Login system Arch with handle billion of request with queue but fast

High-Level Architecture (HLD) for Scalable Login System

We will design a **simple, scalable, and fast** login system using:

✅ **Go** → For backend API (lightweight & concurrent)

✅ **PostgreSQL** → For user data storage

✅ **Redis** → For caching sessions & rate limiting

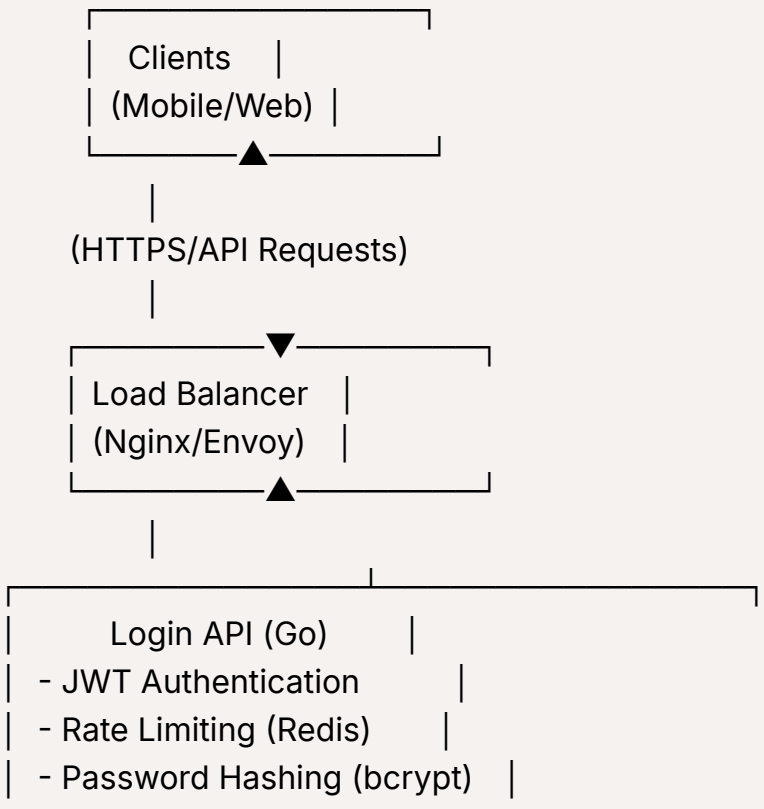
✅ **NATS** → For event-driven queue processing

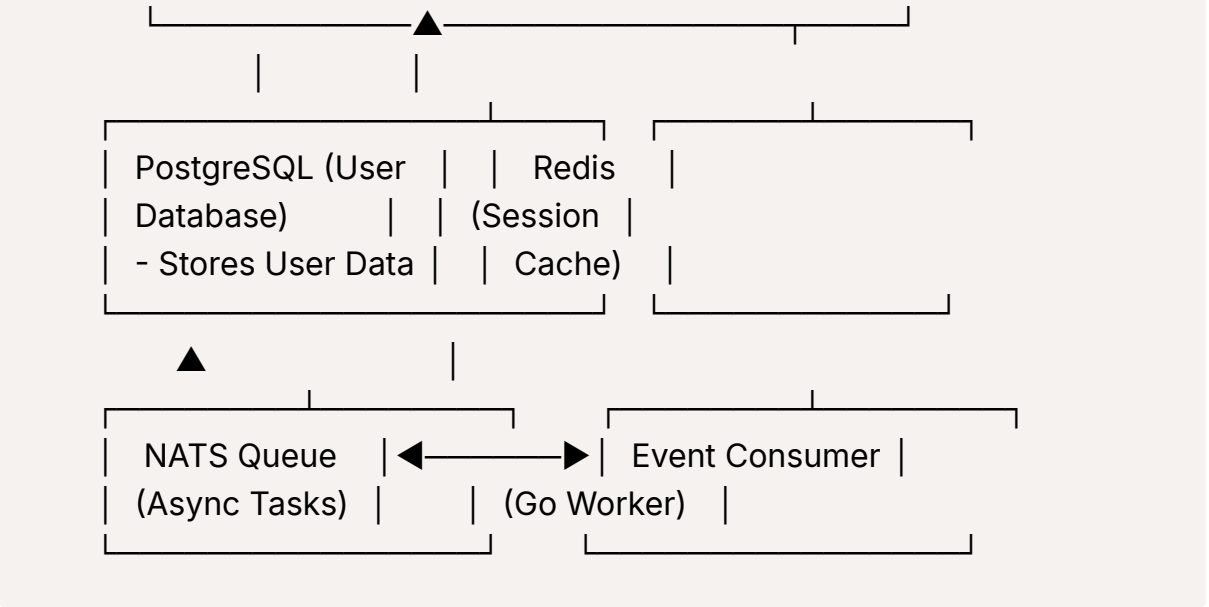
✅ **JWT** → For authentication & stateless session handling

✅ **Docker** → For containerization and easy deployment

---

1. Architecture Overview





2. Component Breakdown

A. Front Door System

- **We use Nginx or Envoy** to manage incoming traffic by:✅ Spreading user requests evenly across multiple servers✅ Handling secure HTTPS connections✅ Preventing too many requests from one user

B. Login API (Go)

- Handles user authentication:

  ✅ **Login with JWT** (stateless session management)

  ✅ **Password hashing** (bcrypt)

  ✅ **Rate limiting** (Redis-based)

  ✅ **Session caching** (Redis)

C. PostgreSQL (Database)

- Stores **user credentials** (hashed) and login history
- Uses **connection pooling** (pgx or pgbouncer)
- Indexing for **fast lookups**

D. Redis (Caching & Rate Limiting)

- Caches **JWT tokens** for quick validation

- Implements **IP-based rate limiting** to block abuse
- Temporary **failed login attempt tracking**

E. NATS (Message Queue)

- Handles **asynchronous event processing** (e.g., login logs, notifications, fraud detection)
- Events are processed by a **Go consumer service**

F. Docker & Deployment

- Each service runs inside **Docker containers**
- Use **Docker Compose** or **Kubernetes** for scaling

---

3. Login Flow

Step 1: User Login Request

1️⃣ Client (Web/Mobile) sends **email & password** to `/login` API

2️⃣ Go server **hashes password** & verifies it with **PostgreSQL**

3️⃣ If valid, **JWT token is generated** & stored in Redis

Step 2: Token Validation (Protected Routes)

1️⃣ Client sends **JWT token** in requests

2️⃣ API verifies token (locally or via Redis)

3️⃣ If valid, request proceeds

Step 3: Queue Processing (NATS)

1️⃣ On successful login, API **publishes an event** to NATS

2️⃣ A **Go worker service** consumes login events (e.g., analytics, fraud monitoring)

---

4. Scaling Strategy

- **Horizontally scale Go API** using Kubernetes (multiple containers)
- **Use read replicas for PostgreSQL** to handle high reads

- **Scale Redis** by clustering
- **NATS ensures async events don’t block login performance**

---

5. Why This Architecture?

✅ **Fast & Stateless:** JWT + Redis minimize DB lookups

✅ **Asynchronous Processing:** NATS queues prevent blocking

✅ **Horizontally Scalable:** Go’s concurrency + Docker + Kubernetes

✅ **Secure & Reliable:** Rate limiting, password hashing, and audit logging

---

LLD