expire(userKey, 60); // 1 minute window
  }
  if (userTokens > 100) {
    return false; // Rate limit exceeded
  }
```

```
  // IP rate limit: 1000 requests/minute (prevents DDoS)
  const ipTokens = await redis.incr(ipKey);
  if (ipTokens === 1) {
    await redis.expire(ipKey, 60);
  }
  if (ipTokens > 1000) {
    return false;
  }

  return true;
}
```

Per-Endpoint Limits:

- Search: 60/min per user (prevent scraping)
- Booking: 10/min per user (prevent spam)
- Listing creation: 5/hour per host (prevent abuse)

Autoscaling

Node.js API Servers:

- **Metric**: CPU > 70% for 5 minutes → scale up
- **Metric**: Request latency P95 > 1s → scale up
- **Min**: 10 instances
- **Max**: 100 instances
- **Scale-down**: Gradual (1 instance per 10 minutes to avoid flapping)

Go Search Service:

- **Metric**: Request queue depth > 100 → scale up
- **Min**: 20 instances (search is critical)
- **Max**: 200 instances

Kafka Consumers:

- **Metric**: Consumer lag > 10,000 messages → scale up
- Scale by partition count (one consumer per partition)

---

11. Fault Tolerance & Reliability

Failure Modes & Mitigations

1. Booking Service Goes Down

Impact: Can't create new bookings (revenue loss)

Mitigation:

- Multiple instances behind load balancer (10+ replicas)
- Health checks: `/health` endpoint every 10 seconds
- Auto-restart failed containers (Kubernetes liveness probes)
- Circuit breaker: If 50% of requests fail, stop routing traffic

- Fallback: Queue booking requests in Kafka, process when service recovers

Circuit Breaker Implementation:

```
enum CircuitState {
  CLOSED = 'closed',   // Normal operation
  OPEN = 'open',       // Failing, reject requests
  HALF_OPEN = 'half_open' // Testing if recovered
}

class CircuitBreaker {
  private state: CircuitState = CircuitState.CLOSED;
  private failures = 0;
  private successCount = 0;
  private lastFailureTime: number = 0;

  async execute<T>(fn: () => Promise<T>): Promise<T> {
    if (this.state === CircuitState.OPEN) {
      // Check if enough time has passed to try again
      if (Date.now() - this.lastFailureTime > 60000) {
        this.state = CircuitState.HALF_OPEN;
      } else {
        throw new Error('Circuit breaker is OPEN. Service unavailable.');
      }
    }

    try {
      const result = await fn();
      this.onSuccess();
      return result;
    } catch (err) {
      this.onFailure();
      throw err;
    }
  }

  private onSuccess() {
    this.failures = 0;
    if (this.state === CircuitState.HALF_OPEN) {
      this.successCount++;
      if (this.successCount >= 5) {
        this.state = CircuitState.CLOSED;
        this.successCount = 0;
      }
    }
  }

  private onFailure() {
    this.failures++;
    this.lastFailureTime = Date.now();

    if (this.failures >= 10) {
```

```
      this.state = CircuitState.OPEN;
    }
  }
}

// Usage
const bookingServiceBreaker = new CircuitBreaker();

async function createBooking(req: BookingRequest) {
  try {
    return await bookingServiceBreaker.execute(() => bookingService.create(req));
  } catch (err) {
    // Fallback: Queue for later processing
    await kafka.publish('booking.pending', req);
    throw new Error('Booking service temporarily unavailable. You will receive
confirmation via email.');
  }
}
```

---

2. PostgreSQL Primary Goes Down

Impact: Can't write new bookings, can't update listings

Mitigation:

- Automatic failover to replica (promote replica to primary)
- Failover time: 30-60 seconds
- During failover: Queue writes in Kafka
- After failover: Replay queued writes

Failover Detection:

```
// Health check every 5 seconds
setInterval(async () => {
  try {
    await db.primary.query('SELECT 1');
    primaryHealthy = true;
  } catch (err) {
    primaryHealthy = false;

    if (failureCount++ > 3) {
      // 3 consecutive failures → trigger failover
      await promoteReplicaToPrimary();
    }
  }
}, 5000);

async function promoteReplicaToPrimary() {
  // 1. Stop accepting writes
  acceptingWrites = false;

  // 2. Promote replica
```

```
  await replica1.query('SELECT pg_promote()');

  // 3. Update connection pool to point to new primary
  db.primary = replica1;

  // 4. Resume accepting writes
  acceptingWrites = true;

  // 5. Alert on-call engineer
  await pagerduty.alert('PostgreSQL failover completed');
}
```

---

3. Redis Cluster Goes Down

Impact: Cache misses, slower responses (but still functional)

Mitigation:

- Redis is cache only, not source of truth
- On cache miss, fall back to database
- Performance degrades, but system stays up
- Monitor cache hit rate: alert if <80%

Graceful Degradation:

```
async function getListing(listingId: string): Promise<Listing> {
  try {
    const cached = await redis.get(`listing:${listingId}`);
    if (cached) {
      return JSON.parse(cached);
    }
  } catch (err) {
    // Redis down, log error but continue
    logger.error('Redis unavailable', err);
  }

  // Fall back to database
  return await db.query('SELECT * FROM listings WHERE id = $1', [listingId]);
}
```

---

4. Elasticsearch Goes Down

Impact: Search doesn't work (can't find listings)

Mitigation:

- Multi-node cluster (3+ nodes)
- Replicas for every shard (1 replica per shard)
- If primary shard fails, replica promoted automatically
- If entire cluster fails, fallback to PostgreSQL search (slower, but functional)

Fallback Search:

```
async function searchListings(req: SearchRequest): Promise<Listing[]> {
  try {
    return await elasticsearchSearch(req);
  } catch (err) {
    logger.error('Elasticsearch unavailable, falling back to PostgreSQL', err);
    return await postgresqlSearch(req);
  }
}

async function postgresqlSearch(req: SearchRequest): Promise<Listing[]> {
  // Much slower, but works
  return await db.query(
    `SELECT * FROM listings
     WHERE ST_Distance(location, ST_Point($1, $2)) < 50000
     AND base_price_per_night BETWEEN $3 AND $4
     AND status = 'active'
     LIMIT 20`,
    [req.lng, req.lat, req.minPrice, req.maxPrice]
  );
}
```

---

5. Payment Service (Stripe) Fails

Impact: Can't create bookings (payment authorization fails)

Mitigation:

- Retry with exponential backoff (3 attempts)
- If all retries fail, return error to user (don't create booking)
- Store failed payment attempts in database for manual review

Retry Logic:

```
async function authorizePayment(amount: number, paymentMethodId: string):
Promise<PaymentIntent> {
  const maxRetries = 3;
  let attempt = 0;

  while (attempt < maxRetries) {
    try {
      return await stripe.paymentIntents.create({
        amount: amount * 100,
        currency: 'usd',
        payment_method: paymentMethodId,
        capture_method: 'manual'
      });
    } catch (err) {
      attempt++;

      if (attempt >= maxRetries) {
        // Log for manual review
        await db.query(
```

```
          'INSERT INTO failed_payments (amount, payment_method_id, error,
created_at) VALUES ($1, $2, $3, NOW())',
          [amount, paymentMethodId, err.message]
        );

        throw new Error('Payment processing failed. Please try again or use a
different payment method.');
      }

      // Exponential backoff: 1s, 2s, 4s
      await sleep(Math.pow(2, attempt) * 1000);
    }
  }
}
```

---

6. Kafka Goes Down

Impact: Event processing stops (notifications delayed, analytics paused)

Mitigation:

- Kafka is used for async processing only
- Booking creation doesn't depend on Kafka (synchronous flow completes without it)
- When Kafka recovers, consumers catch up (process backlog)
- Monitor consumer lag: alert if >10,000 messages

Idempotent Event Processing:

```
// Events may be processed twice if Kafka consumer restarts
async function handleBookingCreatedEvent(event: BookingCreatedEvent) {
  // Check if already processed (idempotency)
  const processed = await db.query(
    'SELECT id FROM processed_events WHERE event_id = $1',
    [event.id]
  );

  if (processed.rows.length > 0) {
    logger.info('Event already processed, skipping', event.id);
    return;
  }

  // Process event
  await sendBookingConfirmationEmail(event);

  // Mark as processed
  await db.query(
    'INSERT INTO processed_events (event_id, processed_at) VALUES ($1, NOW())',
    [event.id]
  );
}
```

Data Durability

RPO (Recovery Point Objective): 0 seconds

- Synchronous replication to 2 replicas
- Write is not acknowledged until both replicas have persisted

RTO (Recovery Time Objective): 60 seconds

- Automatic failover to replica
- DNS update to point to new primary
- Application connection pool refreshes

Backup Strategy:

- Full database backup: Daily at 2 AM UTC
- Incremental backups: Every 6 hours
- WAL (Write-Ahead Log) archiving: Continuous
- Retention: 30 days

Backup Restoration Test:

- Monthly drill: Restore from backup to test cluster
- Verify data integrity
- Measure restoration time

Chaos Engineering

Practices:

- Randomly kill 1 API server instance per day (test auto-restart)
- Simulate database failover quarterly (test failover automation)
- Inject latency into Redis calls (test graceful degradation)
- Shut down 1 Elasticsearch node (test replica promotion)

Goal: Build confidence that system survives failures.

---

12. Observability & Operations

Metrics (Prometheus + Grafana)

Golden Signals:

1. Latency:

```
// Histogram: Track request duration
const httpRequestDuration = new prometheus.Histogram({
  name: 'http_request_duration_seconds',
  help: 'HTTP request latency',
  labelNames: ['method', 'route', 'status'],
  buckets: [0.01, 0.05, 0.1, 0.5, 1, 2, 5]
});

app.use((req, res, next) => {
  const start = Date.now();
```

```
  res.on('finish', () => {
    const duration = (Date.now() - start) / 1000;
    httpRequestDuration.observe({
      method: req.method,
      route: req.route?.path || 'unknown',
      status: res.statusCode
    }, duration);
  });

  next();
});
```

Key Metrics:

- `http_request_duration_seconds{route="/api/v1/search", p95}` < 500ms
- `http_request_duration_seconds{route="/api/v1/bookings", p95}` < 1s
- Alert if P95 exceeds threshold for 5 minutes

2. Traffic:

```
const httpRequestsTotal = new prometheus.Counter({
  name: 'http_requests_total',
  help: 'Total HTTP requests',
  labelNames: ['method', 'route', 'status']
});
```

Key Metrics:

- `http_requests_total{route="/api/v1/search"}` (should be stable, spikes indicate issues)
- `http_requests_total{status="500"}` (alert if >1% of requests)

3. Errors:

```
const bookingCreationFailures = new prometheus.Counter({
  name: 'booking_creation_failures_total',
  help: 'Failed booking attempts',
  labelNames: ['reason']
});

// In booking creation flow
catch (err) {
  if (err.message.includes('dates not available')) {
    bookingCreationFailures.inc({ reason: 'unavailable' });
  } else if (err.message.includes('payment')) {
    bookingCreationFailures.inc({ reason: 'payment_failed' });
  } else {
    bookingCreationFailures.inc({ reason: 'unknown' });
  }
  throw err;
}
```

Key Metrics:

- `booking_creation_failures_total{reason="payment_failed"}` (alert if spike)
- `booking_creation_failures_total{reason="unknown"}` (investigate immediately)

4. Saturation:

```
// Database connection pool usage
const dbConnectionPoolUsage = new prometheus.Gauge({
  name: 'db_connection_pool_usage',
  help: 'Number of active database connections'
});

setInterval(() => {
  dbConnectionPoolUsage.set(db.pool.totalCount);
}, 10000);
```

Key Metrics:

- `db_connection_pool_usage / db_connection_pool_max` (alert if >80%)
- `redis_memory_usage / redis_memory_max` (alert if >85%)
- `elasticsearch_jvm_memory_usage` (alert if >90%)

Business Metrics

Booking Funnel:

```
// Track conversion rates
const searchCount = new prometheus.Counter({ name: 'search_count' });
const listingViews = new prometheus.Counter({ name: 'listing_views' });
const bookingAttempts = new prometheus.Counter({ name: 'booking_attempts' });
const bookingSuccesses = new prometheus.Counter({ name: 'booking_successes' });

// Dashboard shows:
// Search → View: 20% (100M searches → 20M views)
// View → Attempt: 10% (20M views → 2M attempts)
// Attempt → Success: 90% (2M attempts → 1.8M bookings)
```

Revenue Metrics:

```
const bookingRevenue = new prometheus.Counter({
  name: 'booking_revenue_total',
  help: 'Total booking revenue',
  labelNames: ['currency']
});

// On booking creation
bookingRevenue.inc({ currency: booking.currency }, booking.totalAmount);
```

Logging (Structured Logs)

Log Levels:

- **DEBUG**: Verbose, noisy (disabled in production)
- **INFO**: Normal operations (search queries, booking creation)
- **WARN**: Degraded performance (cache miss, slow query)
- **ERROR**: Failures (payment failed, DB connection lost)
- **FATAL**: System crash (uncaught exception)

Structured Format (JSON):

```
logger.info('Booking created', {
  bookingId: booking.id,
  listingId: booking.listingId,
  guestId: booking.guestId,
  checkInDate: booking.checkInDate,
  totalAmount: booking.totalAmount,
  currency: booking.currency,
  durationMs: Date.now() - startTime
});

// Outputs:
{
  "timestamp": "2026-02-14T10:30:00.000Z",
  "level": "info",
  "message": "Booking created",
  "bookingId": "booking-456",
  "listingId": "listing-123",
  "guestId": "user-789",
  "checkInDate": "2026-07-01",
  "totalAmount": 945,
  "currency": "USD",
  "durationMs": 234
}
```

Why Structured Logs:

- Easy to query: `bookingId="booking-456"` retrieves all logs for that booking
- Aggregation: Count bookings by currency, average duration

Distributed Tracing (OpenTelemetry)

Trace a single booking flow:

```
Trace ID: abc123
├─ Span: HTTP POST /api/v1/bookings (500ms)
   ├─ Span: Acquire distributed lock (10ms)
   ├─ Span: Check availability (50ms)
   │  ├─ Span: Redis GET availability:listing-123:2026-07 (2ms)
   │  └─ Span: PostgreSQL SELECT FROM bookings (48ms)
   ├─ Span: Calculate pricing (30ms)
   ├─ Span: Authorize payment (200ms)
   │  └─ Span: Stripe API call (195ms)
```

```
    ├─ Span: Create booking in DB (80ms)
    └─ Span: Publish Kafka event (5ms)
```

Trace shows:

- Stripe API call is slowest (195ms)
- PostgreSQL query is slow (48ms) → investigate index usage
- Total: 500ms (meets SLA of <1s)

Implementation:

```
import { trace } from '@opentelemetry/api';

async function createBooking(req: BookingRequest) {
  const tracer = trace.getTracer('booking-service');

  return tracer.startActiveSpan('createBooking', async (span) => {
    span.setAttribute('bookingId', req.bookingId);
    span.setAttribute('listingId', req.listingId);

    try {
      const booking = await processBooking(req);
      span.setStatus({ code: SpanStatusCode.OK });
      return booking;
    } catch (err) {
      span.setStatus({ code: SpanStatusCode.ERROR, message: err.message });
      throw err;
    } finally {
      span.end();
    }
  });
}
```

Alerts (PagerDuty)

Critical Alerts (Page On-Call Engineer):

- Booking creation error rate >5% for 5 minutes
- Database primary down
- Search service error rate >10% for 3 minutes
- Payment authorization success rate <95% for 5 minutes

Warning Alerts (Slack Only):

- Cache hit rate <80% for 10 minutes
- API response time P95 >2s for 10 minutes
- Database connection pool >80% usage for 10 minutes

Alert Fatigue Prevention:

- No alerts for transient issues (<5 minutes)
- Aggregate similar alerts (don't page 100 times)
- Clear runbooks for each alert

Debugging Runbook (3 AM Scenario)

Engineer receives page: "Booking creation error rate >5%"

Step 1: Check Grafana dashboard

- Which error type is spiking?
  - Payment failures → Check Stripe status page
  - Database errors → Check DB health
  - Unavailable errors → Check if calendar sync is broken

Step 2: Check recent deployments

- Was code deployed in last hour? → Rollback
- Was config changed? → Revert

Step 3: Check logs

- Query: `level=error AND message="Booking creation failed"` for last 10 minutes
- Look for common patterns (same listing ID failing repeatedly?)

Step 4: Check dependencies

- Stripe API latency elevated?
- PostgreSQL query slow?
- Redis connection pool exhausted?

Step 5: Mitigation

- If payment failures: Switch to fallback payment processor
- If database overload: Scale up read replicas
- If bug in code: Rollback deployment

---

13. Security & Abuse Prevention

Authentication & Authorization

JWT Tokens:

```
interface JWTPayload {
  userId: string;
  email: string;
  roles: string[]; // ['guest', 'host', 'admin']
  exp: number; // Expiry timestamp
}

// On login
function generateToken(user: User): string {
  const payload: JWTPayload = {
    userId: user.id,
    email: user.email,
    roles: user.roles,
    exp: Date.now() + 24 * 60 * 60 * 1000 // 24 hours
  };
```

```
  return jwt.sign(payload, process.env.JWT_SECRET);
}

// On every request
function authenticate(req: Request): User {
  const token = req.headers.authorization?.replace('Bearer ', '');
  if (!token) {
    throw new UnauthorizedError('Missing token');
  }

  try {
    const payload = jwt.verify(token, process.env.JWT_SECRET) as JWTPayload;
    return { id: payload.userId, email: payload.email, roles: payload.roles };
  } catch (err) {
    throw new UnauthorizedError('Invalid token');
  }
}

// Authorization middleware
function requireRole(role: string) {
  return (req: Request, res: Response, next: NextFunction) => {
    const user = req.user; // Set by authenticate middleware

    if (!user.roles.includes(role)) {
      return res.status(403).json({ error: 'Forbidden' });
    }

    next();
  };
}

// Usage
app.post('/api/v1/listings', authenticate, requireRole('host'), createListing);
```

Rate Limiting (Abuse Prevention)

Per-user limits:

```
// Search: 60 requests/minute
// Booking: 10 requests/minute
// Listing creation: 5 requests/hour

async function rateLimitMiddleware(req: Request, res: Response, next: NextFunction)
{
  const userId = req.user?.id || req.ip; // Use IP if not authenticated
  const endpoint = req.route.path;

  const limit = getRateLimitForEndpoint(endpoint);
  const key = `ratelimit:${endpoint}:${userId}`;

  const current = await redis.incr(key);
```

```
  if (current === 1) {
    await redis.expire(key, limit.windowSeconds);
  }

  if (current > limit.maxRequests) {
    return res.status(429).json({
      error: 'Rate limit exceeded',
      retryAfter: await redis.ttl(key)
    });
  }

  res.setHeader('X-RateLimit-Limit', limit.maxRequests);
  res.setHeader('X-RateLimit-Remaining', limit.maxRequests - current);

  next();
}
```

Fraud Detection

Suspicious Patterns:

1. Booking Spam:

- Same user creates 10+ bookings in 1 minute
- Same payment method used for bookings at different locations simultaneously

Detection:

```
async function detectBookingSpam(userId: string): Promise<boolean> {
  const recentBookings = await db.query(
    `SELECT COUNT(*) as count FROM bookings
     WHERE guest_id = $1
     AND created_at > NOW() - INTERVAL '1 minute'`,
    [userId]
  );

  if (recentBookings.rows[0].count > 10) {
    await flagUser(userId, 'booking_spam');
    return true;
  }

  return false;
}
```

2. Fake Listings:

- Listing photos stolen from other sites (reverse image search)
- Host account created recently with no reviews

Detection:

```
async function reviewNewListing(listingId: string) {
  const listing = await getListing(listingId);

  // Check if host is new
  const host = await getUser(listing.hostId);
  if (daysSince(host.createdAt) < 7) {
    await flagListing(listingId, 'new_host');
  }

  // Check if photos are duplicates
  for (const photo of listing.photos) {
    const duplicates = await reverseImageSearch(photo.url);
    if (duplicates.length > 0) {
      await flagListing(listingId, 'duplicate_photos');
    }
  }
}
```

3. Payment Fraud:

- Stolen credit cards
- Chargeback after stay

Mitigation:

- Require 3D Secure (3DS) for high-risk transactions
- Hold payment until 48 hours after check-in
- Payout to host only after checkout

Data Privacy (GDPR, CCPA)

Right to Delete:

```
app.delete('/api/v1/users/:userId/account', authenticate, async (req, res) => {
  const userId = req.params.userId;

  // Verify user owns account
  if (req.user.id !== userId) {
    return res.status(403).json({ error: 'Forbidden' });
  }

  // Start deletion job (async, takes hours)
  await kafka.publish('user.deletion.requested', { userId });

  res.json({ message: 'Account deletion initiated. This may take 24-48 hours.' });
});

// Background worker
async function handleUserDeletion(userId: string) {
  // 1. Cancel future bookings
  await db.query(
    `UPDATE bookings SET status = 'cancelled', cancelled_by = 'system'
```

```
     WHERE guest_id = $1 AND check_in_date > NOW()`,
    [userId]
  );

  // 2. Anonymize past bookings (retain for legal/financial reasons)
  await db.query(
    `UPDATE bookings SET guest_id = 'deleted-user' WHERE guest_id = $1`,
    [userId]
  );

  // 3. Delete messages
  await db.query(`DELETE FROM messages WHERE sender_id = $1 OR recipient_id = $1`,
[userId]);

  // 4. Delete reviews (or anonymize)
  await db.query(`UPDATE reviews SET reviewer_id = 'deleted-user' WHERE reviewer_id
= $1`, [userId]);

  // 5. Delete user account
  await db.query(`DELETE FROM users WHERE id = $1`, [userId]);

  // 6. Delete from Redis cache
  await redis.del(`user:${userId}`);

  // 7. Delete S3 photos
  await s3.deleteObjects({ Bucket: 'user-avatars', Keys: [`${userId}.jpg`] });
}
```

SQL Injection Prevention

Always use parameterized queries:

```
// WRONG (vulnerable to SQL injection)
const query = `SELECT * FROM listings WHERE city = '${req.query.city}'`;
await db.query(query);

// CORRECT (safe)
await db.query('SELECT * FROM listings WHERE city = $1', [req.query.city]);
```

XSS Prevention

Sanitize user input before displaying:

```
import DOMPurify from 'dompurify';

// Listing description (user-provided HTML)
function sanitizeListingDescription(html: string): string {
  return DOMPurify.sanitize(html, {
    ALLOWED_TAGS: ['b', 'i', 'em', 'strong', 'p', 'br', 'ul', 'ol', 'li'],
    ALLOWED_ATTR: []
```

```
  });
}
```

HTTPS & TLS

All traffic encrypted:

- TLS 1.3 only (disable older versions)
- Certificate management: Let's Encrypt (auto-renewal)
- HSTS header: `Strict-Transport-Security: max-age=31536000; includeSubDomains`

Insider Threat

Database Access:

- Engineers cannot query production database directly
- All queries logged and audited
- Sensitive fields (payment methods, SSN) encrypted at rest

Encryption:

```
// Encrypt sensitive data before storing
import { encrypt, decrypt } from './crypto';

async function storePaymentMethod(userId: string, cardNumber: string) {
  const encrypted = encrypt(cardNumber, process.env.ENCRYPTION_KEY);
  await db.query(
    'INSERT INTO payment_methods (user_id, encrypted_card) VALUES ($1, $2)',
    [userId, encrypted]
  );
}

async function getPaymentMethod(userId: string): Promise<string> {
  const result = await db.query(
    'SELECT encrypted_card FROM payment_methods WHERE user_id = $1',
    [userId]
  );

  return decrypt(result.rows[0].encrypted_card, process.env.ENCRYPTION_KEY);
}
```

Airbnb PASS 2 complete.

Key topics covered:

- **Consistency**: Distributed locking + row locking prevents double-booking
- **Caching**: 5-layer cache (browser → CDN → Redis → Elasticsearch → PostgreSQL)
- **Scaling**: Sharding by listing_id, 16 shards, read replicas, autoscaling
- **Reliability**: Circuit breakers, failover automation, graceful degradation, RPO=0s, RTO=60s
- **Observability**: Prometheus metrics (golden signals + business metrics), structured logs, distributed tracing, PagerDuty alerts

- **Security**: JWT auth, rate limiting, fraud detection, GDPR compliance, encryption

Total sections: 13 (Problem → Architecture → Workflows → Consistency → Scaling → Observability → Security)