System Design Day8: Chat system design

💬 Chat System Design – Summary

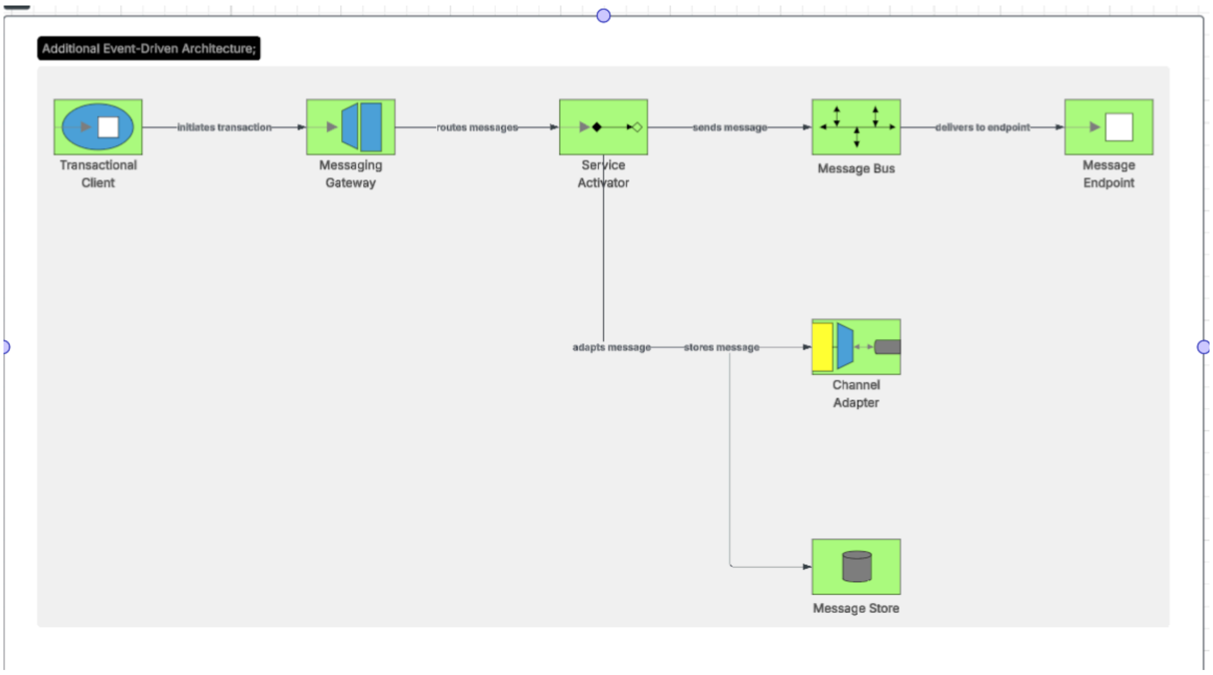
🔷 Core Functional Requirements

- **1:1 and group messaging**
- **Message delivery guarantee** (at least once)
- **Message ordering**
- **Real-time delivery**
- **Online/offline presence**
- **Typing indicators, read receipts**

🔷 Non-Functional Requirements

- Low latency (<100ms for delivery)
- High availability & reliability
- Horizontal scalability
- Eventual consistency (across devices)
- Fault tolerance

🏗️ High-Level Architecture



🔧 Components:

Component	Responsibility
Client (Mobile/Web)	Opens persistent connection via WebSocket or HTTP long polling
WebSocket Gateway	Maintains persistent connection; handles message push
Chat Service	Validates, stores, and routes messages
Kafka/Queue	Ensures message durability, ordering, and delivery retries
Redis	Stores online/offline presence, recent messages, typing indicators
Database (MySQL/MongoDB)	Stores chat history, metadata

🔁 Message Flow (1:1 chat)

1. **Client** sends message via WebSocket → App Server.
2. **App Server** authenticates and pushes to Kafka topic (partitioned by chat ID).
3. **Chat Consumer Service** reads from Kafka, persists to DB, updates Redis if recipient is online.
4. **Delivery Service** pushes message to recipient via WebSocket if connected.

5. If offline, marks message as undelivered or queues for push notification.

🛠️ Java & Spring Boot Tie-ins

1. **WebSocket Support**

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConf
igurer {
  public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/chat").setAllowedOrigins("*").withSockJS();
  }
  public void configureMessageBroker(MessageBrokerRegistry registry) {
    registry.enableSimpleBroker("/topic", "/queue");
    registry.setApplicationDestinationPrefixes("/app");
  }
}
```

- Use `@MessageMapping` for incoming messages.
- Use `SimpMessagingTemplate` to push to client queues.

2. **Kafka Integration**

```
@KafkaListener(topics = "chat-messages", groupId = "chat")
public void listen(MessagePayload payload) {
    chatService.persistAndForward(payload);
}
```

- Ensures **decoupling** and **asynchronous delivery**.
- Kafka ensures **durability**, **partitioning**, and **retries**.

3. **Redis Use Cases**

- Store **user presence**: `user:online:{userId}`
- Cache **recent messages** (for fast reloads)
- **Pub/Sub** between multiple WebSocket nodes

```
redisTemplate.opsForValue().set("user:online:123", true, Duration.ofMinutes
(5));
```

4. **Spring Security** (Optional)

- Authenticate WebSocket connection using JWT or session.
- Protect `@MessageMapping` endpoints.

📦 Features to Support

Feature	How to Handle (Spring/Infra)
Typing status	Send ephemeral status updates via Redis pub/sub or WebSocket
Read receipts	Update DB + notify sender via WebSocket
Offline storage	Use DB (e.g., MongoDB) with TTL or disk-based Kafka topics for durability
Reconnect resume	Store last-received message ID in Redis/session
Mobile push	Queue message if user offline, send via FCM/APNs

🧠 System Design Considerations

Concern	Strategy
Scalability	Stateless WebSocket servers, Redis for session state, Kafka for scaling consumers
Ordering	Kafka partitions by `chat_id` , maintain sequence ID
Fault Tolerance	Use durable Kafka topics, persistent DB, Redis replication
Latency	Redis caching, direct WebSocket push, load balancing WebSocket endpoints
High Throughput	Use Kafka + async processing, horizontal scaling of ChatProcessor microservice

🧩 Optional Advanced Additions

- **End-to-end encryption (E2EE)** — client-side only.
- **Media messages** — use blob storage (e.g., AWS S3, GCS).

- **Rate limiting** — apply at gateway using Spring filters or API gateway like Kong/NGINX.
- **Monitoring** — with Prometheus, ELK, and Grafana.
- **Tracing** — use Sleuth + Zipkin for distributed tracing.

✅ Summary Table

Feature	Technology (Spring/Java)
Real-time messaging	Spring WebSocket, STOMP
Message routing	Kafka (partitioned by chat ID)
User presence	Redis
Chat history	MongoDB / MySQL
Authentication	Spring Security + JWT
Rate Limiting	API Gateway filters or Redis token bucket
Distributed delivery	Kafka + WebSocket Cluster
