The Contract: Defining Scope & Constraints

FUNCTIONAL REQUIREMENTS (The Product)

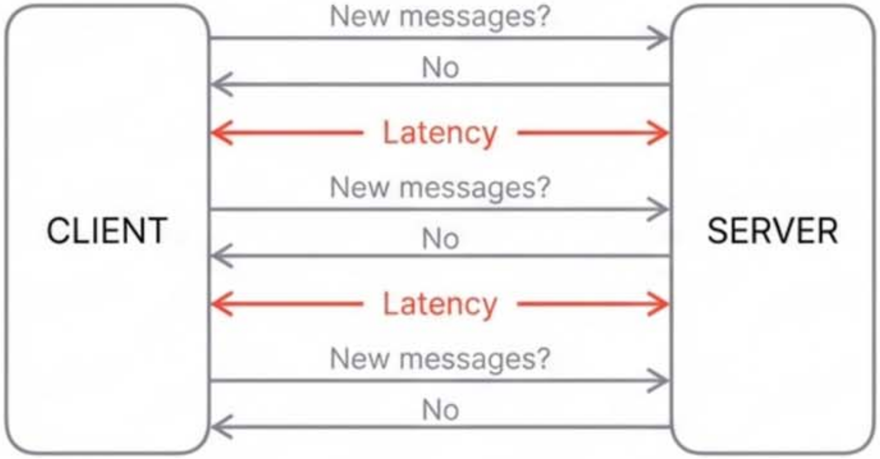
- **1:1 & Group Chat:** Support for multi-user participation (Cap: 100 users/group).
- **Message Delivery:** Text, Images, and Video support.
- **Offline Support:** Store & Forward mechanism for disconnected users.
- **Ack Mechanism:** Sent, Delivered, and Read receipts.

NON-FUNCTIONAL REQUIREMENTS (The Engineering)

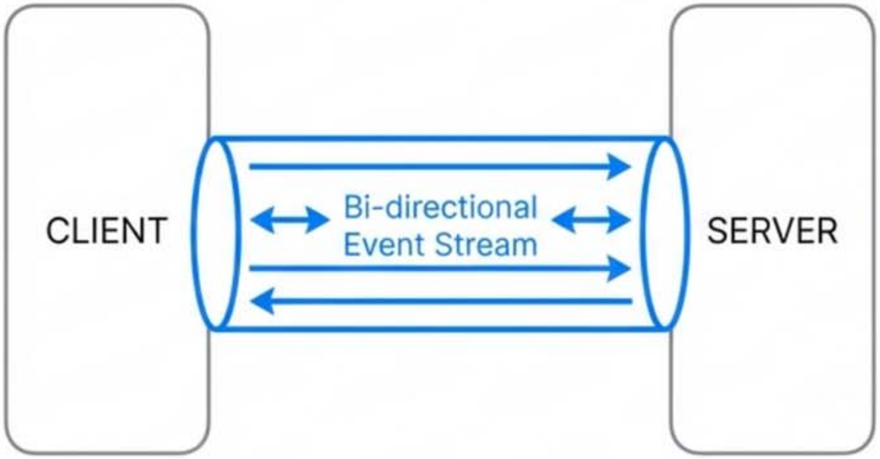
- **Scale:** 1 Billion DAU. ~40k msg/sec average, ~230k+ peak RPS.
- **Latency:** Real-time delivery target <500ms (P99).
- **Reliability:** 'At least once' delivery guarantee. Zero message loss.
- **Ephemeral Storage:** Strict privacy policy. Messages deleted from server upon delivery (Max retention: 30 days).

The Protocol: Why We Don’t Use REST

HTTP Polling (Inefficient)



WebSocket (Persistent)



The API Contract (Events)

```
-> createChat(participants) : returns chatId
-> sendMessage(chatId, content) : returns status
<- receiveMessage(chatId, senderId, content) : [SERVER PUSH]
-> ackMessage(messageId) : [CLIENT CONFIRM]
```

The Data Model: Optimizing for Write Heavy Workloads

Database Choice: DynamoDB (Key-Value / Wide-Column)

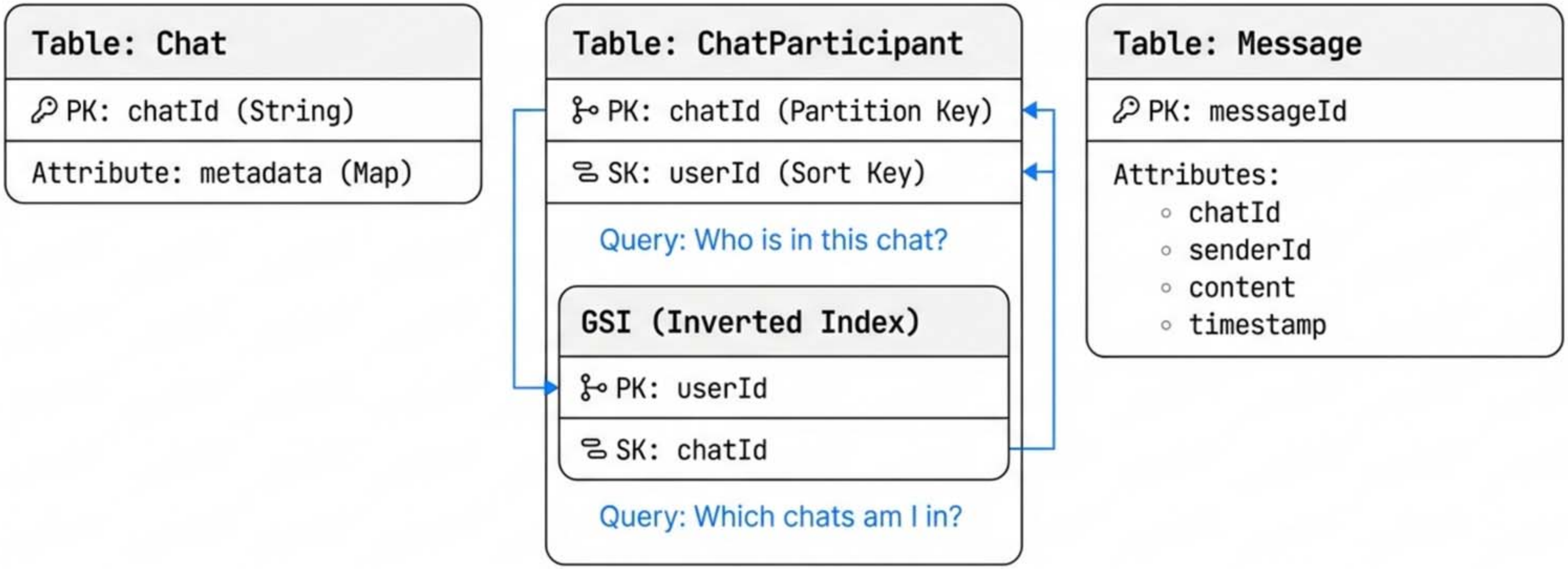


Table: Chat

- PK: chatId (String)
- Attribute: metadata (Map)

Table: ChatParticipant

- PK: chatId (Partition Key)
- SK: userId (Sort Key)

Query: Who is in this chat?

GSI (Inverted Index)

- PK: userId
- SK: chatId

Query: Which chats am I in?

Table: Message

- PK: messageId

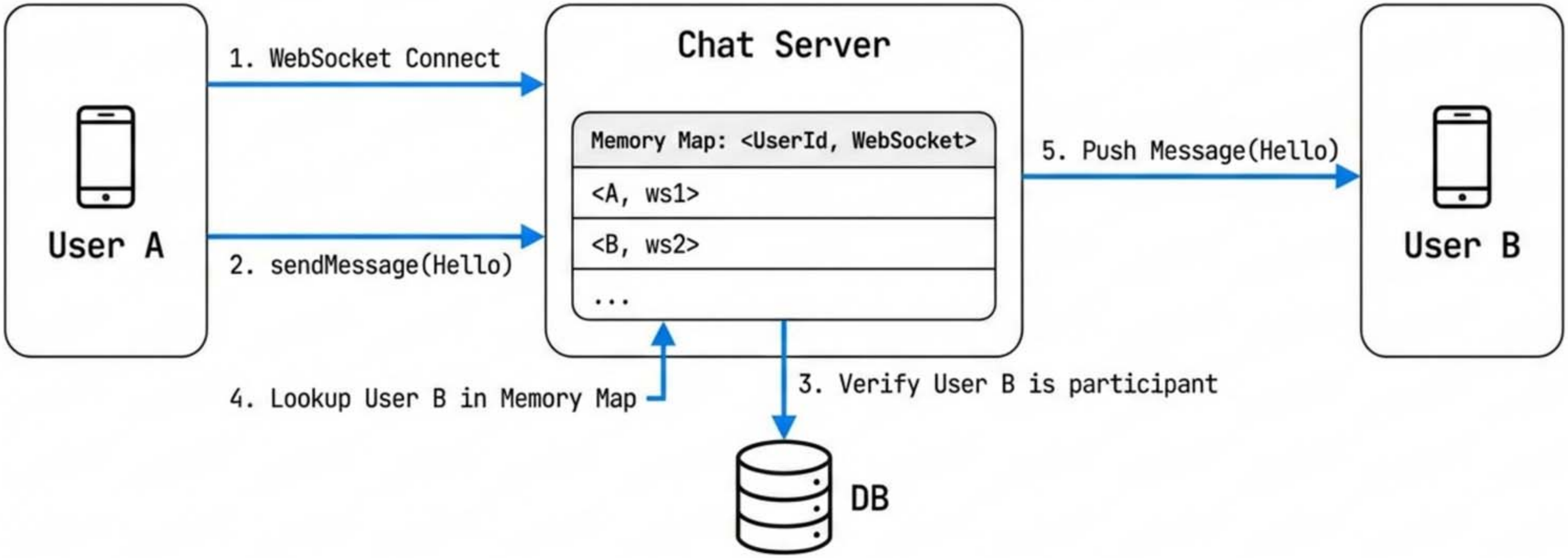
Attributes:

- chatId
- senderId
- content
- timestamp

Reasoning: DynamoDB handles massive horizontal scaling for write-heavy chat logs better than traditional SQL joins.

The Functional Baseline (The MVP)

Act II: A Single Server Solution



User A

1. WebSocket Connect

2. sendMessage(Hello)

Chat Server

Memory Map: <UserId, WebSocket>
<A, ws1>
<B, ws2>
...

3. Verify User B is participant

4. Lookup User B in Memory Map

5. Push Message(Hello)

User B

DB

CONSTRAINT: Works for 500 users. Fails at 1 Billion (Memory/CPU limits).

The ‘Offline’ Problem: Store and Forward

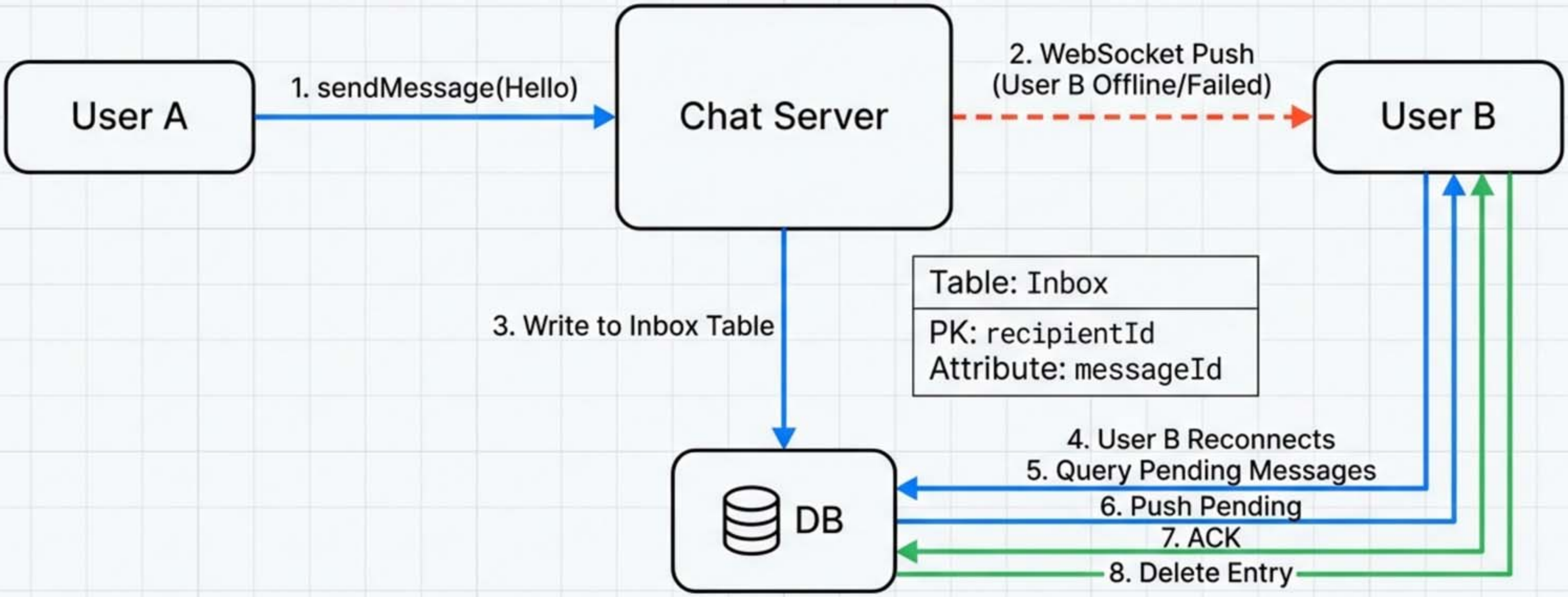
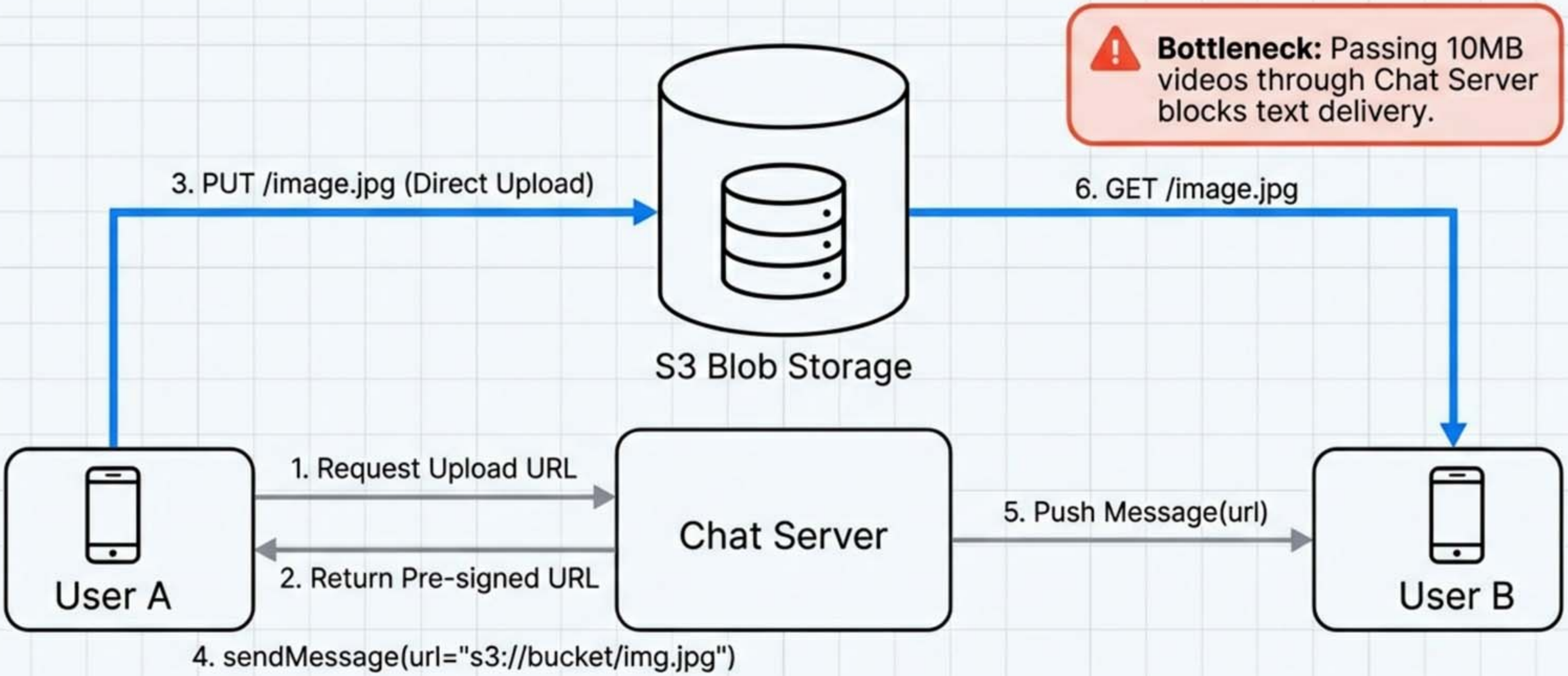


Table: `Inbox`

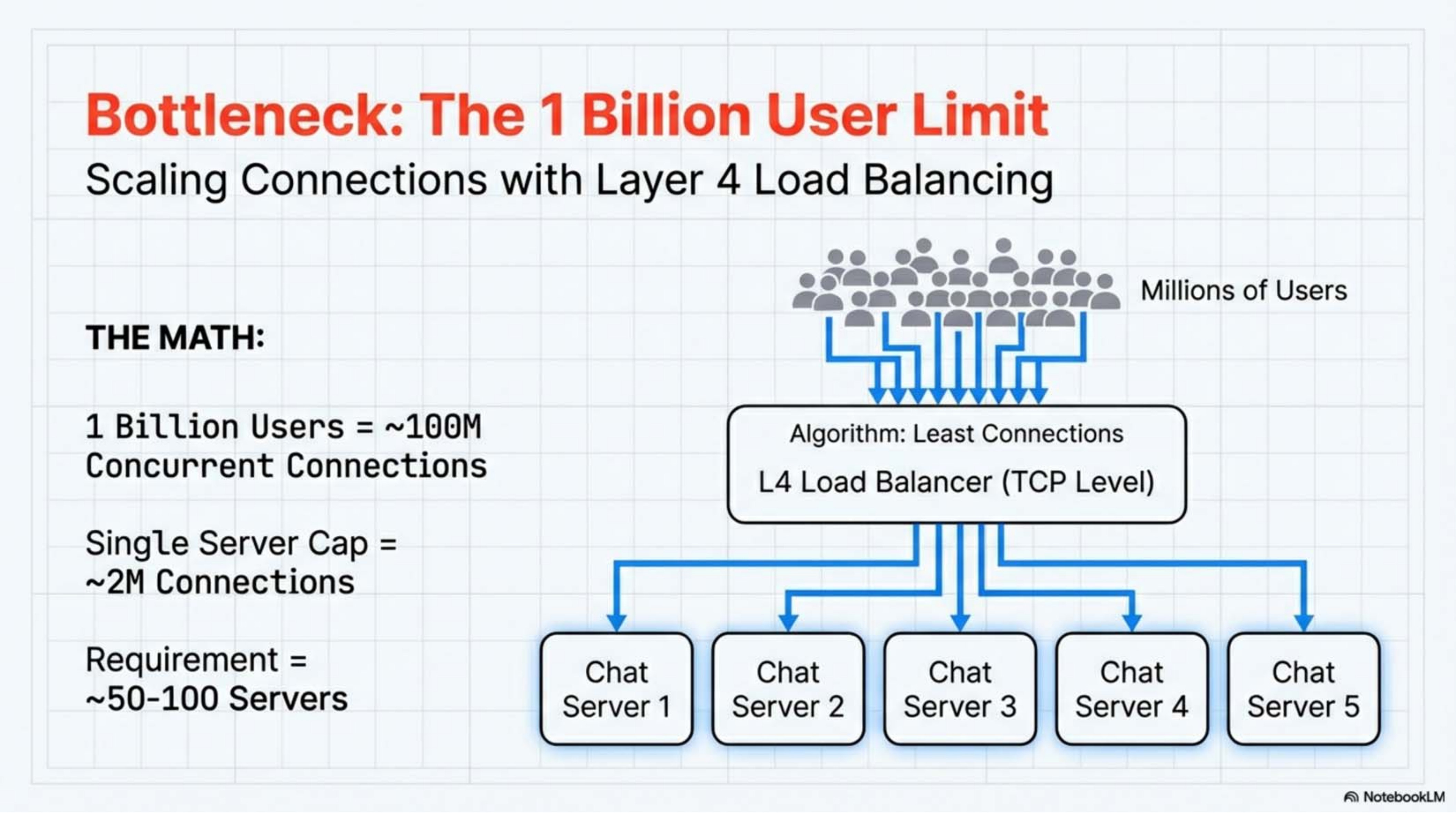
PK: `recipientId`
Attribute: `messageId`

Handling Media: Decoupling Data from Signaling



Bottleneck: Passing 10MB videos through Chat Server blocks text delivery.

NotebookLM



Bottleneck: The 1 Billion User Limit

Scaling Connections with Layer 4 Load Balancing

THE MATH:

1 Billion Users = ~100M Concurrent Connections

Single Server Cap = ~2M Connections

Requirement = ~50-100 Servers

Millions of Users

Algorithm: Least Connections

L4 Load Balancer (TCP Level)

Chat Server 1

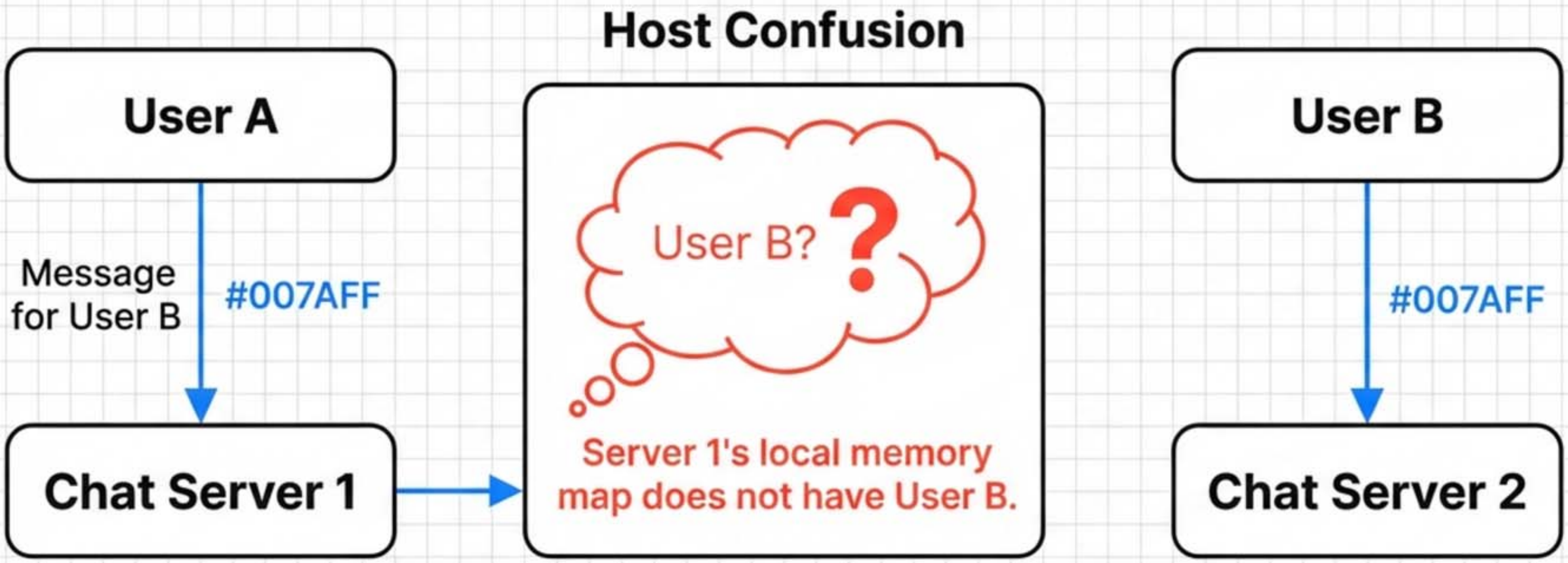
Chat Server 2

Chat Server 3

Chat Server 4

Chat Server 5

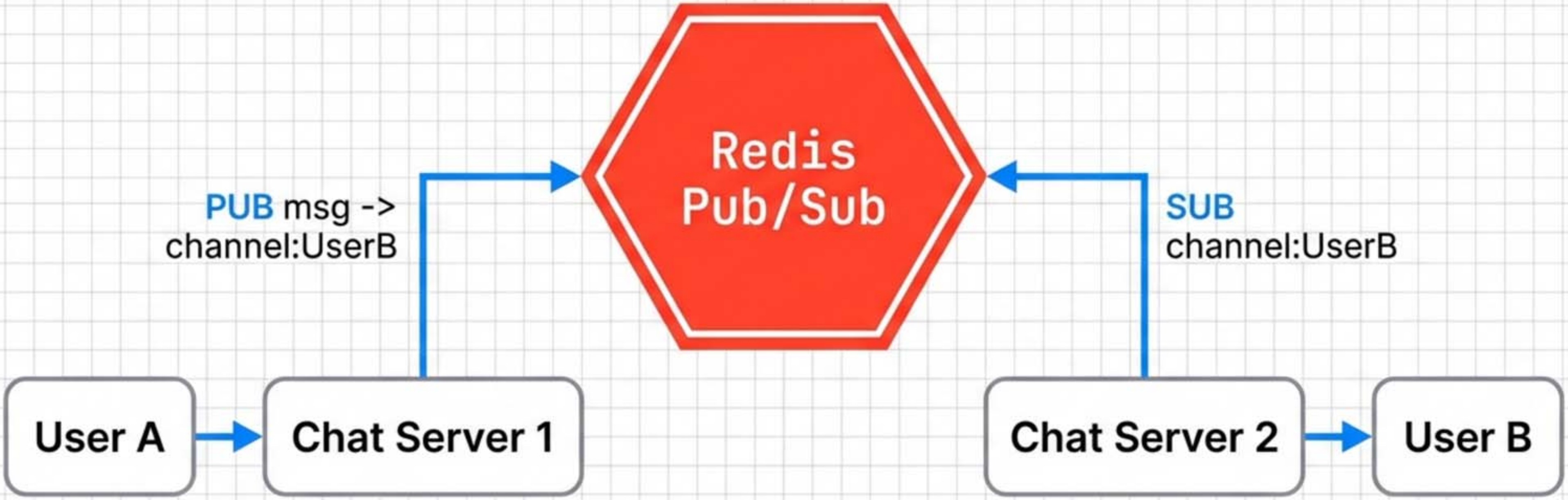
The Routing Challenge: Bridging the Islands



WHY NOT KAFKA? Kafka topics are heavy. Creating **`1 Billion`** topics (one per user) is inefficient. Kafka is for streams, not ephemeral routing.

The Solution: Redis Pub/Sub

Turning isolated servers into a unified mesh.



User A

Chat Server 1

PUB msg -> channel:UserB

Redis Pub/Sub

SUB channel:UserB

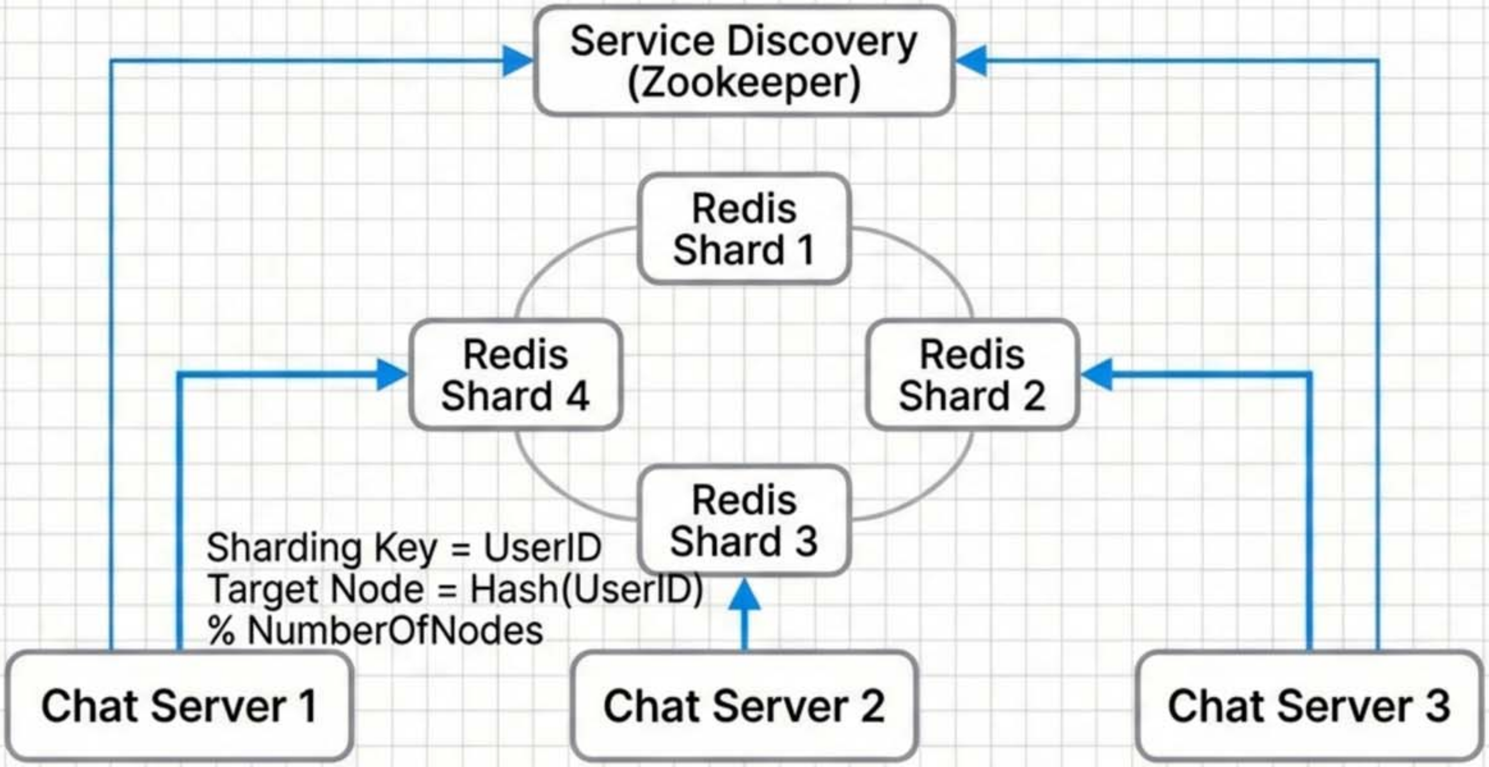
Chat Server 2

User B

Redis acts as a lightweight, ephemeral message bus.

Scaling Redis: Sharding and Clusters

⚠ Single Redis Node = Memory/CPU Bound.



Service Discovery (Zookeeper)

Redis Shard 1

Redis Shard 2

Redis Shard 3

Redis Shard 4

Sharding Key = UserID
Target Node = Hash(UserID) % NumberOfNodes

Chat Server 1

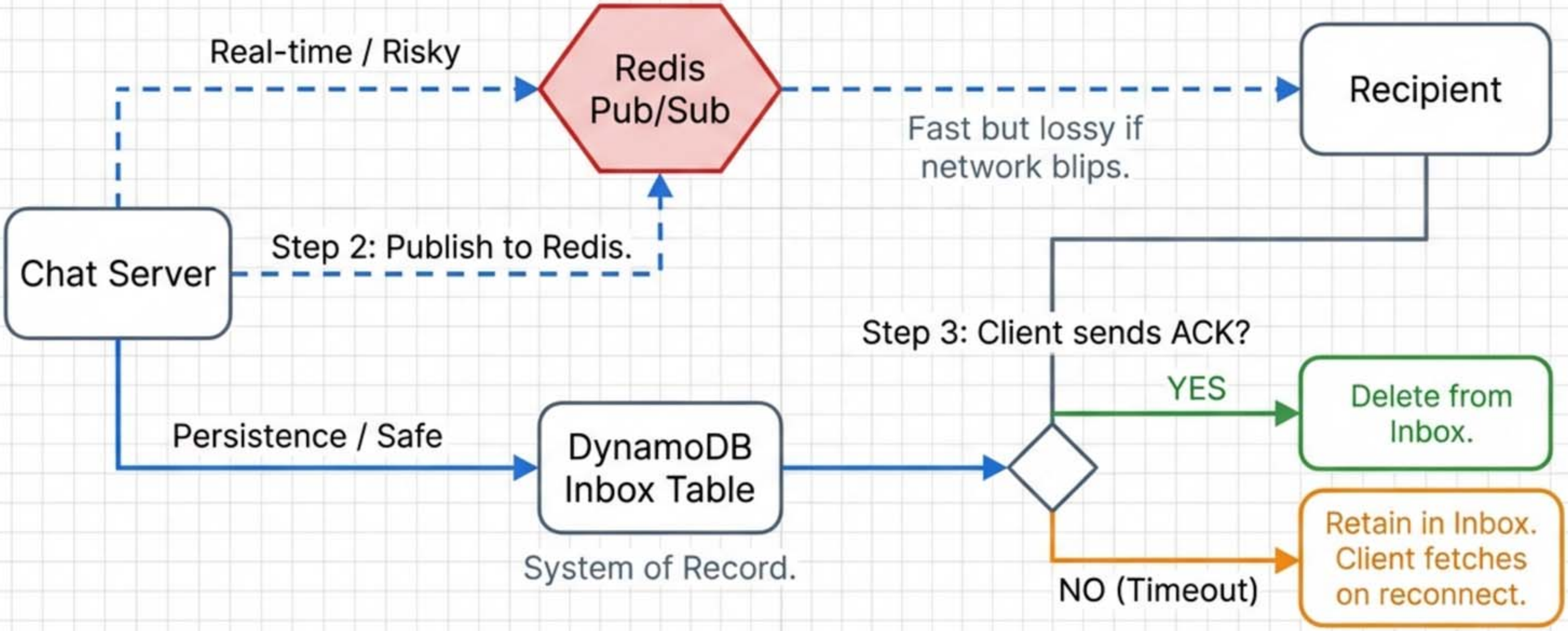
Chat Server 2

Chat Server 3

Connection Pooling enabled to minimize TCP handshake overhead.

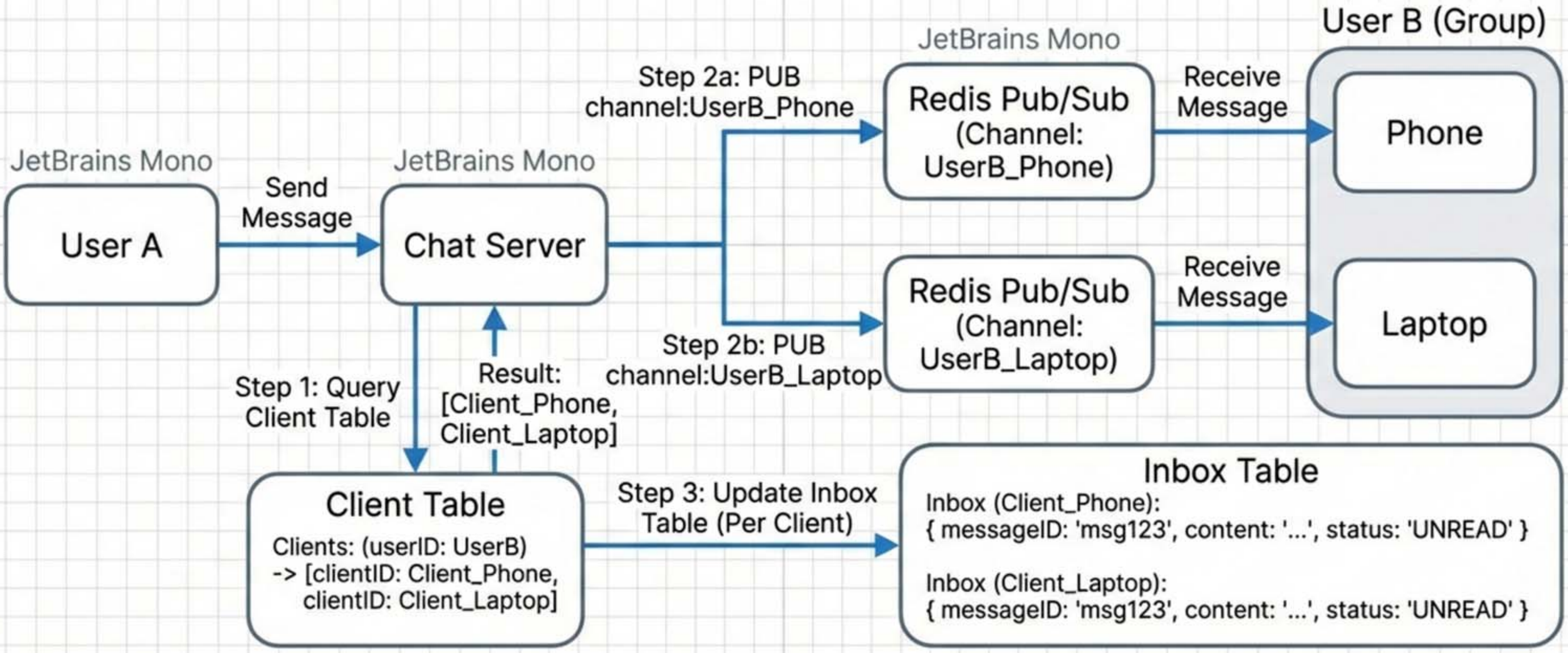
Reliability: Solving “At Most Once” Delivery

Implementing a persistent messaging strategy to prevent data loss.



Multi-Device Support: Fan-out Complexity

Implementing a multi-dart messaging strategy to prevent data loss.


User B (Group)
JetBrains Mono
Step 2a: PUB channel:UserB_Phone
Redis Pub/Sub (Channel: UserB_Phone)
Receive Message
Phone
JetBrains Mono
JetBrains Mono
User A
Send Message
Chat Server
Redis Pub/Sub (Channel: UserB_Laptop)
Receive Message
Laptop
Step 2b: PUB channel:UserB_Laptop
Step 1: Query Client Table
Result: [Client_Phone, Client_Laptop]
Client Table
Step 3: Update Inbox Table (Per Client)
Inbox Table

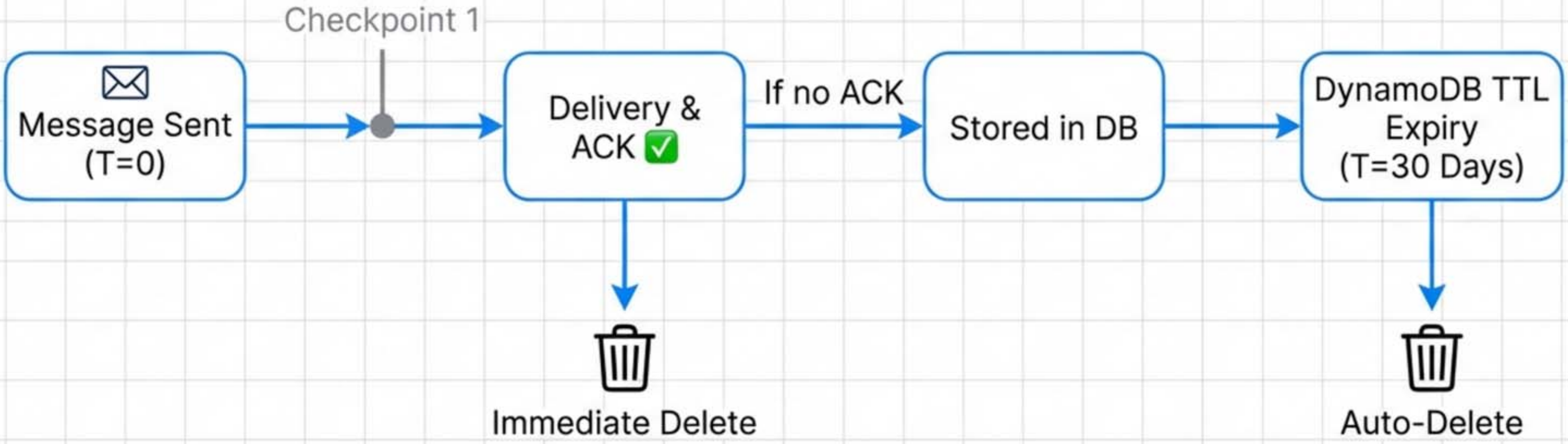
Clients: (userID: UserB)
-> [clientID: Client_Phone,
clientID: Client_Laptop]

Inbox (Client_Phone):
{ messageID: 'msg123', content: '...', status: 'UNREAD' }

Inbox (Client_Laptop):
{ messageID: 'msg123', content: '...', status: 'UNREAD' }

Ephemeral Storage & Cleanup

Privacy Principle: Data is Toxic Sludge.



Checkpoint 1

Message Sent (T=0)

Delivery & ACK ✅

If no ACK

Stored in DB

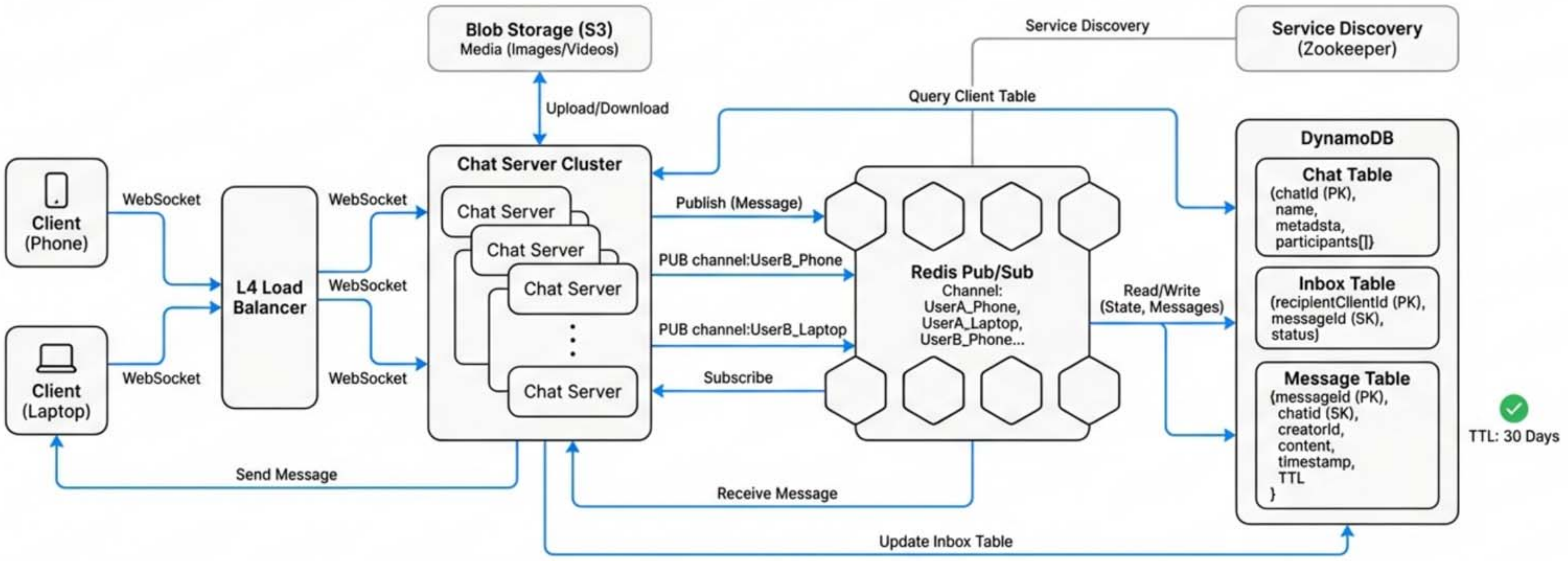
DynamoDB TTL Expiry (T=30 Days)

Immediate Delete

Auto-Delete

Mechanism: DynamoDB TTL (Time To Live). Database engine automatically expires rows based on timestamp. No custom cleanup code required.

The Final Architecture



Summary & Trade-offs

PROS:

- Low latency (<500ms)
- Horizontal Scale (1B Users)
- High Reliability (Hybrid DB/Redis)

CONS:

- Operational complexity of Redis Cluster
- Client-side logic needed for message ordering

NotebookLM