Distributed GUID System Design

Architecture Diagram

```
[Client: Payment/Booking Service] <--> [CDN: CloudFront]
<--> [Load Balancer: ELB]
                                    |
                                    v
[API Gateway] <--> [GUID Service]
  |                |
  |                v
  |   [GUID Generator Nodes] <--> [Redis: Node Metadata]
  |   [ZooKeeper: Node Coordination]
  |   [Kafka: GUID Assignment]
  |   [PostgreSQL: GUID Metadata (Optional)]
  |   [Payment Service] <--> [3rd Party: Stripe]
  |   [Booking Service] <--> [PostgreSQL: Bookings]
  |   [Notification Service] <--> [WebSocket/Kafka Pub-Sub]
  |   [Recommendation Service] <--> [Two-Tower Model + Faiss
(HNSW)]
  |
  v
[Message Queue: Kafka] <--> [Consumer Service]
  |
  v
[Background Jobs: Lambda] <--> [Redis: Cleanup]
```

Components

1. Client

- **Services**: Payment Service, Booking Service, and other microservices in the Buy Ticket Service requesting GUIDs.
- **Interaction**: Call GUID Service via API (e.g., POST /guid/generate) to obtain unique IDs for payments, bookings, etc.
- **Frontend**: Web/mobile apps (React) display GUIDs in booking confirmations.

2. CDN

- **Purpose**: Serve static content (e.g., API documentation) for low latency.
- **Tech**: AWS CloudFront, minimal usage as GUID generation is dynamic.

3. Load Balancer

- **Purpose**: Distribute 100K QPS across GUID Service instances.
- **Tech**: AWS Elastic Load Balancer (ELB) with Round Robin algorithm.

4. API Gateway

- **Purpose**: Route GUID requests, enforce rate limiting (e.g., 1000 requests/sec/client), and handle authentication (JWT optional).
- **Endpoints**:
  - POST /guid/generate?type={uuid|snowflake}: Generate a GUID (UUID or Snowflake-like ID).
  - GET /guid/status/{guid}: Check GUID metadata (e.g., creation time, node).
- **Tech**: AWS API Gateway, rate limiting to prevent abuse.

5. GUID Service

- **Purpose**: Generate unique GUIDs across distributed nodes without duplication.
- **Implementation (Snowflake-like ID Generator)**:
  - **ID Structure**: 64-bit ID composed of:
    - **Timestamp (41 bits)**: Milliseconds since epoch (e.g., 2025-01-01), supports ~69 years.
    - **Node ID (10 bits)**: Unique per node (supports 1024 nodes).
    - **Sequence Counter (12 bits)**: Per-node counter (supports 4096 IDs/millisecond/node).
    - **Random Bits (1 bit)**: Optional for additional entropy.
    - Total: 41 + 10 + 12 + 1 = 64 bits.
  - **Generation Logic**:

- Each node generates IDs locally using:
  `def generate_snowflake_id(node_id, epoch=2025_01_01_ms):`
- `    timestamp = int(time.time() * 1000) - epoch`
- `    sequence = redis.incr(f"guid:node:{node_id}:sequence") % 4096`
- `    return (timestamp << 23) | (node_id << 12) | sequence`
- **Node ID Assignment**: ZooKeeper assigns unique node IDs (0-1023) at startup.
- **Sequence Counter**: Redis increments per-node counter (guid:node:{node_id}:sequence, reset per millisecond).
- **Timestamp**: Synchronized via NTP (Network Time Protocol) to avoid clock drift.

  - **Uniqueness Guarantee**:
    - Timestamp ensures IDs are unique over time.
    - Node ID ensures uniqueness across nodes.
    - Sequence counter handles high-frequency requests within the same millisecond.
  - **Fault Tolerance**:
    - If a node fails, ZooKeeper reassigns its ID to a new node after a timeout.
    - Clock drift handled by waiting if timestamp decreases (rare, <1ms).
  - **Scalability**:
    - 1024 nodes × 4096 IDs/ms/node = 4.2M IDs/second, sufficient for 100K QPS.

- **Flow**:
  - Payment Service calls POST /guid/generate → GUID Service node generates Snowflake ID → Returns ID (e.g., 1234567890123456789) in <5ms.

6. ZooKeeper

- **Purpose**: Coordinate node IDs and ensure uniqueness across distributed nodes.
- **Implementation**:
    - Maintain a registry of active nodes (/guid/nodes/{node_id}).
    - Assign unique node IDs (0-1023) to GUID Service instances at startup.
    - Handle node failures by reassigning IDs after a lease timeout (e.g., 30s).
- **Flow**:
    - New GUID node starts → ZooKeeper assigns ID 42 → Node uses ID for Snowflake generation.

7. Redis

- **Purpose**: Store sequence counters and handle idempotency for downstream services.
- **Implementation**:
    - **Sequence Counter**: Increment guid:node:{node_id}:sequence per millisecond (reset via Lua script).

```
local key = KEYS[1]

local current_time = redis.call('TIME')[1] * 1000
local last_time = redis.call('GET', key ..
':last_time') or 0
if current_time > last_time then
    redis.call('SET', key .. ':last_time',
current_time)
    redis.call('SET', key, 0)
end
return redis.call('INCR', key)
```

    - **Idempotency**: Store processed GUIDs for downstream services (e.g., payments:processed:{payment_id}, TTL=24h).
- **Flow**:
    - GUID node requests sequence → Redis increments counter → Returns sequence (1ms).

8. Kafka

- **Purpose**: Handle async GUID assignment for downstream services (e.g., Payment Service, Booking Service).

- **Implementation**:
    - Topic: guid_assignments for exactly-once delivery of GUIDs to consumers.
    - Producer: GUID Service sends GUIDs transactionally:
      ```
      from kafka import KafkaProducer
      ```
    - ```
      producer =
      KafkaProducer(bootstrap_servers=['localhost:9092']
      ,
      ```
    - ```
                               enable_idempotence=True,
      ```
    - ```
      transactional_id='guid_txn')
      ```
    - ```
      producer.init_transactions()
      ```
    - ```
      producer.begin_transaction()
      ```
    - ```
      producer.send('guid_assignments',
      ```
    - ```
                    value={'guid':
      '12345678901234567889', 'service': 'payment',
      ```
    - ```
                           'booking_id': 'book_1001'})
      ```
    - ```
      producer.commit_transaction()
      ```
    - Consumer: Payment/Booking Service processes GUIDs transactionally.
- **Flow**:
    - GUID generated → Kafka enqueues to guid_assignments → Payment Service consumes (150ms).

9. PostgreSQL (Optional)

- **Purpose**: Store GUID metadata for auditing (e.g., creation time, node ID).
- **Schema**:
  ```
  CREATE TABLE GUIDs (
  ```
- ```
      guid BIGINT PRIMARY KEY,
  ```
- ```
      node_id INT,
  ```
- ```
      created_at DATETIME,
  ```
- ```
      service VARCHAR(50) -- e.g., 'payment', 'booking'
  ```
- ```
  );
  ```
- **Use Case**: Rarely queried, used for debugging or compliance.

10. Downstream Integration

- **Payment Service**:

- Requests GUID for payment_id → Uses in Stripe payment intent.
- Exactly-once delivery via Kafka transactions.

- **Booking Service**:
  - Consumes GUID for booking_id → Updates Bookings and Seats tables.
  - Example:

```
INSERT INTO Bookings (booking_id, user_id,
event_id, seat_id, status)
VALUES ('book_1001', 42, 1234, 'A1', 'CONFIRMED')
ON CONFLICT (booking_id) DO NOTHING;
```

- **Notification Service**:
  - Sends at-most-once notifications (e.g., "Booking ID book_1001 confirmed") via WebSocket.

- **Recommendation Service**:
  - Uses Two-Tower Model + Faiss (HNSW) to suggest events if booking fails (~40ms, NDCG@10=0.87).

11. Exactly-Once Delivery

- **Approach**: Transactional Messaging (Kafka).
- **Implementation**:
  - GUID Service sends GUIDs to guid_assignments topic transactionally.
  - Consumer (Payment Service) processes GUIDs and updates PostgreSQL atomically:

```
consumer = KafkaConsumer('guid_assignments',

bootstrap_servers=['localhost:9092'],

isolation_level='read_committed')
for msg in consumer:
    guid_data = msg.value
    with psycopg2.connect(...) as conn:
        with conn.cursor() as cur:
            cur.execute("""
```

- ```
                 INSERT INTO Payments (payment_id, booking_id, user_id, amount, status)
  ```
- ```
                 VALUES (%s, %s, %s, %s, %s)
  ```
- ```
                 ON CONFLICT (payment_id) DO NOTHING
  ```
- ```
             """, (guid_data['guid'], guid_data['booking_id'], 42, 200.00, 'PENDING'))
  ```
- ```
             conn.commit()
  ```
- ```
     consumer.commit()
  ```
- **Outcome**: GUIDs assigned exactly once to payments/bookings (150ms).

12. Scalability and Concurrency

- **Horizontal Scaling**:
  - Deploy 100+ GUID Service nodes, each with a unique node ID (ZooKeeper).
  - Kafka partitions guid_assignments by service or event_id for 100K QPS.
- **Concurrency**:
  - Local sequence counters (Redis) avoid contention.
  - ZooKeeper ensures unique node IDs.
- **Traffic Spikes**:
  - Handle 100K QPS with load balancing and Redis caching.
  - Auto-scale GUID Service nodes via Kubernetes.

13. Performance Metrics

- **Latency**:
  - GUID generation: ~5ms (timestamp + Redis counter 1ms + processing 4ms).
  - Downstream assignment: ~150ms (Kafka transaction 30ms + PostgreSQL 120ms).
  - Recommendations: ~40ms (Faiss HNSW).
- **QPS**: 4.2M IDs/second (1024 nodes × 4096 IDs/ms), supports 100K QPS.
- **Storage**: 100MB for Redis counters, 6GB/year for PostgreSQL metadata.

- **Uniqueness**: Guaranteed by timestamp + node ID + counter.

14. Error Handling

- **Clock Drift**: Wait if timestamp decreases (rare, <1ms).
- **Node Failure**: ZooKeeper reassigns node ID after timeout.
- **Duplicate GUIDs**: Impossible due to unique node IDs and sequence counters.
- **Kafka Failures**: Transactional messaging ensures exactly-once delivery.

15. Security

- **Randomness**: Add 1-bit random field to prevent predictable IDs.
- **Authentication**: JWT for API access (optional).
- **Rate Limiting**: API Gateway limits 1000 requests/sec/client to prevent abuse.

16. Deployment

- **Cloud**: AWS (EC2, RDS PostgreSQL, ElastiCache Redis, Kafka, ZooKeeper, CloudFront).
- **Monitoring**:
  - Prometheus/Grafana for QPS, latency, and collision rates.
  - CloudWatch for logs, alerts for >10ms latency or node failures.
- **CI/CD**: Docker/Kubernetes for zero-downtime deployments.