WhatsApp Messenger System Design - High-Level Design (HLD)

Table of Contents

Problem Statement

Design a real-time messaging platform like WhatsApp that enables:

- One-on-one text messaging with delivery guarantees
- Group chat (up to 256 participants)
- Media sharing (images, videos, documents)
- Voice and video calls (out of scope for detailed design)
- Online/offline status indicators
- Read receipts (blue ticks)
- Message encryption (end-to-end)
- Cross-platform support (mobile, web, desktop)

Scale Requirements

- **2 billion daily active users (DAU)**
- **100 billion messages per day**
- **500 million active users simultaneously**
- **4.5 billion group messages per day**
- **Average message size: 100 bytes**
- **Latency requirement: < 100ms for message delivery**
- **99.99% availability**

Functional Requirements

Must Have (P0)

1. User Management

- User registration with phone number
- Profile management (name, status, profile picture)
- Contact list synchronization
- Block/unblock users

2. Messaging

- Send text messages (1:1)
- Receive messages in real-time
- Message delivery status:
  - Sent (single tick) ✓
  - Delivered (double tick) ✓✓
  - Read (blue tick) ✓✓ (blue)
- Message history (retrieve old messages)
- Delete messages (delete for me, delete for everyone)
- Forward messages
- Reply to specific messages (quoted)

3. Group Chat

- Create groups (up to 256 members)
- Add/remove participants
- Admin roles (only admins can change settings)
- Group name, description, icon
- See participant list
- Group delivery receipts (show who read)
- Leave group

4. Media Sharing

- Share images (JPEG, PNG, max 16MB)
- Share videos (MP4, max 16MB for sending)
- Share documents (PDF, DOC, max 100MB)
- Share voice messages
- Share location
- Share contacts

5. Status & Presence

- Online/offline status
- Last seen timestamp
- Typing indicators
- "Recording audio" indicator

6. Calls (Basic mention)

- Voice calls (1:1)
- Video calls (1:1, group)
- Call history

Nice to Have (P1)

- Stories/Status updates (24-hour expiry)
- Message reactions (emoji)
- Voice messages
- Live location sharing
- Disappearing messages
- Archived chats
- Starred messages
- Message search
- Two-factor authentication
- Multi-device support

Non-Functional Requirements

Performance

- **Message delivery latency**: < 100ms (p95)
- **Message send latency**: < 50ms
- **Connection establishment**: < 1 second
- **Image upload**: < 3 seconds (5MB image)
- **Conversation load**: < 200ms

Scalability

- Support 2 billion DAU
- Handle 100 billion messages/day (~1.15M messages/second)
- Support 500M concurrent connections
- Handle message spikes during peak hours (New Year, events)
- Scale to 10x current load

Availability

- **99.99% uptime** (< 53 minutes downtime/year)
- Multi-region deployment
- No message loss (durability guarantee)
- Offline message queueing

Reliability

- **Message delivery guarantee**: At-least-once delivery
- **Message ordering**: Maintain order per conversation
- **No duplicate messages**: Idempotent delivery

- **Persistent storage**: Messages stored forever (unless deleted)

Security

- **End-to-end encryption** (E2EE) for all messages
- **Perfect forward secrecy** (compromised key doesn't decrypt past messages)
- **Server-side encryption** at rest
- **TLS** for all connections
- **2FA** for account access

Consistency

- **Eventual consistency** for last seen, online status
- **Strong consistency** for messages (no message loss)
- **Causal ordering** for conversation messages

Capacity Estimation

Traffic Estimates

```
Daily Active Users (DAU): 2B
Messages per user per day: 50
Total messages/day: 2B × 50 = 100B

Messages/second (average): 100B / 86,400 ≈ 1.15M QPS
Messages/second (peak - 3x): 3.5M QPS

Group messages/day: 4.5B (10% have media)
1:1 messages/day: 95.5B

Concurrent WebSocket connections: 500M
```

Storage Estimates

Text Messages:

```
Average message size: 100 bytes (text + metadata)
Daily text messages: 100B
Daily storage: 100B × 100 bytes = 10 TB/day

Yearly storage: 10 TB × 365 = 3.65 PB/year
Storage for 5 years: 18.25 PB (text only)
```

Media Messages:

```
Media messages/day: 10B (10% of total)
Average sizes:
- Images: 500 KB
- Videos: 5 MB
- Voice messages: 50 KB
- Documents: 2 MB

Assumption: 60% images, 20% videos, 15% voice, 5% docs
Weighted average: (0.6 × 500KB) + (0.2 × 5MB) + (0.15 × 50KB) + (0.05 ×
2MB)
                = 300KB + 1MB + 7.5KB + 100KB ≈ 1.4 MB per media message

Daily media storage: 10B × 1.4 MB = 14 PB/day
Yearly: 14 PB × 365 = 5,110 PB = 5.1 EB/year

With compression (50% reduction): 2.55 EB/year
```

Total Storage:

```
Year 1: 2.6 EB (2.55 media + 0.05 text)
Year 5: 13 EB

Note: WhatsApp stores messages temporarily server-side
Permanent storage is on client devices
Server retention: 30 days for undelivered messages
```

Bandwidth Estimates

```
Incoming (message sends):
10 TB text + 14 PB media = 14.01 PB/day
14.01 PB / 86,400 seconds ≈ 162 GB/second

Outgoing (message deliveries - 1:1 each message):
Similar to incoming: ~162 GB/second

Peak (3x average): ~486 GB/second
```

Memory Requirements (Caching)

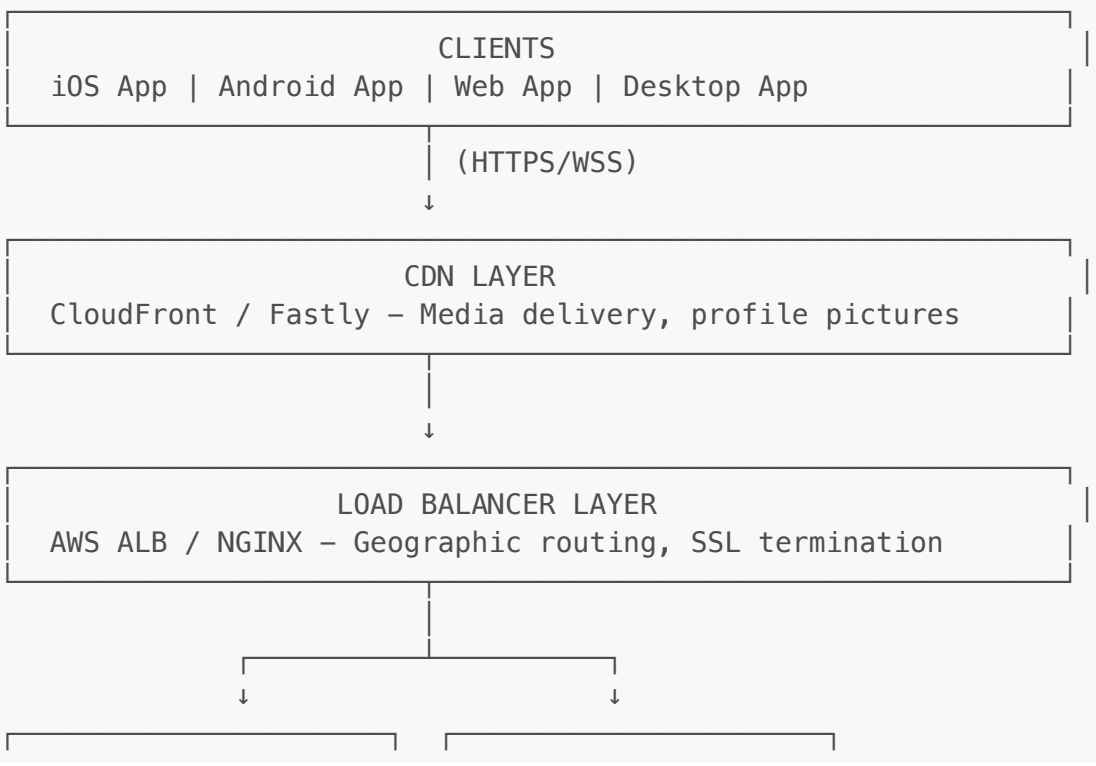
```
Online user presence:
500M concurrent × 100 bytes = 50 GB

Recent conversations (per user, last 50 messages):
500M users × 50 messages × 100 bytes = 2.5 TB
```

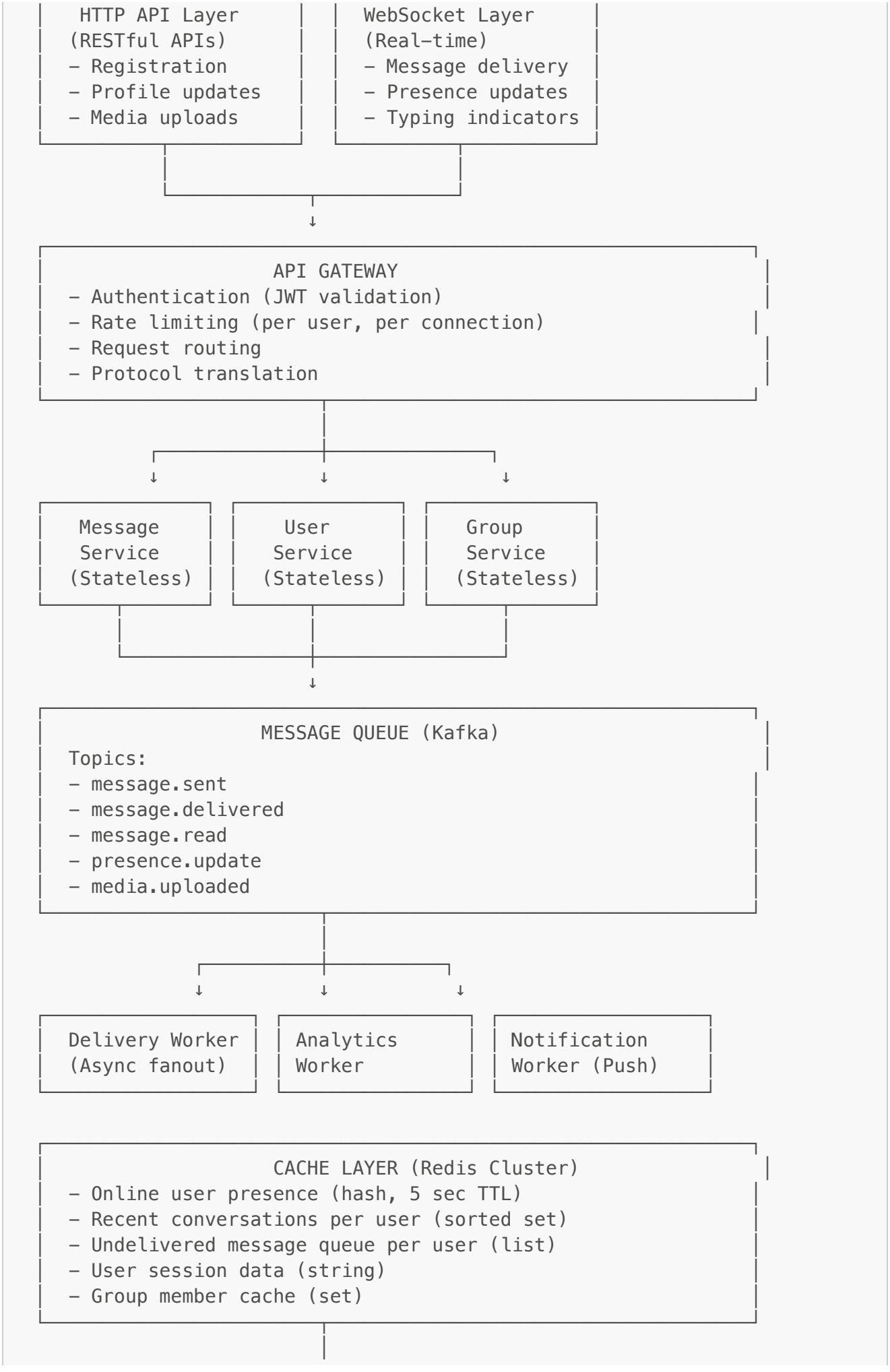
```
Undelivered messages queue (24 hour buffer):
Assume 10% users offline at any time
200M offline users × 100 messages × 100 bytes = 2 TB

Connection metadata:
500M connections × 500 bytes = 250 GB

Total memory needed: ~5 TB
Distributed across 200+ Redis nodes = 25 GB per node
```

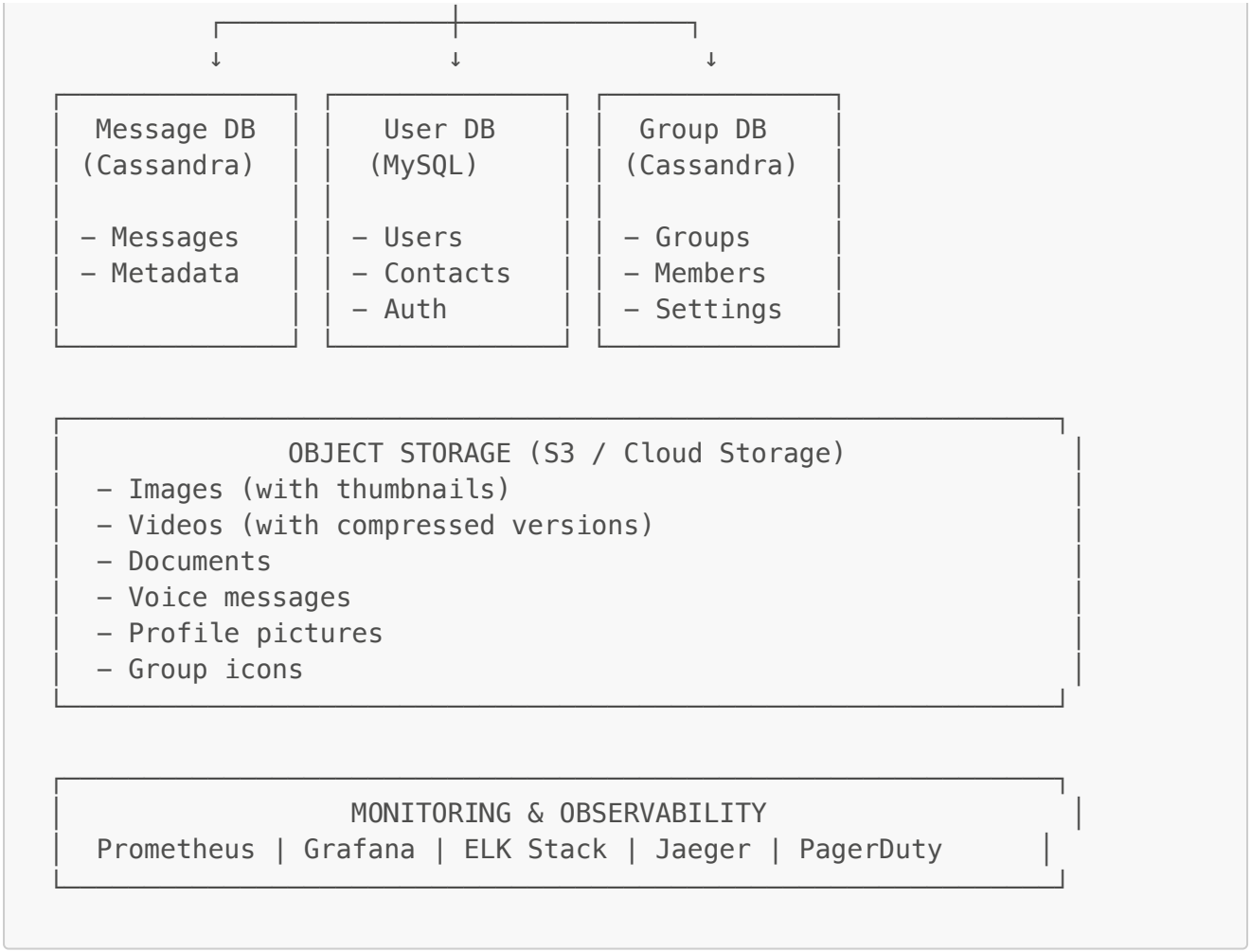
Server Estimates

```
WebSocket connections per server: 65K (C10K problem limit)
Servers needed for 500M connections: 500M / 65K ≈ 8,000 servers

With redundancy and headroom (2x): 16,000 servers
Organized into clusters by region

Cost estimation (AWS):
- WebSocket servers: 16K × $100/month = $1.6M/month
- Database servers: $500K/month
- Storage (S3): 13 EB × $0.023/GB = $300K/month
- Bandwidth: $200K/month
- Total: ~$2.6M/month
```

High-Level Architecture




HTTP API Layer
(RESTful APIs)
- Registration
- Profile updates
- Media uploads
WebSocket Layer
(Real-time)
- Message delivery
- Presence updates
- Typing indicators
API GATEWAY
- Authentication (JWT validation)
- Rate limiting (per user, per connection)
- Request routing
- Protocol translation
Message Service (Stateless)
User Service (Stateless)
Group Service (Stateless)
MESSAGE QUEUE (Kafka)
Topics:
- message.sent
- message.delivered
- message.read
- presence.update
- media.uploaded
Delivery Worker (Async fanout)
Analytics Worker
Notification Worker (Push)
CACHE LAYER (Redis Cluster)
- Online user presence (hash, 5 sec TTL)
- Recent conversations per user (sorted set)
- Undelivered message queue per user (list)
- User session data (string)
- Group member cache (set)



Key Architectural Decisions

1. **WebSocket for Real-time Communication**
    - Persistent bidirectional connection
    - Low latency (< 100ms)
    - Push capability (server → client)
2. **Cassandra for Message Storage**
    - Write-heavy workload (1.15M messages/sec)
    - Time-series data (sorted by timestamp)
    - Linear scalability
    - No single point of failure
3. **Redis for Presence & Session**
    - In-memory speed for online status
    - Sub-millisecond lookups
    - TTL support for automatic cleanup
    - Pub/Sub for real-time updates
4. **Kafka for Message Pipeline**
    - High throughput (1M+ messages/sec)

- Durability (no message loss)
- Replay capability (disaster recovery)
- Decouples producers and consumers

5. **S3 for Media Storage**
    - Virtually unlimited storage
    - 99.999999999% durability (11 nines)
    - Integrated with CDN
    - Lifecycle policies (auto-archival)

Core Components

1. WebSocket Gateway

Purpose: Maintain persistent connections with clients

Responsibilities:

- Establish WebSocket connections
- Handle heartbeats (keep-alive every 30 seconds)
- Route messages to/from clients
- Maintain connection metadata (user_id, device_id, connection_id)
- Handle reconnection logic
- Graceful degradation to HTTP long-polling

Architecture:

```
┌─────────────────────────────────────────┐
│     WebSocket Gateway (Stateful)        │
│                                         │
│  Connection Manager:                    │
│  - Active connections: Map<user_id,     │
│    List<connection>>                    │
│  - Connection state: CONNECTING,        │
│    CONNECTED, DISCONNECTED              │
│  - Last heartbeat timestamp             │
│                                         │
│  Message Router:                        │
│  - Route incoming → Message Service     │
│  - Route outgoing → Specific connection │
│                                         │
│  Heartbeat Handler:                     │
│  - Ping every 30 seconds                │
│  - Timeout after 60 seconds             │
│  - Clean up dead connections            │
└─────────────────────────────────────────┘
```

Connection Distribution:

```
User connects → Load Balancer → Available WebSocket Gateway

Gateway Selection:
- Geographic proximity (latency)
- Current connection count (load balancing)
- Server health status

Connection sticky: Yes (same user → same gateway if possible)
Benefit: Reduced state synchronization
```

Scaling:

- Each server: 65,000 concurrent connections (C10K problem solved)
- Total servers: 500M / 65K ≈ 8,000 servers
- Organized in clusters per region

State Management:

```
Problem: WebSocket is stateful, but we need to route messages

Solution: Redis-based session store
  Key: "session:{user_id}"
  Value: {
    "gateway_server": "ws-gateway-42.us-east",
    "connection_id": "conn_abc123",
    "device_id": "device_xyz789",
    "connected_at": 1704672000,
    "last_heartbeat": 1704673000
  }
  TTL: 70 seconds (auto-cleanup if connection dies)
```

2. Message Service

Purpose: Core business logic for message handling

Responsibilities:

- Validate message content (length, format)
- Generate unique message IDs (Snowflake algorithm)
- Store messages in Cassandra
- Publish to Kafka for async processing
- Handle message deletion
- Manage message metadata

Message ID Generation (Snowflake):

```
64-bit Message ID structure:
┌──────────┬──────────────┬──────────────┬──────────────┐
│ 1 bit    │ 41 bits      │ 10 bits      │ 12 bits      │
│ unused   │ timestamp    │ server       │ sequence     │
│          │ (ms)         │ ID           │ (4096/ms)    │
└──────────┴──────────────┴──────────────┴──────────────┘

Benefits:
- Globally unique
- Sortable by time
- No coordination needed
- 4096 IDs per millisecond per server
```

Message Processing Pipeline:

```
1. Receive message from sender
2. Validate (length, format, permissions)
3. Generate message_id
4. Check if recipient online (Redis presence)
5. If online:
   - Push to recipient's gateway (via Redis Pub/Sub)
   - Update delivery status → SENT
6. If offline:
   - Queue in Redis undelivered messages
   - Send push notification
7. Store in Cassandra (async via Kafka)
8. Return ACK to sender with message_id
```

Total Latency Breakdown:

```
Validation: 1ms
Generate ID: 0.1ms
Presence check (Redis): 1ms
Push to recipient: 10-50ms (WebSocket)
Store in Cassandra (async): 0ms (non-blocking)
ACK to sender: 2ms
─────────────────────────────
Total: ~15-55ms (well under 100ms target)
```

3. User Service

Purpose: Manage user accounts and profiles

Responsibilities:

- User registration (phone number verification)

- Authentication (JWT tokens)
- Profile management (name, status, picture)
- Contact list synchronization
- Block/unblock management
- Privacy settings

Registration Flow:

```
1. User enters phone number (+1234567890)
2. System sends 6-digit OTP via SMS (Twilio/SNS)
3. User enters OTP
4. System validates OTP (5-minute expiry)
5. Create user account:
   - Generate user_id (UUID)
   - Hash phone number for privacy
   - Create initial profile
6. Generate JWT tokens (access + refresh)
7. Return to client with tokens
```

Contact Sync:

```
Problem: User has 1000 contacts, need to check who's on WhatsApp

Naive approach: Query DB for each contact (1000 queries) ❌

Optimized approach (Bloom Filter):
1. Client sends hash of all phone numbers
2. Server checks against Bloom filter (probabilistic)
3. Possible matches: Query database (100 queries)
4. Return list of users on WhatsApp
5. Cache result on client (refresh daily)

Benefits:
- Reduces queries by 90%
- Fast membership testing
- Privacy-preserving (hashes, not plain numbers)
```

4. Group Service

Purpose: Manage group chats and memberships

Group Creation:

```
1. User creates group → Group Service
2. Generate group_id (UUID)
3. Add creator as admin
4. Invite initial members (send invites)
```

```
5. Store group metadata in Cassandra:
   - group_id, name, description, icon_url
   - created_at, created_by, admins[]
6. Create group_members table entry for each member
7. Return group_id to creator
```

Group Message Fanout:

```
Challenge: Send message to 256 group members efficiently

Solution: Async fanout via Kafka

Flow:
1. User sends message to group
2. Store message once in Cassandra
3. Publish to Kafka: "group.message.sent"
4. Fanout worker:
   - Fetch all group members (cached)
   - For each member:
     a. Check if online (Redis)
     b. If online: Push to WebSocket
     c. If offline: Queue message + push notification
   - Update delivery receipts
5. Total time: ~100-200ms for 256 members
```

Group Metadata Caching:

```
Cache group members (Redis):
  Key: "group:{group_id}:members"
  Type: Set
  Members: {user_id_1, user_id_2, ...}
  TTL: 1 hour

Benefits:
- Fast member lookup (O(1))
- Avoid DB query for every message
- Invalidate on member add/remove
```

5. Presence Service

Purpose: Track online/offline status of users

Online Status Flow:

```
User opens app:
  1. Establish WebSocket connection
```

```
  2. Send "presence.online" event
  3. Update Redis: SET user:{user_id}:online 1 EX 70
  4. Broadcast to user's contacts (Pub/Sub)

User closes app / disconnects:
  1. Connection drops (detected by gateway)
  2. Update Redis: DEL user:{user_id}:online
  3. Set last_seen: SET user:{user_id}:last_seen {timestamp}
  4. Broadcast offline status

Heartbeat (every 30 seconds):
  1. Client sends ping
  2. Server updates TTL: EXPIRE user:{user_id}:online 70
  3. Server sends pong
```

Last Seen Timestamp:

```
Storage: Redis + MySQL

Redis (hot data):
  Key: "user:{user_id}:last_seen"
  Value: Unix timestamp
  TTL: 7 days

MySQL (permanent):
  UPDATE users SET last_seen_at = NOW() WHERE user_id = ?
  (Batched updates every 5 minutes to reduce DB load)
```

Privacy Controls:

```
User settings:
- Everyone: Show last seen to all
- My contacts: Show only to contacts
- Nobody: Don't show last seen

Implementation:
  Before returning last_seen, check:
    1. Requester's relationship with user
    2. User's privacy setting
    3. Return last_seen or "hidden" accordingly
```

6. Media Service

Purpose: Handle media uploads and processing

Upload Flow:

```
1. Client requests upload URL:
   POST /api/v1/media/upload/init
   Body: {"type": "image", "size": 5242880}

2. Server validates:
   - Size within limits
   - User has storage quota
   - Rate limit not exceeded

3. Generate pre-signed S3 URL:
   URL expires in 15 minutes
   Unique key: media/{user_id}/{timestamp}/{uuid}.jpg

4. Return upload URL to client

5. Client uploads directly to S3 (parallel, chunked)

6. Client confirms upload:
   POST /api/v1/media/upload/complete
   Body: {"media_id": "abc123", "s3_key": "media/..."}

7. Async processing (Lambda / Worker):
   - Generate thumbnails (150px, 400px)
   - Compress original (WebP for images, H.264 for video)
   - Extract metadata (dimensions, duration)
   - Virus scan
   - Store processed versions

8. Update media metadata in database

9. Media now ready to attach to messages
```

Media Storage Organization:

```
S3 Bucket Structure:
  whatsapp-media-{region}/
    ├── images/
    │   ├── original/{user_id}/{yyyy-mm}/{uuid}.jpg
    │   ├── large/{user_id}/{yyyy-mm}/{uuid}_800.webp
    │   └── thumb/{user_id}/{yyyy-mm}/{uuid}_150.webp
    ├── videos/
    │   ├── original/{user_id}/{yyyy-mm}/{uuid}.mp4
    │   └── compressed/{user_id}/{yyyy-mm}/{uuid}_720p.mp4
    ├── documents/
    │   └── {user_id}/{yyyy-mm}/{uuid}.{ext}
    └── voice/
        └── {user_id}/{yyyy-mm}/{uuid}.opus

Lifecycle Policy:
- Hot tier (SSD): 0-30 days
```

```
- Standard tier: 31-90 days
- Glacier: > 90 days (if user doesn't access)
```

CDN Integration:

```
CloudFront in front of S3:
  - Cache popular media (24 hour TTL)
  - Serve from nearest edge location
  - ~90% cache hit ratio

Origin: S3 bucket
Edge Locations: 400+ globally
Latency: ~20ms (edge) vs 200ms (origin)
```

7. Notification Service

Purpose: Send push notifications when user offline

Notification Types:

```
1. New message (priority: HIGH)
2. Missed call (priority: HIGH)
3. Group mention (priority: MEDIUM)
4. Group message (priority: LOW if muted)
```

Push Notification Flow:

```
1. Message arrives, user offline
2. Check user's device tokens (cached in Redis)
3. Build notification payload:
   {
     "title": "Alice",
     "body": "Hey, how are you?",
     "message_id": "msg_123",
     "conversation_id": "conv_456",
     "priority": "HIGH",
     "badge_count": 5  // Unread messages
   }
4. Send to push service:
   - iOS → APNs (Apple Push Notification Service)
   - Android → FCM (Firebase Cloud Messaging)
5. Handle delivery status:
   - Success: Log
   - Failure (token expired): Remove from cache
   - Retry: 3 attempts with exponential backoff
```

Optimization: Batching:

```
Don't send notification for every message in group chat

Strategy:
- Buffer messages for 3 seconds
- If multiple messages in group: Send one notification
  "3 new messages in Family Group"
- Prevents notification spam
```

8. Delivery Worker

Purpose: Ensure message reaches recipient(s)

Delivery Guarantee: At-least-once delivery

Flow for 1:1 Message:

```
1. Message stored in Cassandra
2. Check recipient online status (Redis)
3. If online:
   - Get recipient's WebSocket gateway from Redis
   - Push message directly via that gateway
   - Wait for ACK from recipient
   - Update status: DELIVERED
4. If offline:
   - Store in Redis undelivered queue
   - Send push notification
   - Status remains: SENT
5. When recipient comes online:
   - Retrieve undelivered messages from queue
   - Deliver in order
   - Update status: DELIVERED
6. When recipient opens chat:
   - Send READ receipt
   - Update status: READ
```

Flow for Group Message:

```
1. Message stored once in Cassandra
2. Publish to Kafka: "group.message.new"
3. Fanout worker consumes event:
   - Fetch all 256 group members (cached)
   - For each member (parallel processing):
     a. Skip sender
     b. Check online status
     c. If online: Push via WebSocket
```

```
     d. If offline: Queue + push notification
   - Track delivery: {member_1: DELIVERED, member_2: SENT, ...}
4. Update group message status
5. Show sender: "Delivered to 200/255"
```

Idempotency:

```
Problem: Network issues may cause duplicate deliveries

Solution: Message ID + De-duplication

Client side:
  - Track received message IDs (last 1000)
  - Ignore duplicates based on message_id
  - Send ACK for both original and duplicate

Server side:
  - Use message_id as idempotency key
  - Cassandra: PRIMARY KEY handles duplicates
  - Redis: SETNX for atomic operations
```

9. Analytics Service

Purpose: Track usage metrics and generate insights

Metrics Collected:

```
User Metrics:
- DAU, WAU, MAU
- Message sent per user
- Active time per session
- Retention rate (D1, D7, D30)

Message Metrics:
- Messages per second
- Average message length
- Media vs text ratio
- Group vs 1:1 ratio

Performance Metrics:
- Message delivery latency (p50, p95, p99)
- Connection success rate
- Heartbeat response time
- Error rates by type
```

Data Pipeline:

```
[User Actions] → [Kafka] → [Apache Flink]
                                ↓
                        [Real-time Aggregation]
                                ↓
                          [Time-series DB]
                          (InfluxDB/TimescaleDB)
                                ↓
                          [Dashboards]
                          (Grafana/Kibana)
```

Database Design

1. Message Database (Cassandra)

Why Cassandra?

- Write-optimized (1.15M writes/second)
- Time-series data (messages sorted by time)
- Linear scalability (add nodes for more throughput)
- Multi-datacenter replication
- No single point of failure
- Tunable consistency

Schema Design:

Messages Table (1:1 conversations):

```
CREATE TABLE messages (
    conversation_id uuid,
    message_id timeuuid,
    sender_id uuid,
    receiver_id uuid,
    content text,
    message_type text,  -- TEXT, IMAGE, VIDEO, DOCUMENT, VOICE
    media_url text,
    created_at timestamp,
    status text,  -- SENT, DELIVERED, READ
    is_deleted boolean DEFAULT false,
    PRIMARY KEY ((conversation_id), message_id)
) WITH CLUSTERING ORDER BY (message_id DESC);

-- Index for user's all conversations
CREATE TABLE user_messages (
    user_id uuid,
    conversation_id uuid,
    message_id timeuuid,
    created_at timestamp,
```

```
    PRIMARY KEY ((user_id), created_at, message_id)
) WITH CLUSTERING ORDER BY (created_at DESC);
```

Group Messages Table:

```
CREATE TABLE group_messages (
    group_id uuid,
    message_id timeuuid,
    sender_id uuid,
    content text,
    message_type text,
    media_url text,
    created_at timestamp,
    PRIMARY KEY ((group_id), message_id)
) WITH CLUSTERING ORDER BY (message_id DESC);

-- Delivery receipts for group messages
CREATE TABLE group_message_receipts (
    group_id uuid,
    message_id timeuuid,
    user_id uuid,
    status text,  -- DELIVERED, READ
    timestamp timestamp,
    PRIMARY KEY ((group_id, message_id), user_id)
);
```

Partitioning Strategy:

```
Partition by conversation_id for 1:1 messages
Partition by group_id for group messages

Benefits:
- All messages in conversation co-located
- Efficient range queries (get last N messages)
- Natural hot-spot distribution

Replication Factor: 3 (across datacenters)
Consistency Level: QUORUM (2 of 3 must acknowledge)
```

2. User Database (MySQL)

Why MySQL?

- User data requires ACID transactions
- Complex queries (contact matching)
- Relatively small dataset (2B users × 2KB ≈ 4TB)
- Strong consistency for user profiles

Schema:

```
CREATE TABLE users (
    user_id CHAR(36) PRIMARY KEY,
    phone_number_hash CHAR(64) UNIQUE NOT NULL,  -- SHA-256 hash
    username VARCHAR(50),
    display_name VARCHAR(100),
    status_message VARCHAR(139),  -- Like Twitter
    profile_picture_url VARCHAR(500),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_seen_at TIMESTAMP,
    is_online BOOLEAN DEFAULT FALSE,
    privacy_last_seen ENUM('everyone', 'contacts', 'nobody') DEFAULT
'everyone',
    privacy_profile_pic ENUM('everyone', 'contacts', 'nobody') DEFAULT
'everyone',
    privacy_status ENUM('everyone', 'contacts', 'nobody') DEFAULT
'everyone',
    INDEX idx_phone_hash (phone_number_hash),
    INDEX idx_last_seen (last_seen_at)
);

CREATE TABLE contacts (
    user_id CHAR(36),
    contact_user_id CHAR(36),
    contact_name VARCHAR(100),  -- User's local name for contact
    added_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, contact_user_id),
    INDEX idx_contact (contact_user_id)
);

CREATE TABLE blocked_users (
    user_id CHAR(36),
    blocked_user_id CHAR(36),
    blocked_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, blocked_user_id)
);

CREATE TABLE device_tokens (
    user_id CHAR(36),
    device_id VARCHAR(100),
    platform ENUM('ios', 'android', 'web'),
    push_token VARCHAR(500),  -- APNs/FCM token
    last_active_at TIMESTAMP,
    PRIMARY KEY (user_id, device_id)
);
```

Sharding Strategy:

```
Shard by user_id using consistent hashing

Number of shards: 100 (can scale to 1000)
Shard calculation: hash(user_id) % 100

Example:
  user_id: "550e8400-e29b-41d4-a716-446655440000"
  hash: 12345
  shard: 12345 % 100 = 45
  → Database shard 45
```

3. Group Database (Cassandra)

Schema:

```
CREATE TABLE groups (
    group_id uuid PRIMARY KEY,
    name text,
    description text,
    icon_url text,
    created_by uuid,
    created_at timestamp,
    max_members int DEFAULT 256,
    member_count int
);

CREATE TABLE group_members (
    group_id uuid,
    user_id uuid,
    role text,  -- ADMIN, MEMBER
    joined_at timestamp,
    added_by uuid,
    PRIMARY KEY ((group_id), user_id)
);

-- Index for user's groups
CREATE TABLE user_groups (
    user_id uuid,
    group_id uuid,
    joined_at timestamp,
    PRIMARY KEY ((user_id), joined_at, group_id)
) WITH CLUSTERING ORDER BY (joined_at DESC);
```

4. Conversation Metadata (Redis)

Purpose: Fast access to conversation data

Data Structures:

```
Recent Conversations (per user):
  Key: "conversations:{user_id}"
  Type: Sorted Set (ZSET)
  Score: Last message timestamp
  Value: conversation_id

  ZADD conversations:user123 1704672000 conv-abc
  ZREVRANGE conversations:user123 0 49  # Last 50 conversations
  TTL: 30 days

Unread Count:
  Key: "unread:{user_id}:{conversation_id}"
  Type: Integer
  Value: Count of unread messages

  INCR unread:user123:conv-abc
  GET unread:user123:conv-abc  → 5
  TTL: 30 days

Typing Indicator:
  Key: "typing:{conversation_id}:{user_id}"
  Type: String
  Value: 1 (is typing)

  SETEX typing:conv-abc:user123 5 1  # 5 second TTL
  GET typing:conv-abc:user123
```

Message Delivery

End-to-End Message Flow

Scenario: Alice sends "Hello" to Bob

```
┌─────────────────────────────────────────────────────────────┐
│ PHASE 1: Message Send (Alice's side)                        │
└─────────────────────────────────────────────────────────────┘

T=0ms: Alice types "Hello" and hits send

T=1ms: Alice's app encrypts message (E2EE)
  - Generate ephemeral key pair
  - Encrypt content with Bob's public key
  - Add digital signature

T=5ms: Send via WebSocket to gateway
  Message payload: {
    "message_id": "temp_client_id_xyz",
    "receiver_id": "bob_uuid",
```

```
    "encrypted_content": "base64_encrypted_data",
    "message_type": "TEXT",
    "timestamp": 1704672000000
  }

T=10ms: Gateway forwards to Message Service

┌─────────────────────────────────────────────────────────────────┐
│ PHASE 2: Server Processing                                      │
└─────────────────────────────────────────────────────────────────┘

T=11ms: Message Service processing
  - Validate message format
  - Check Alice not blocked by Bob
  - Generate server message_id (Snowflake)
  - Replace temp_client_id with server message_id

T=12ms: Check Bob's online status (Redis)
  Result: Bob is ONLINE

T=15ms: Lookup Bob's WebSocket gateway (Redis)
  Result: ws-gateway-15.us-west

T=18ms: Publish to Redis Pub/Sub channel
  Channel: "messages:ws-gateway-15"
  Message: {message_id, receiver_id: bob, ...}

T=20ms: Send ACK to Alice
  ACK: {
    "temp_id": "temp_client_id_xyz",
    "message_id": "msg_server_id_abc",
    "status": "SENT" ✓
  }

Alice's app: Show single tick ✓

T=22ms: Async Kafka publish (non-blocking)
  Topic: "message.created"
  Purpose: Persistence, analytics, audit

┌─────────────────────────────────────────────────────────────────┐
│ PHASE 3: Delivery to Bob                                        │
└─────────────────────────────────────────────────────────────────┘

T=23ms: ws-gateway-15 receives message (Redis Pub/Sub)

T=25ms: Gateway pushes to Bob's WebSocket connection

T=30ms: Bob's app receives encrypted message

T=32ms: Bob's app decrypts message
  - Use Bob's private key
  - Verify Alice's signature
```

```
  - Decrypt content → "Hello"

T=35ms: Bob's app sends DELIVERED ACK to gateway

T=37ms: Gateway forwards ACK to Message Service

T=40ms: Message Service updates status:
  - Update Redis: message:msg_abc:status = DELIVERED
  - Publish to Kafka: "message.delivered"

T=45ms: Gateway pushes status update to Alice

Alice's app: Show double tick ✓✓

T=50ms: Bob opens chat, reads message

T=52ms: Bob's app sends READ receipt

T=55ms: Message Service updates status: READ

T=60ms: Alice receives READ receipt

Alice's app: Ticks turn blue ✓✓ (blue)

┌─────────────────────────────────────────────────────────────┐
│ PHASE 4: Persistence (Async, parallel with above)           │
└─────────────────────────────────────────────────────────────┘

T=25ms: Kafka consumer picks up "message.created" event

T=30ms: Write to Cassandra:
  INSERT INTO messages (
    conversation_id, message_id, sender_id,
    receiver_id, content, created_at, status
  ) VALUES (...);

T=50ms: Cassandra write complete (3 replicas)

T=100ms: All async processes complete

TOTAL END-TO-END LATENCY: 60ms (Alice sees blue ticks)
```

Offline Message Handling

```
Scenario: Alice sends message, Bob is offline

Flow:
1. Alice sends message (T=0)
2. Server checks Bob status: OFFLINE
3. Server actions:
```

```
   a. Store in Cassandra
   b. Queue in Redis: LPUSH undelivered:bob msg_id
   c. Send push notification to Bob
   d. Return SENT status to Alice (single tick ✓)

4. Bob comes online (T=3600s, 1 hour later)
5. Connection established
6. Server retrieves undelivered messages:
   messages = LRANGE undelivered:bob 0 -1
   → [msg_abc, msg_def, msg_xyz]

7. Deliver messages in order:
   For each message:
     - Fetch from Cassandra
     - Push to Bob's WebSocket
     - Wait for ACK
     - Update status: DELIVERED
     - Remove from queue: LREM undelivered:bob

8. Alice receives DELIVERED updates (ticks turn double ✓✓)

9. Bob opens chat with Alice
10. Bob's app sends READ receipts
11. Alice's ticks turn blue ✓✓
```

Message Ordering Guarantee

```
Problem: Messages may arrive out of order due to:
- Network delays
- Retries
- Multiple paths

Solution: Sequence numbers + Client-side reordering

Server side:
  - Use timeuuid for message_id (sortable by time)
  - Messages naturally ordered in Cassandra

Client side:
  - Buffer messages temporarily
  - Sort by message_id (time-sortable)
  - Display in correct order
  - Handle gaps (missing sequence numbers)

Example:
  Receive: msg_5, msg_3, msg_7, msg_4
  Buffer and sort: msg_3, msg_4, msg_5, msg_7
  Gap detected: msg_6 missing
  Request msg_6 from server
  Display complete sequence
```

Deep Dives

1. WebSocket Connection Management

Connection Lifecycle:

```
┌─────────────────────────────────────────────┐
│ 1. CONNECTING                               │
│    - TCP handshake                          │
│    - TLS negotiation                        │
│    - WebSocket upgrade                      │
│    Duration: ~200-500ms                     │
└─────────────────────────────────────────────┘
         ↓
┌─────────────────────────────────────────────┐
│ 2. AUTHENTICATING                           │
│    - Send JWT token                         │
│    - Server validates                       │
│    - Load user session                      │
│    Duration: ~100ms                         │
└─────────────────────────────────────────────┘
         ↓
┌─────────────────────────────────────────────┐
│ 3. CONNECTED                                │
│    - Register connection in Redis           │
│    - Start heartbeat (every 30s)            │
│    - Update presence: ONLINE                │
│    - Deliver queued messages                │
│    Duration: Indefinite                     │
└─────────────────────────────────────────────┘
         ↓
┌─────────────────────────────────────────────┐
│ 4. DISCONNECTED                             │
│    - Heartbeat timeout (60s) OR             │
│    - Clean disconnect signal                │
│    - Remove from Redis                      │
│    - Update presence: OFFLINE               │
│    - Record last_seen                       │
└─────────────────────────────────────────────┘
```

Heartbeat Mechanism:

```
Purpose: Detect dead connections quickly

Client → Server (every 30 seconds):
  {
    "type": "PING",
```

```
    "timestamp": 1704672000
  }

Server → Client (immediate response):
  {
    "type": "PONG",
    "timestamp": 1704672001,
    "server_time": 1704672001  // For time sync
  }

Timeout handling:
  - If no PING for 60 seconds: Mark connection dead
  - Clean up Redis session
  - Update presence to OFFLINE
```

Reconnection Strategy:

```
Network interruption detected:

Client side:
  1. Detect connection drop (no PONG received)
  2. Attempt reconnect with exponential backoff:
     - Try 1: Immediately
     - Try 2: After 1 second
     - Try 3: After 2 seconds
     - Try 4: After 4 seconds
     - Max backoff: 30 seconds
  3. If reconnect succeeds:
     - Re-authenticate
     - Sync messages (get missed messages)
     - Resume normal operation

Server side:
  1. Old connection cleanup (automatic via TTL)
  2. New connection accepted
  3. Return missed messages since last_ack
```

2. End-to-End Encryption (E2EE)

Signal Protocol (Used by WhatsApp):

Key Components:

```
1. Identity Key Pair:
   - Long-term key pair per user
   - Public key stored on server
   - Private key never leaves device
```

```
2. Pre-Keys:
   - One-time keys for initial handshake
   - User uploads 100 pre-keys to server
   - Server distributes to initiators

3. Session Keys:
   - Ephemeral keys for each session
   - Rotated regularly (every 1000 messages)
   - Provides perfect forward secrecy
```

Initial Handshake (Alice → Bob, first message):

```
1. Alice's device:
   - Fetch Bob's identity key (from server)
   - Fetch one of Bob's pre-keys (from server)
   - Derive shared secret using ECDH
   - Generate session key

2. Encrypt message:
   - Use session key + AES-256-GCM
   - Include metadata for Bob to derive same key

3. Send encrypted message + handshake data

4. Bob's device:
   - Receive message
   - Use handshake data + own private keys
   - Derive same shared secret
   - Decrypt message

5. Both devices now have session key
6. Subsequent messages use session key directly
```

Perfect Forward Secrecy:

```
Mechanism: Rotate keys frequently

After every 1000 messages or 1 week:
  - Generate new ephemeral key
  - Derive new session key
  - Delete old session key

Result:
  If attacker compromises current key,
  they can't decrypt past messages
  (old keys already deleted)
```

Server's Role in E2EE:

```
Server CANNOT read message content!

Server can see:
✓ Sender user_id
✓ Receiver user_id
✓ Timestamp
✓ Message type (text/image)
✓ Encrypted payload (can't decrypt)

Server cannot see:
✗ Message content
✗ Media content (encrypted before upload)
✗ Who user talks to (metadata encrypted too in latest version)
```

3. Group Chat Architecture

Group Limits:

```
WhatsApp Groups:
- Max members: 256 (1024 in newer versions)
- Max admins: Unlimited
- Messages: No limit

Technical constraints:
- 256 members = 256 delivery operations
- Must complete in < 200ms
- Need efficient fanout mechanism
```

Group Message Fanout Strategy:

```
Challenge: Send 1 message to 256 recipients efficiently

Solution: Parallel async fanout

Architecture:
┌─────────────────────────────────────────────┐
│  Group Message Arrives                      │
└──────────────────┬──────────────────────────┘
                   ↓
┌─────────────────────────────────────────────┐
│  Store Once in Cassandra                    │
│  (group_id + message_id as composite key)   │
└──────────────────┬──────────────────────────┘
                   ↓
┌─────────────────────────────────────────────┐
```

```
│  Publish to Kafka                                    │
│  Topic: "group.message.fanout"                       │
│  Partition by: group_id (keeps order)                │
└──────────────────────┬───────────────────────────────┘
                       ↓
┌──────────────────────────────────────────────────────┐
│  Fanout Workers (20+ instances)                      │
│  Consume in parallel                                 │
└──────────────────────┬───────────────────────────────┘
                       ↓
              ┌────────┴────────┐
              ↓                 ↓
┌──────────────────┐   ┌──────────────────┐
│ Online Members   │   │ Offline Members  │
│ (Push via WS)    │   │ (Queue + Push)   │
└──────────────────┘   └──────────────────┘

Processing:
1. Fetch 256 members from cache (Redis SET)
2. Split into online (150) and offline (106)
3. Parallel delivery:
   - Online: Push to 150 WebSockets (parallel)
   - Offline: Queue for 106 users + push notifications
4. Track delivery: Store receipt per member
5. Update sender: "Delivered to 150/256"

Total time: ~100-150ms
```

Group Admin Features:

```
Admin Operations:
- Add/remove members
- Change group name/description/icon
- Promote/demote admins
- Delete messages (for everyone)
- Change group settings (who can message, edit info)

Implementation:
  1. Check if user is admin (cached role)
  2. If yes: Perform operation
  3. If no: Reject with 403 Forbidden
  4. Broadcast change to all members
  5. Update cache
```

4. Multi-Device Support

Challenge: User has phone, tablet, desktop - sync messages across all

Solution: Linked Devices Architecture

Primary Device (Phone):

```
- Stores encryption keys
- Can add/remove linked devices
- Acts as "master" device
```

Linked Devices (Tablet, Desktop):

```
- Paired with primary device
- Get encryption keys from primary
- Receive messages independently
- Send messages through primary device bridge
```

Pairing Flow:

```
1. User opens WhatsApp Web on desktop
2. Desktop generates QR code containing:
   - Device public key
   - Random session token

3. User scans QR code with phone

4. Phone:
   - Sends device public key to server
   - Establishes secure channel with desktop
   - Shares encryption keys

5. Server:
   - Registers new device for user
   - Creates device_id
   - Links to user account

6. Desktop now receives all messages
7. Phone can revoke desktop access anytime
```

Message Sync:

```
When message arrives:
1. Server delivers to ALL online devices
2. Each device independently:
   - Receives encrypted message
   - Decrypts using shared keys
   - Displays message

When primary device receives message:
```

```
1. Decrypt on phone
2. Re-encrypt for each linked device
3. Server forwards to linked devices

Result: All devices stay in sync
```

5. Message Search

Challenge: Search through billions of messages per user

Local Search (Preferred):

```
Storage: SQLite database on device
  - Messages stored locally
  - Full-text search index
  - Fast, private (no server query)

Query: SELECT * FROM messages
       WHERE conversation_id = ?
       AND content LIKE '%keyword%'
       ORDER BY created_at DESC
       LIMIT 50;

Benefits:
✓ Fast (local database)
✓ Private (search term not sent to server)
✓ Works offline
✓ No server load
```

Server-Side Search (Fallback):

```
Use case: Search across devices, or old messages not on device

Architecture:
  Messages → Kafka → Indexer → Elasticsearch

Elasticsearch Index:
  {
    "message_id": "msg_abc",
    "user_id": "user_123",  // Who can search
    "conversation_id": "conv_xyz",
    "content_encrypted": "...",  // Still encrypted!
    "created_at": "2024-01-08T10:00:00Z",
    "message_type": "text"
  }

Note: Content encrypted even in search index!
```

```
Search by metadata only (sender, date, type)
Full-text search requires client-side decryption
```

6. Read Receipts (Blue Ticks)

Privacy Design:

```
User can control read receipts:
- ON: Others see when you read (default)
- OFF: Others don't see, but you also can't see theirs

Implementation:
  - Setting stored in user profile
  - Enforced bidirectionally
  - If Alice disables: Bob doesn't see Alice's reads,
    and Alice doesn't see Bob's reads
```

Group Read Receipts:

```
Show who read the message in group

Display:
  ✓✓ Delivered to 200/256
  ✓✓ (blue) Read by 150/256

  Tap to see: List of members who read
  - Alice (read 2 mins ago)
  - Bob (read 5 mins ago)
  - Charlie (delivered, not read)
  - ...

Storage:
  Cassandra group_message_receipts table
  Track per-member delivery status
```

7. Typing Indicators

Requirements:

- Show "Alice is typing..." in real-time
- Don't spam (rate limit updates)
- Ephemeral (don't store permanently)

Implementation:

```
Client side:
  User starts typing:
    - After 1 second of typing: Send "typing_start"
    - While typing: Send "typing_continue" every 3 seconds
    - Stop typing: Send "typing_stop" OR wait 5 seconds

Server side:
  Receive "typing_start":
    1. Store in Redis: SETEX typing:conv_abc:user123 5 1
    2. Broadcast to conversation participants:
       - 1:1: Send to other person
       - Group: Send to all members (max 256)
    3. Auto-expire after 5 seconds (TTL)

  Receive "typing_stop":
    1. Delete from Redis: DEL typing:conv_abc:user123
    2. Broadcast "stopped" to participants

Recipient side:
  - Show indicator if Redis key exists
  - Hide automatically after 5 seconds (TTL)
```

Optimization: Debouncing

```
Don't send on every keystroke

Client logic:
  - Wait 1 second after first keystroke
  - Then send "typing_start"
  - Send "typing_continue" every 3 seconds
  - Send "typing_stop" on submit or 5s inactivity

Result: Reduce network traffic by 95%
```

8. Message Deletion

Two Types:

Delete for Me:

```
- Delete locally on device
- Mark as deleted in local DB
- Server still has message
- Other participants unaffected

Flow:
  1. User selects "Delete for me"
```

```
  2. App marks message as deleted locally
  3. Hide from UI
  4. Optional: Tell server to not sync to new devices
```

Delete for Everyone:

```
- Delete for all conversation participants
- Time limit: 1 hour after sending (configurable)
- Leaves tombstone: "This message was deleted"

Flow:
  1. User selects "Delete for everyone"
  2. Check time limit (< 1 hour since sent)
  3. Server updates message:
     - Set is_deleted = TRUE
     - Keep message_id (for ordering)
     - Clear content
  4. Server broadcasts deletion event:
     - To all online participants: Delete immediately
     - To offline participants: Deliver deletion on next connect
  5. All devices show: "This message was deleted"

Storage:
  - Keep message_id to maintain conversation flow
  - Clear content to save space
  - Mark with tombstone
```

Scalability & Reliability

Horizontal Scaling

Stateless Services (Easy to scale):

- Message Service
- User Service
- Group Service
- API Gateway

Auto-Scaling Rules:

```
Scale Up Triggers:
├── CPU > 70% for 5 minutes
├── Memory > 80% for 5 minutes
├── Request queue > 1000
└── WebSocket connections > 80% capacity

Scale Down Triggers:
```

```
  ├── CPU < 30% for 15 minutes
  ├── Request queue < 100
  └── Min instances: 100 per region (always)

  Scaling speed:
  - Scale up: +50% capacity in 2 minutes
  - Scale down: -10% capacity in 10 minutes (gradual)
```

Stateful Services (Complex scaling):

WebSocket Gateways:

```
  Challenge: Can't just kill connections

  Graceful scaling:
    Scale up: Simple, new servers accept new connections

    Scale down:
      1. Mark server for removal
      2. Stop accepting new connections
      3. Wait for existing connections to disconnect naturally
      4. Or drain connections:
         - Send reconnect signal to clients
         - Clients reconnect to other gateways
      5. After all connections drained: Remove server

  Drain time: 5-10 minutes
```

Database Scaling

Cassandra Scaling:

```
  Horizontal scaling (add nodes):
    Current: 100 nodes
    Each node: 2TB storage, 10K writes/sec
    Total capacity: 200TB, 1M writes/sec

    To double capacity:
      Add 100 nodes
      Cassandra automatically rebalances data
      Time: 2-4 hours for rebalancing

    No downtime during scaling
```

MySQL Scaling:

```
Vertical scaling (larger instances):
  - Upgrade from db.r5.4xlarge → db.r5.8xlarge
  - Requires brief downtime (5-10 minutes)
  - Plan during maintenance window

Horizontal scaling (read replicas):
  - Add read replicas for read queries
  - Write still goes to master
  - Eventual consistency for reads (< 1 second lag)

Current setup:
  - 1 master (writes)
  - 5 read replicas per region (reads)
  - Cross-region replicas for DR
```

Redis Scaling:

```
Redis Cluster mode:
  - 16,384 hash slots
  - Distributed across nodes
  - Add nodes to increase capacity

Current setup:
  - 200 nodes × 25GB = 5TB total memory
  - 3 replicas per shard
  - Handle 10M operations/sec
```

Multi-Region Architecture

Global Deployment:

```
┌─────────────────────────────────────────────────────┐
│  REGION: US-EAST (Primary)                          │
│  ├── WebSocket Gateways: 3000                       │
│  ├── Cassandra Cluster: 30 nodes                    │
│  ├── MySQL: Master + 5 replicas                     │
│  ├── Redis: 50 nodes                                │
│  └── Serves: North America, South America           │
└─────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────┐
│  REGION: EU-WEST (Secondary)                        │
│  ├── WebSocket Gateways: 2500                       │
│  ├── Cassandra Cluster: 25 nodes                    │
│  ├── MySQL: Replica                                 │
│  ├── Redis: 40 nodes                                │
│  └── Serves: Europe, Middle East, Africa            │
```

```
└──────────────────────────────────────────────────────────┘


┌──────────────────────────────────────────────────────────┐
│  REGION: AP-SOUTH (Secondary)                            │
│  ├── WebSocket Gateways: 2500                            │
│  ├── Cassandra Cluster: 25 nodes                         │
│  ├── MySQL: Replica                                      │
│  ├── Redis: 40 nodes                                     │
│  └── Serves: Asia-Pacific                                │
└──────────────────────────────────────────────────────────┘

Cross-Region Replication:
- Cassandra: Automatic multi-datacenter replication
- MySQL: Async replication (< 1 second lag)
- Redis: Independent per region (eventual consistency OK)
```

User Routing:

```
Strategy: Route to nearest region

User in India:
  1. DNS resolves to ap-south load balancer
  2. Connect to ap-south WebSocket gateway
  3. Messages stored in global Cassandra
  4. Low latency (<50ms in-region)

Cross-region messaging:
  Alice (US) → Bob (India):
    1. Alice sends to us-east gateway
    2. Stored in Cassandra (replicated to ap-south)
    3. Bob's gateway (ap-south) reads from local Cassandra
    4. Delivers to Bob

  Total latency: ~100-150ms (acceptable)
```

High Availability

Failure Scenarios & Handling:

WebSocket Gateway Failure:

```
Detection: Health check failure (3 consecutive)

Impact: Users on that gateway disconnected

Mitigation:
  1. Load balancer removes failed gateway
```

```
2. Clients detect disconnection
3. Clients reconnect to
```