Functional Requirements:

- **1-to-1 Chat:** Users can send and receive messages in a private chat.
- **Group Chat:** Users can create, join, and message in groups.
- **Media Sharing:** Support for sharing images, videos, and files with content deduplication.
- **Presence & Status:**
    - Show a user's **last seen** timestamp.
    - Indicate message status: **Sent** (✓), **Delivered** (✓✓), and **Read** (blue ✓✓).

Non-Functional Requirements:

- **Low Latency:** Real-time message delivery is critical.
- **High Availability:** The system must be highly fault-tolerant with no single point of failure.
- **Consistency:** Eventual consistency is acceptable for "last seen" status, but message delivery must be reliable (no lost messages). Strong consistency is not required for chat history across all devices instantly.
- **High Scalability:** The architecture must scale to handle hundreds of millions of daily active users.

---

2. Scale Estimation (Back-of-the-Envelope)

Let's assume a scale similar to WhatsApp to guide our design choices.

- **Daily Active Users (DAU):** 500 million
- **Messages per User per Day:** 40 (average)
- **Total Messages per Day:** 500M users * 40 messages = 20 Billion messages
- **Peak Write QPS (Queries Per Second):** Assuming traffic is 2x the average during peak hours:
    - 20B / (24 hours * 3600s) * 2 ≈ 500,000 messages/sec
- **Storage (Messages):** Assume an average message size of 200 bytes.
    - 20B messages/day * 200 bytes/message = 4 PB/day. Storing this for 5 years would be enormous. This confirms the original text's point: WhatsApp likely only stores messages until they are delivered, not permanently. We'll design for this "store-until-delivered" model.
- **Storage (Media):** Assume 10% of messages are media, with an average size of 300 KB.
    - 2B media messages/day * 300 KB = 600 TB/day. This is the dominant storage factor.

---

3. High-Level Architecture

We'll use a decoupled, microservices-based architecture optimized for real-time communication and massive scale.

Core Components:

1. **Load Balancer:** The entry point for all traffic. It terminates TLS and routes requests to the appropriate service.
2. **WebSocket Service (WSH):** A stateful service that maintains persistent WebSocket connections with online users. Each server can handle up to ~60k concurrent connections.
3. **Session Manager:** A highly available service (backed by **Redis**) that acts as a central registry. It maps each `user_id` to the specific `WebSocket Service` IP they are connected to.
4. **Message Service:** A stateless API that serves as the gateway for all chat-related logic (sending messages, status updates).
5. **Kafka:** A distributed message bus. It's the backbone of our system, used to decouple services and handle the massive write load of incoming messages and events.
6. **Fan-out Service:** A consumer that reads messages from Kafka, identifies the recipients (for 1-to-1 or group chats), and orchestrates the delivery.
7. **Presence Service:** A dedicated service that tracks user status (online, offline, last seen). It consumes a stream of user activity events.
8. **Message Store (Cassandra):** A NoSQL database optimized for high write throughput. It temporarily stores messages until they are confirmed as delivered.
9. **Asset Service & S3/Blob Storage:** Manages the upload/download of media files, using a globally distributed object store like S3.
10. **User/Group Service (MySQL + Redis):** Manages user profiles and group metadata with a relational database for data integrity, heavily cached in Redis for fast reads.

---

4. Deep Dive into Core Flows

Flow 1: User Connection and Presence

When a user opens the app, a persistent WebSocket connection is established.

1. The client connects to the **Load Balancer**, which forwards the request to a **WebSocket Service** instance.
2. The WebSocket Service authenticates the user and registers their connection with the **Session Manager**, storing the mapping: `user_id_1 -> ws_server_ip_1`.
3. The WebSocket Service also publishes a `user_online` event to a Kafka topic.
4. The **Presence Service** consumes this event and updates the user's status to "online" and refreshes their "last seen" timestamp in a fast key-value store (like Redis or Cassandra).

Flow 2: 1-to-1 Chat (Recipient Online)

1. User U1 sends a message to U2. The message is sent over the WebSocket to their connected **WebSocket Service (WSH1)**.
2. WSH1 forwards the message to the stateless **Message Service**.
3. The Message Service assigns a `message_id`, publishes the message to a Kafka topic (e.g., `incoming_messages`), and immediately returns an ACK (✓ Sent) to U1.
4. A **Fan-out Service** worker consumes the message from Kafka.
5. It queries the **Session Manager** to find U2's connection details. The Session Manager returns that U2 is connected to **WSH2**.
6. The Fan-out Service forwards the message directly to WSH2.
7. WSH2 pushes the message down the WebSocket to U2's client.
8. U2's client sends a "Delivered" ACK back to WSH2, which is published to another Kafka topic (`status_updates`).
9. The Fan-out Service consumes this status update and pushes the "Delivered" status (✓✓) to U1.

Flow 3: Group Chat

1. The initial flow is the same: U1 sends a message for Group G to the **Message Service**, which publishes it to a Kafka topic (e.g., `group_messages`).
2. A **Fan-out Service** worker consumes the message.
3. It calls the **Group Service** to get the list of all members in Group G.
4. For **each member** in the list, it follows the logic from Flow 2:
    - Query the **Session Manager** to check if the member is online.
    - If **online**, forward the message to their connected WebSocket Service.
    - If **offline**, store the message in the **Message Store (Cassandra)** marked as undelivered for that user and trigger a push notification.

Flow 4: Media & Asset Delivery

This process avoids sending large binary files through the real-time messaging channel.

1. U1's client first generates a hash (e.g., SHA-256) of the image and sends it to the **Asset Service**. The service checks if a file with that hash already exists to prevent duplicates.
2. If the hash is new, the Asset Service provides a secure, one-time upload URL.
3. U1's client uploads the encrypted image directly to **S3** via the Load Balancer.
4. Upon successful upload, S3 notifies the Asset Service, which stores the metadata (S3 URL, encryption keys, thumbnails) and returns an `asset_id`.
5. U1's client now sends a regular chat message containing this `asset_id` (and thumbnail) through the WebSocket flow.
6. When U2 receives this message, their client sees the `asset_id` and downloads the image directly from S3 (or a CDN in front of it).

---

5. Database and Schema Design

- **User/Group Service (MySQL):**
  - `users`: `user_id` (PK), `name`, `phone_number`, `profile_pic_url`
  - `groups`: `group_id` (PK), `name`, `group_icon_url`, `created_at`
  - `group_members`: (`group_id`, `user_id`) (PK), `role` (admin/member)
- **Message Store (Cassandra):** Designed for fast writes and time-series queries.
  - `messages`:
    - `chat_id` (Partition Key - could be `user_id` for 1-to-1 or `group_id` for group chat)
    - `message_id` (Clustering Key - time-based UUID for ordering)
    - `sender_id`, `content`, `asset_id`

---

6. Addressing Key Challenges

- **Race Condition (User Coming Online):** The original text identifies a valid race condition where a user comes online just as a message is being sent. Polling is inefficient. A better solution:
  1. When a user connects, their client asks the **Message Service** for all undelivered messages since its last sync time.
  2. This initial fetch ensures historical messages are retrieved.
  3. The live WebSocket connection handles all *new* messages from that point forward. This two-step process (historical fetch + live stream) closes the race condition gap.
- **Horizontal Scaling:** All components (except the stateful WebSocket Service) are stateless and can be scaled horizontally by adding more instances. The WebSocket Service can also be scaled, with the Session Manager keeping track of the distribution. Kafka, Cassandra, and Redis are all natively distributed and scalable.