**1. Functional Requirements**

**🗨️ Messaging System**

* Clients must be able to send and receive messages in real time.
* Messages must be routed correctly even if clients are connected to different chat servers.
* The system should support **message persistence** for history and offline delivery.

**👥 User Presence Management**

* The system must track user online/offline status.
* Clients should send **periodic heartbeats** to maintain online status.
* Presence changes must be broadcast to relevant users (e.g., active conversations or contacts).

**🔀 Message Routing**

* A **centralized message router** (or distributed routing strategy) must determine which server handles a user.
* Chat servers must query the router to deliver messages to the correct server or store them if offline.

**🕓 Sequence Management**

* Messages should be assigned a **unique ID** and/or **sequence number** for ordering, using strategies like:
  + Snowflake ID
  + Per-conversation sequencing
  + Logical clocks or Lamport timestamps

**💾 Message Storage**

* Store chat messages persistently using a **scalable NoSQL database** (e.g., HBase).
* Retrieve chat history efficiently using composite keys (e.g., user1\_user2).
* Support **offline message queuing and delivery**.

**🛡️ 2. Non-Functional Requirements**

**⚙️ Scalability**

* Must support **horizontal scaling** of chat servers behind a load balancer.
* Auto-scale chat server instances based on traffic/load.
* Should support millions of concurrent connections.

**⚡ Low Latency**

* Ensure near real-time delivery using **WebSockets or MQTT** for persistent, full-duplex communication.
* Optimize with proximity-based routing and edge deployments.

**🏃 High Availability**

* Redundancy at every layer (load balancers, chat servers, databases).
* Must tolerate partial failures without full system outages.
* Should gracefully handle node or region failures.

**🧠 Efficiency**

* Use efficient data serialization (e.g., Protocol Buffers).
* Use asynchronous I/O models (Node.js, Go, Python asyncio) for scalable connection handling.

**🏗️ 3. System/Infrastructure Requirements**

**☁️ Chat Servers**

* Stateless servers, enabling any instance to handle any client.
* Must support WebSocket upgrade from HTTP.
* Efficient connection and resource management (e.g., async I/O, pooling).

**⚖️ Load Balancing**

* Support **Layer 7** (application-level) load balancing for sticky sessions if needed.
* Must be compatible with WebSocket or MQTT persistent connections.
* Use appropriate algorithms: Round Robin, Least Connections, IP Hashing.

**🧭 Routing & Metadata Store**

* Redis or ZooKeeper to track:
  + Connected users and their chat server mappings
  + Online/offline presence states
* Must support fast read/write and high availability (e.g., Redis Cluster)

**📡 Presence Service**

* Lightweight service using Redis Pub/Sub for real-time status broadcasting.
* Should detect offline status based on heartbeat timeouts.

**📚 Database**

* Use HBase or similar for:
  + High-write throughput
  + Schema-less, column-family organization
* Support chat history queries by conversation ID or user pair.
* Archive older messages to cold storage if needed.

**🛠️ 4. Monitoring, Logging, and Alerting**

* Track connection health, message delivery status, and latency.
* Log chat events, message IDs, presence changes, etc.
* Setup alerts for server failures, high latency, heartbeat drop-offs.

**🔐 5. Optional/Advanced Requirements**

* **Keyword search** across chat history (may require secondary indexes).
* **Time-based filtering** for chat history (consider TSDB integration).
* **End-to-end encryption** for message content (out of scope unless specified).