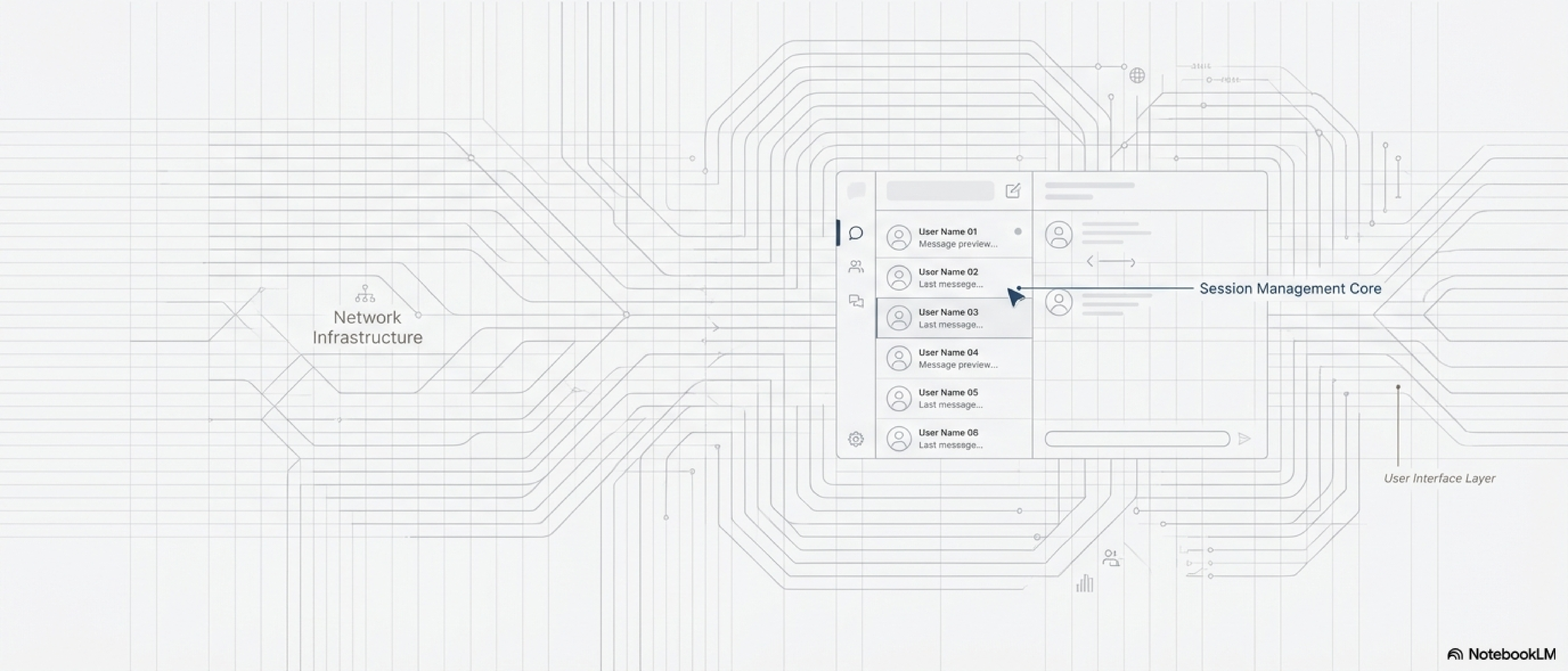
Architecting Conversations at Scale

The Design and Philosophy of the IM Session Model



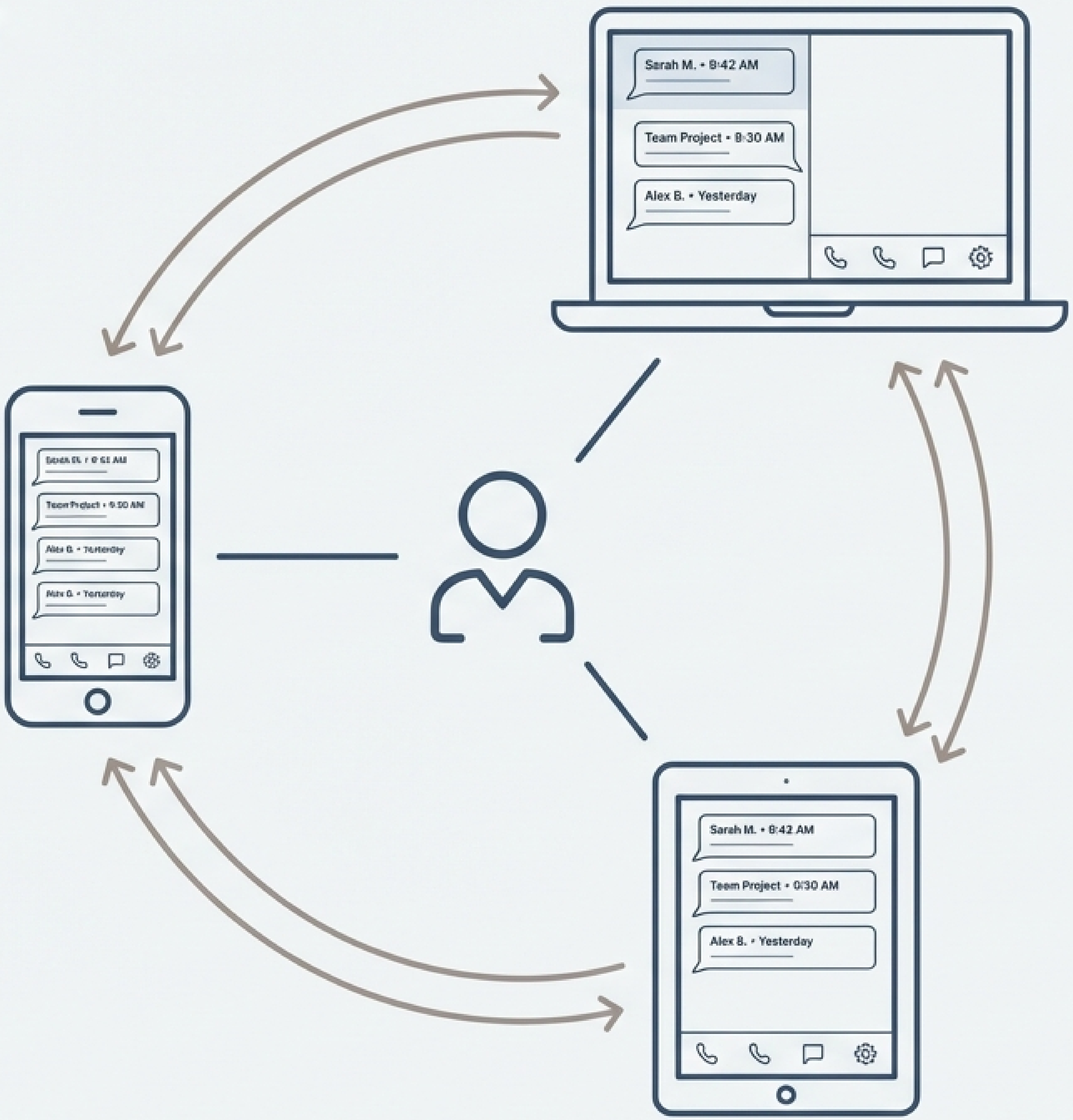
NotebookLM

The Goal: A Flawless, Synchronized Conversation List

The primary user interface for any IM application is the session list. The user expects it to be:

- ✓ **Instantly Responsive:** New messages and state changes appear immediately.
- ✓ **Perfectly Synchronized:** The list is identical across all devices (mobile, desktop, web).
- ✓ **Intelligently Ordered:** The list is sorted in a way that feels intuitive and useful.

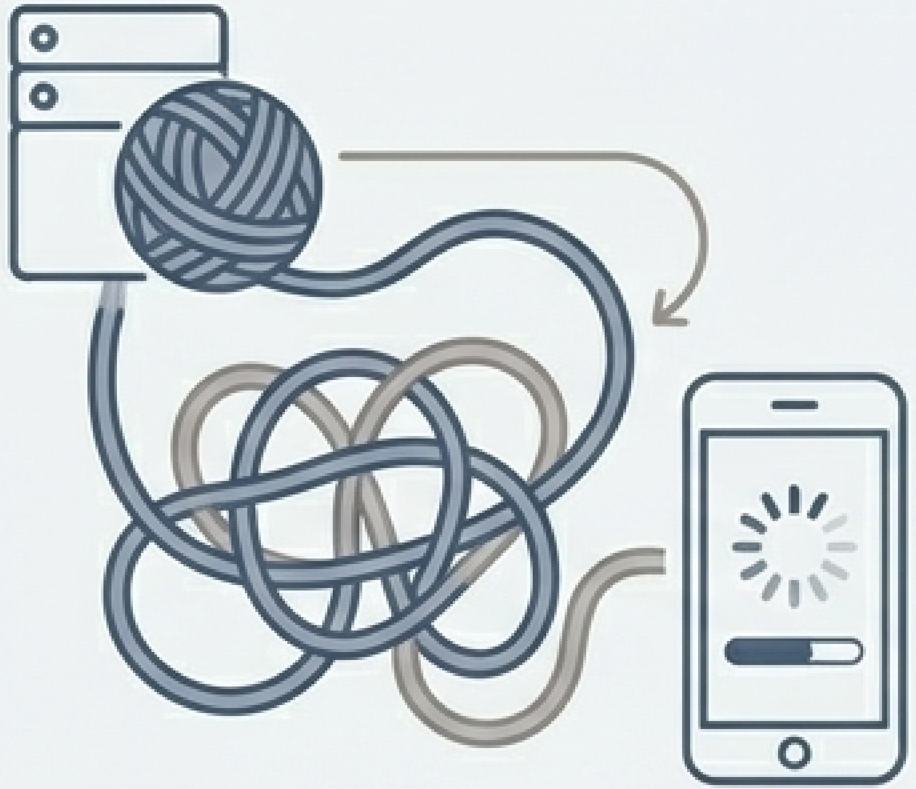
The Engineering Challenge: How do we deliver this experience reliably for a system handling over 100 billion unique conversations?



Why Not Just Build the List from a Message Queue?

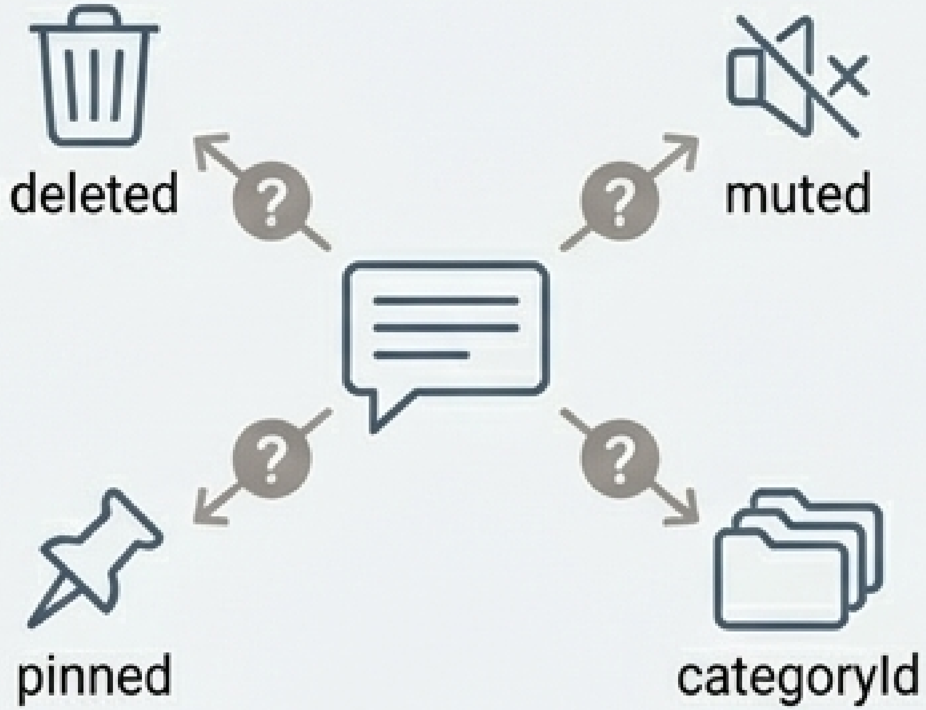
A common initial thought is to reconstruct the session list on the client by pulling from a user-specific message queue (or ‘message inbox’). While simple in theory, this model fails at scale and for complex features.

Performance Bottleneck



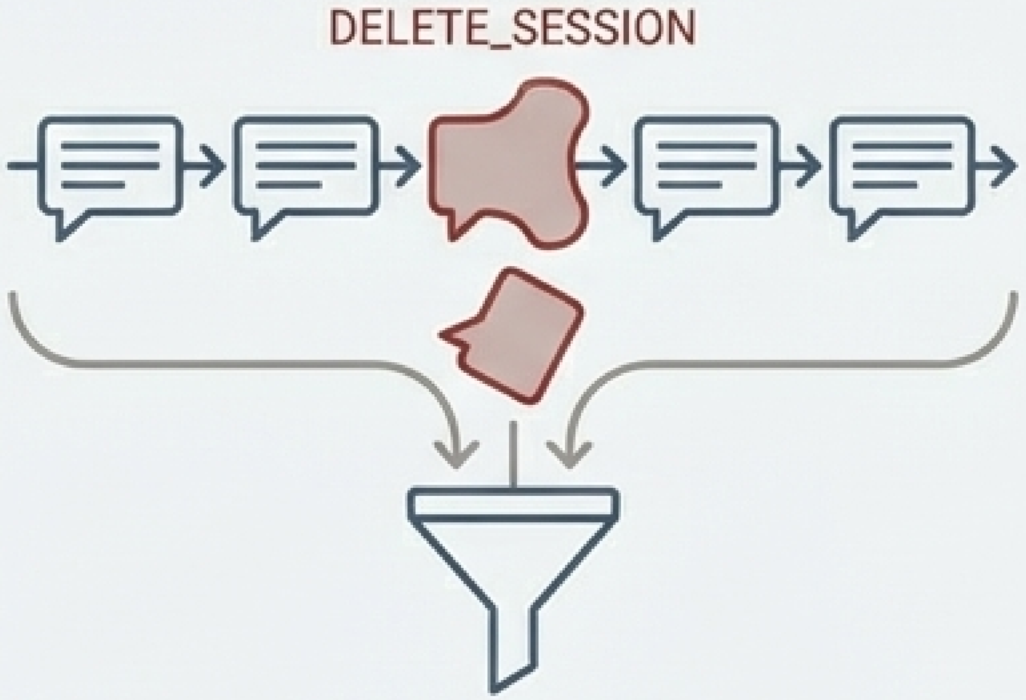
Syncing requires pulling all incremental messages, which is slow if a user has many messages across few sessions. This degrades the login/launch experience.

Incomplete State Representation



A message stream cannot natively represent session-level metadata. How do you model these states with just messages?

Model Impurity



Forcing session state changes into the message stream corrupts the message model. This requires complex client logic to filter and interpret these special ‘signal’ messages.

The Solution: The Session as a First-Class Entity

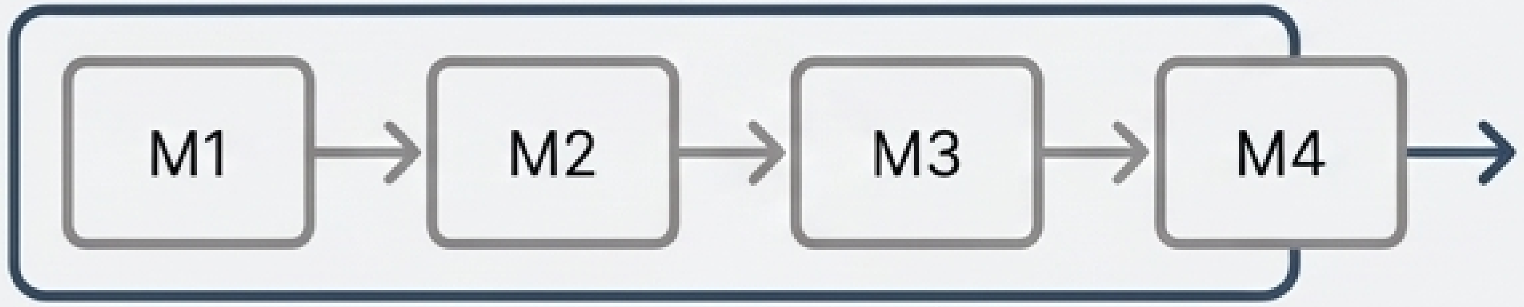
Instead of inferring state from a stream of messages, we model the "Session" as a distinct object.
A Session represents the complete **context** of a conversation for a single user.

- For a 1-on-1 chat, User A and User B each have their own separate Session object.
- For a group chat, every member has their own Session object for that group.

A Tale of Two Models

Message Queue Model

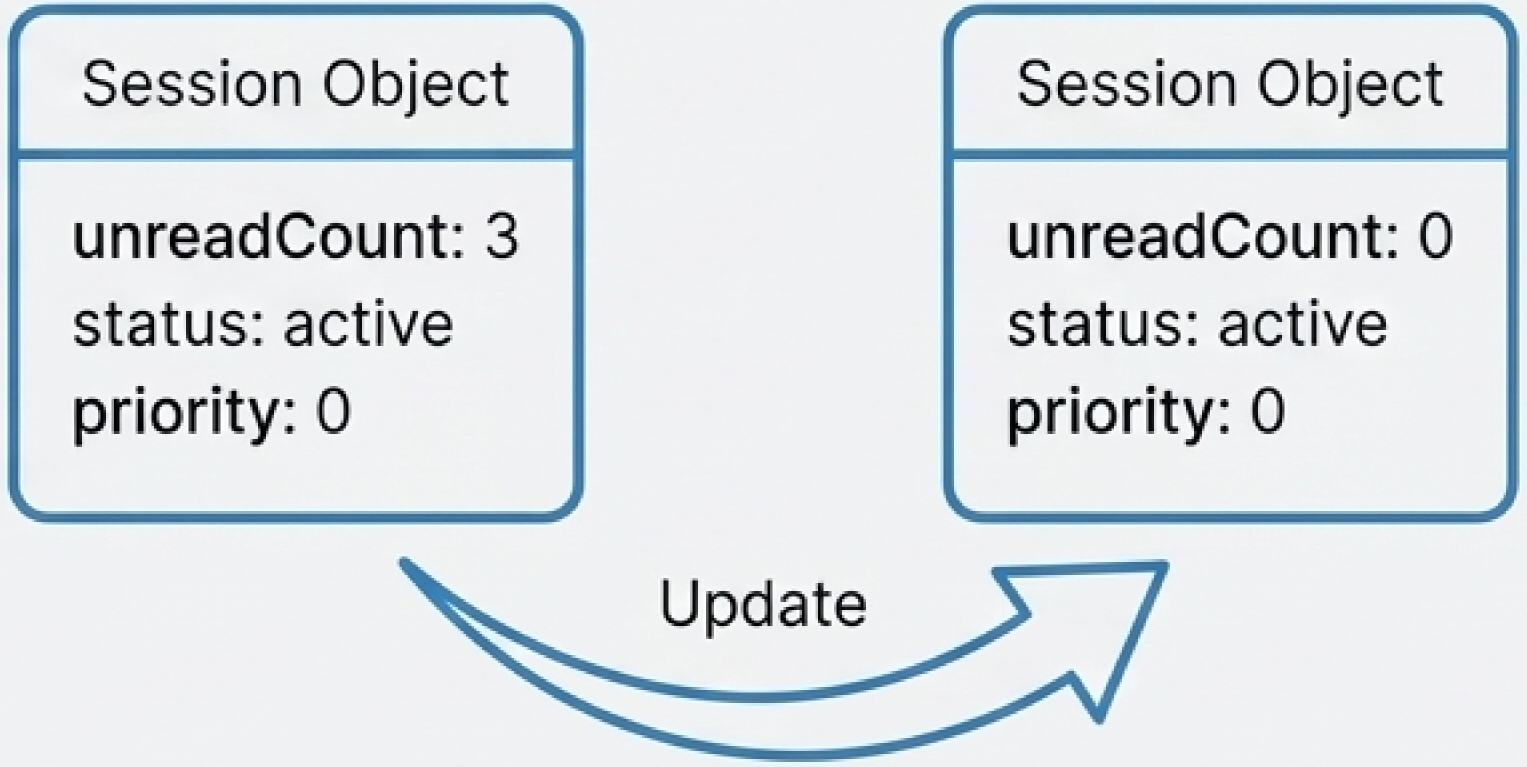
Append, Don’t Lose



Append-only Log

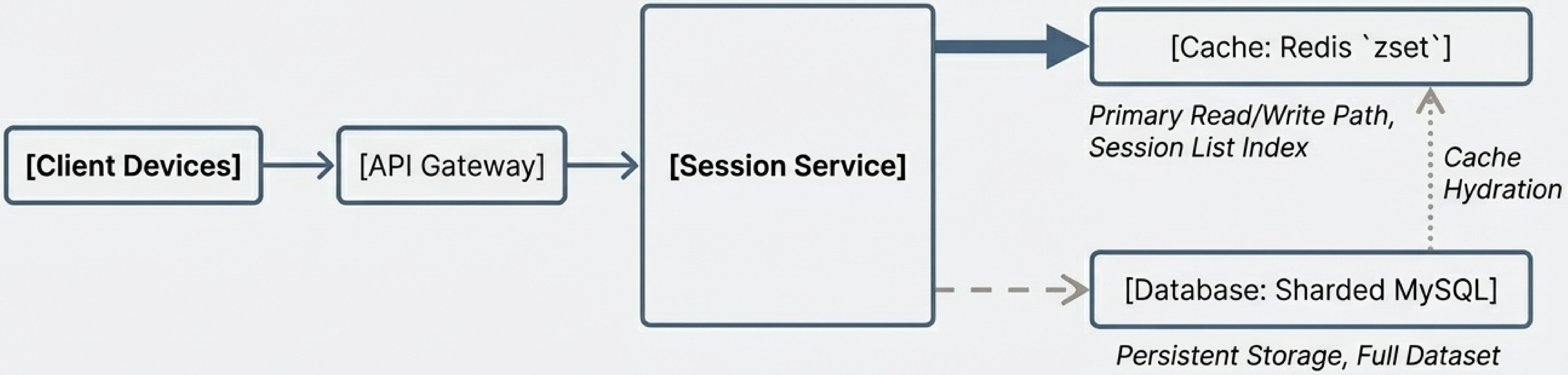
Session Model

Overwrite, Don’t Append



The Architectural Blueprint: Cache-First, DB-Backed

To achieve low latency, the system is designed around a cache-first pattern. The database serves as the persistent source of truth, but the hot data path is optimized through Redis.



> The session list itself is stored as a sorted set (`zset`) in Redis, acting as a dynamic index. This is the key to fast, incremental synchronization.

The Core Mechanism: Decoupling Sync Logic from Display Logic

The key to a great user experience is ensuring the session list only re-sorts when the user expects it to. We achieve this with two distinct timestamps for every session.

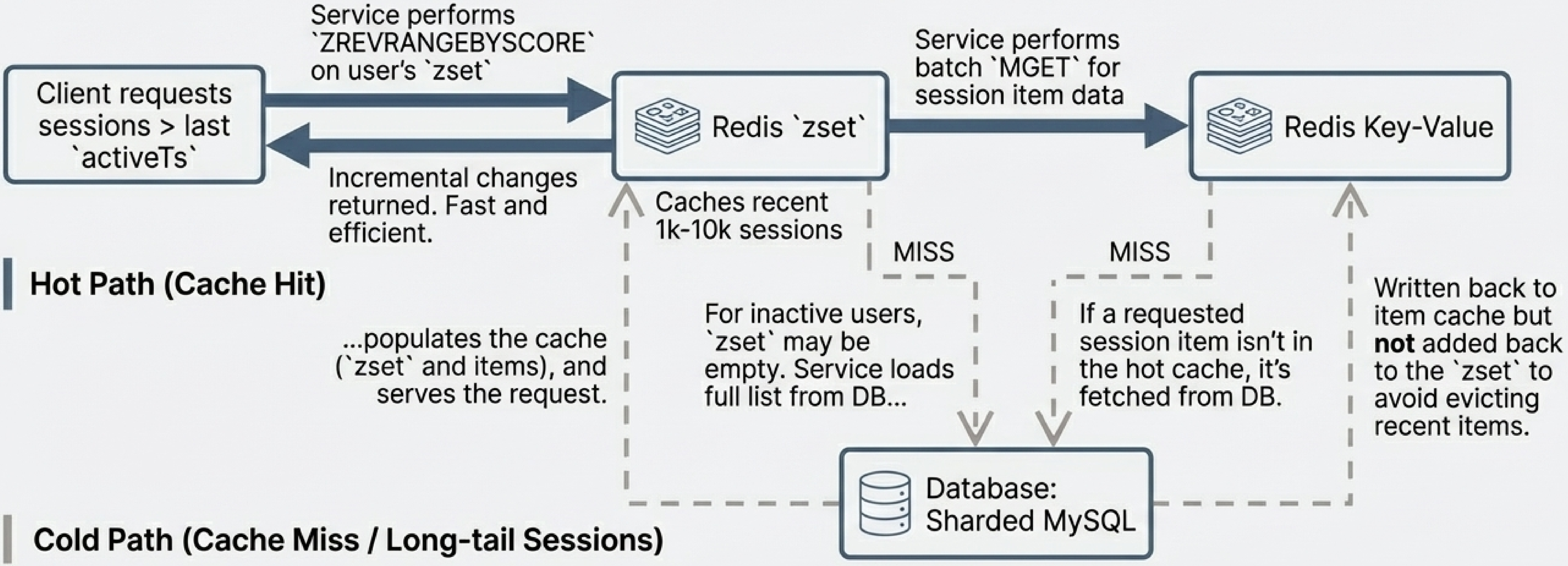
activeTs (Active Timestamp)	writeTs (Write Timestamp)
Purpose: Server-side synchronization. **Trigger:** Updated on any change to the session object. **Analogy:** The absolute ‘last modified’ time of the data record.	**Purpose:** Client-side sorting and UX. **Trigger:** Updated only on changes that should reorder the list. **Analogy:** The ‘last important event’ time for the user.
Marking a session as read Receiving a read receipt Muting a session *Also triggered by all writeTs events.*	Receiving a new message Pinning a conversation Manually marking as unread

The server sends all updates based on `activeTs`. The client uses these updates to refresh local data but only re-sorts its list based on `writeTs`.

Data Flow: Hydrating the Cache and Handling Misses

Primary Data Structure:
The user's session list is a Redis `zset`.
- **Score:** `activeTs` (the timestamp for sync)
- **Value:** Session ID

Hot Path (Cache Hit)



The Session Lifecycle: Managing State Through Core Operations

All **write operations** on a user's session data are protected by a **distributed lock**. **This is critical** to ensure **atomic** updates to unread counts and prevent **timestamp rollbacks.**

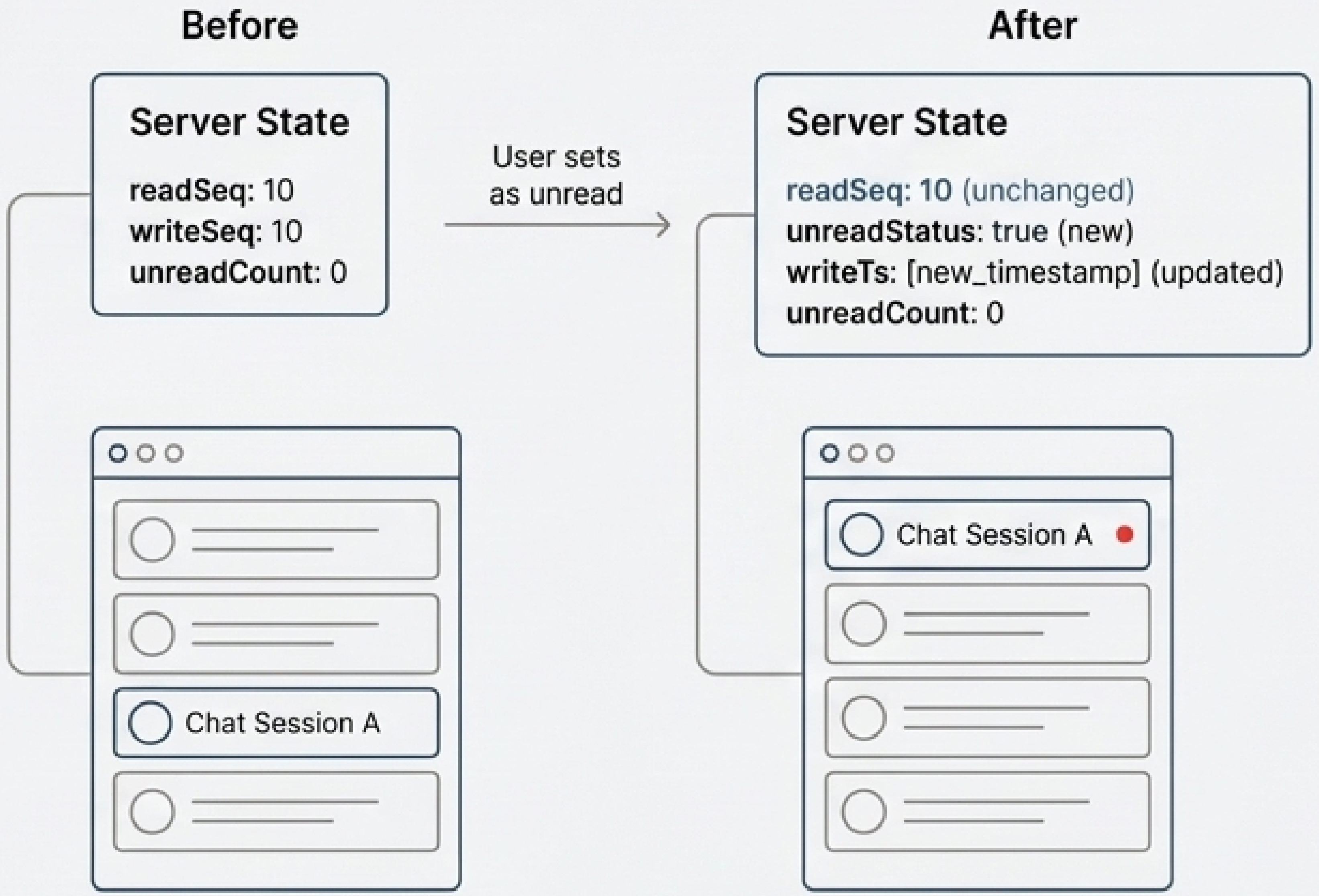
Operation	Key Server-Side Logic
Session Read	1. **Clear** session's `unreadCount`. 2. Update **total unread count.** 3. Set `readSeq` to match `writeSeq`. 4. Update `activeTs`. 5. Trigger multi-device `SyncSession` push.
Session Delete	1. Perform a **soft delete** by setting an `invalid` status flag. 2. **Clear unread counts.** 3. Update `activeTs`. 4. Trigger `SyncSession` push. *(Hard delete is supported but rare).*
Session Mute	1. Toggle `muteStatus` flag. 2. Atomically **add/subtract** session's `unreadCount` from total unread count. 3. Update `activeTs`. 4. Trigger `SyncSession` push.
Session Pin	1. Update `priority` field to a higher value. 2. Update **both** `activeTs` and `writeTs` to force re-sync and re-sort to the top. 3. Trigger `SyncSession` push.

A Deeper Look: The 'Set as Unread' Operation

The Challenge

A 'Set as Unread' action does not mean rolling back the server's read state.

The server cannot 'un-see' messages. The user is simply creating a personal reminder.



Resetting the State

The `unreadStatus` flag is automatically reset to `false` on the next user action (e.g., Read, Write, Mute), returning the session to its normal state.

The Implementation

Server Action

- The server does not revert the `readSeq`. Instead, it:
  - Sets a boolean flag: `unreadStatus = true`.
  - Updates the `writeTs` to the current time, forcing the session to the top.
  - Updates `activeTs` for synchronization.

Client Interpretation

- Sees `unreadStatus = true` and displays a simple red dot (no number).
- The updated `writeTs` ensures the session moves to the top.

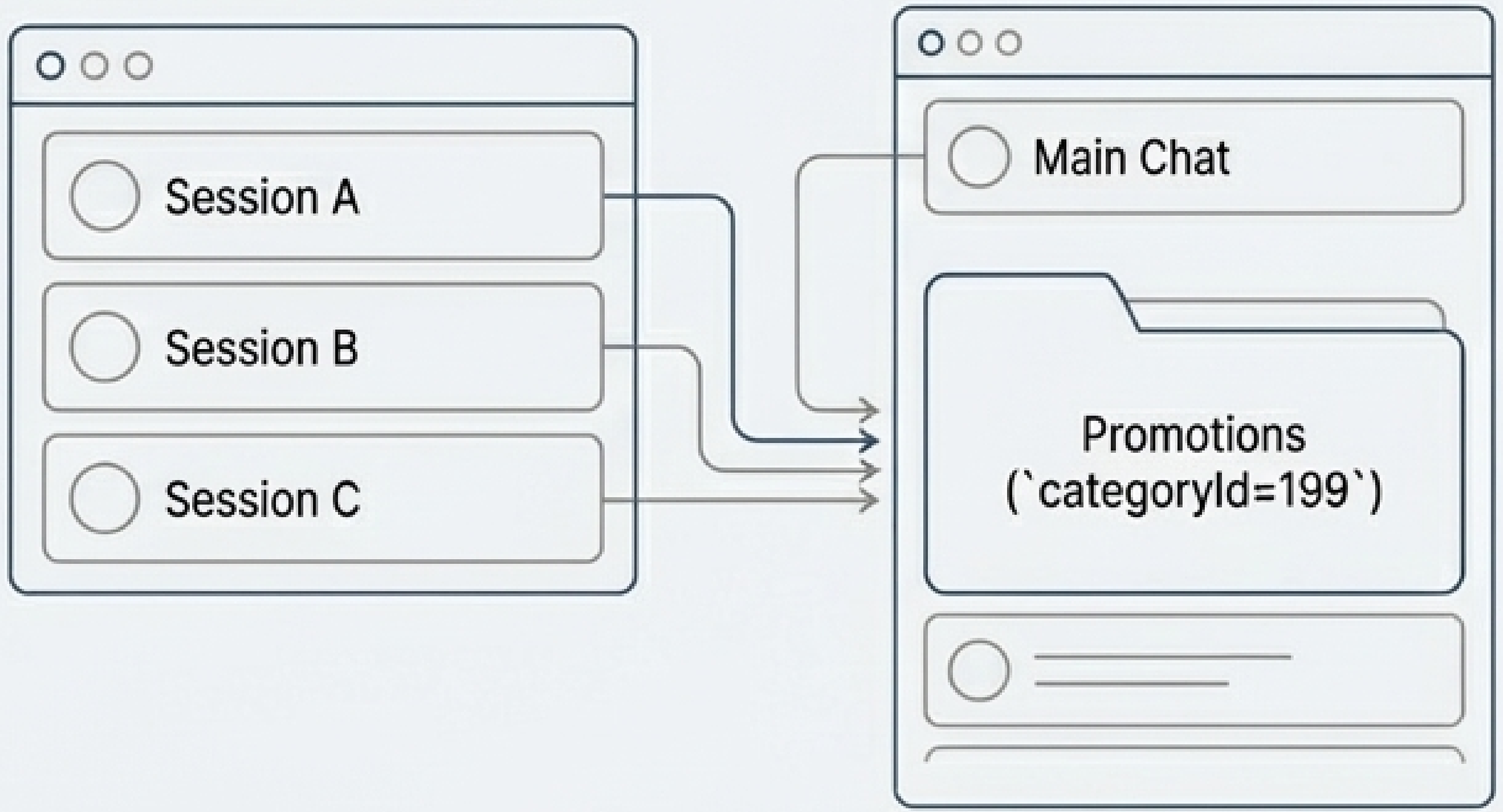
Taming Complexity: Aggregation and Hierarchical Sessions

The model extends beyond a flat list using two primary aggregation concepts to create folders, service accounts, and nested chat experiences.

Concept 1: `categoryId` Aggregation

Use Case: Grouping sessions into logical folders like “Promotions” or “Official Accounts”.

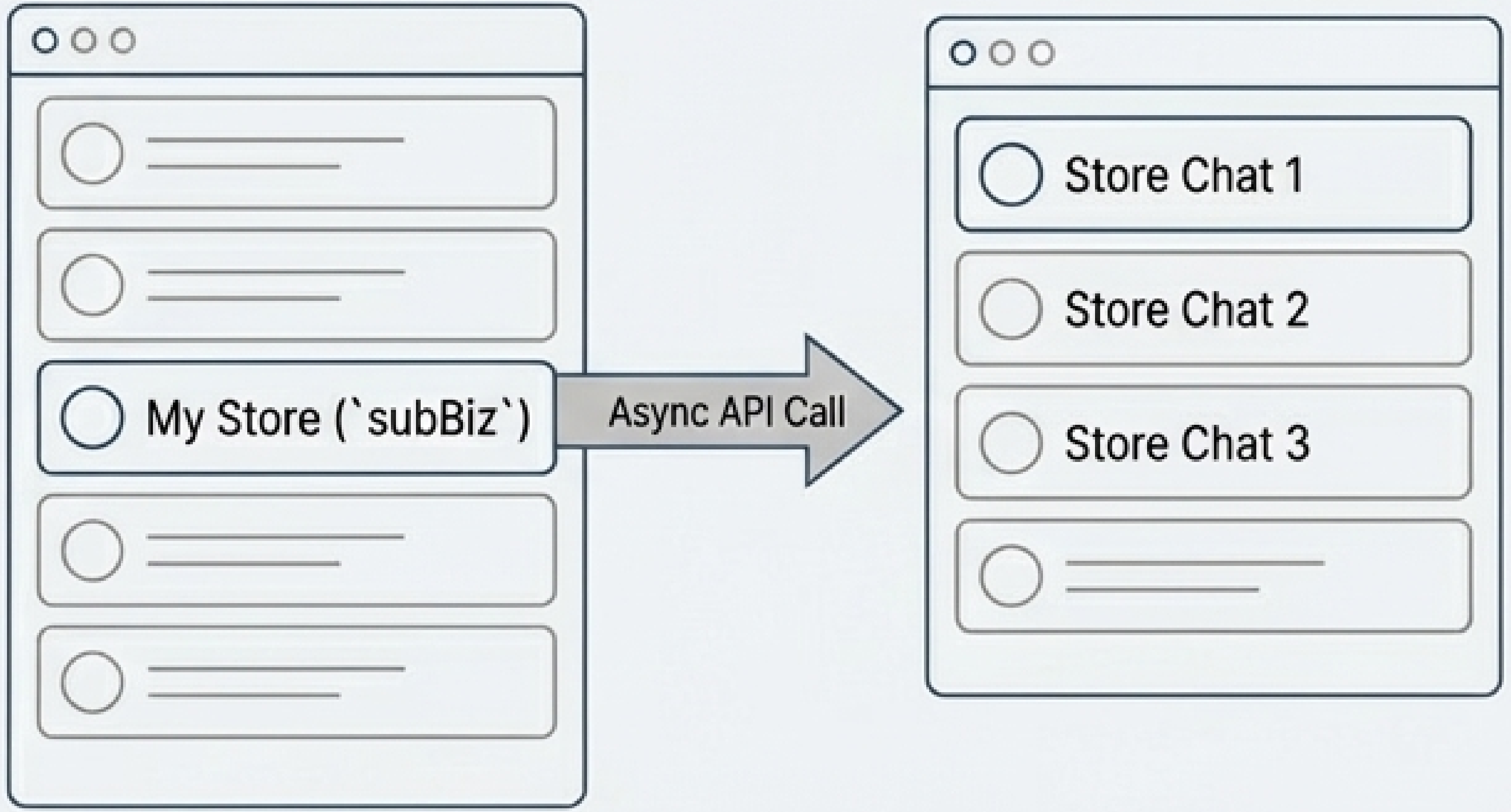
Implementation: Sessions are assigned a `categoryId`. The client UI can then filter and group them. The server may maintain a physical “aggregate session” to represent the folder itself.



Concept 2: `subBiz` Aggregation

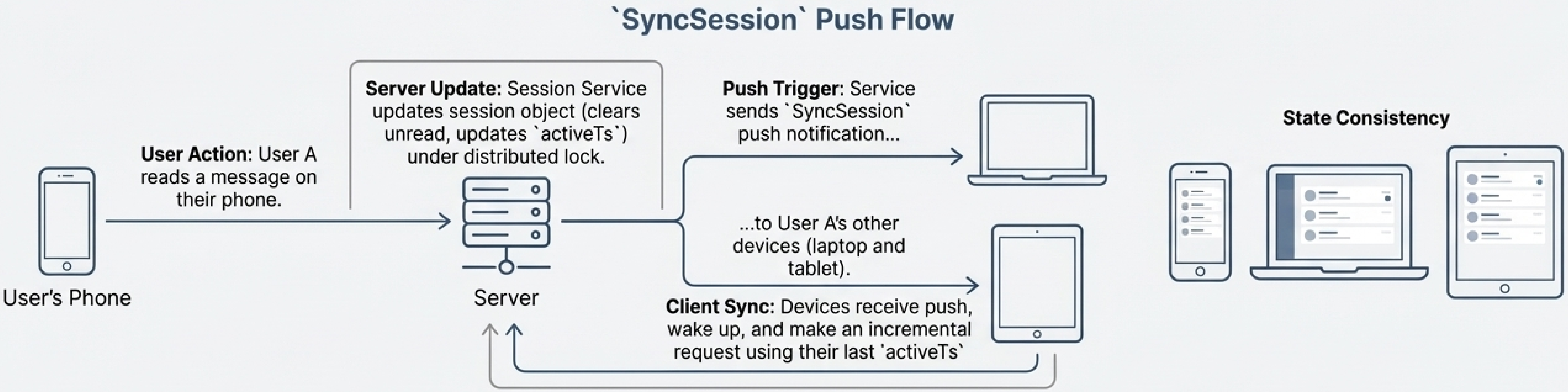
Use Case: More complex, multi-level nesting, such as embedding an e-commerce store’s chat list inside the main application.

Implementation: A `subBiz` session is a physical entity that acts as a gateway. The list of sessions within this context is fetched via a separate, asynchronous API call.



End-to-End: Multi-Device Synchronization in Action

State changes must be reflected across all of a user's logged-in devices. This is handled by a server-initiated push notification that prompts clients to fetch updates.



Operations that Trigger a `SyncSession` Push:

- Read / Clean
- Mute / Unmute
- Delete / Remove
- Set as Unread
- Pin / Unpin

**Note: New messages have their own push mechanism. `SyncSession` is specifically for session state changes.*

Validated at Scale: System Statistics

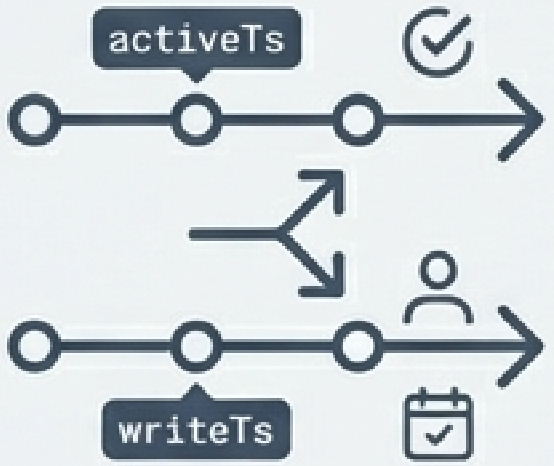
~100 Billion Total Sessions	**~28 TB** Database Size (single replica)	**100 / 10,000** DB Shards / Tables
15 Billion E-commerce Vertical Sessions	**43** User Group Count (P99)	**7,000** User Group Count (P999)

Core Architectural Principles Summarized

1. State, Not a Stream

Treat the session context as a first-class, overwritable entity. This simplifies state management and is more robust than parsing a message log.

2. Decouple Sync from Sort



Use a server-centric timestamp (activeTs`) for data consistency and a user-centric timestamp (`writeTs`) for intuitive UI sorting. This is the key to a non-jarring user experience.

3. Optimize for the Common Case

Design for fast, incremental syncs via a Redis `zset` index. Full table scans and DB loads are rare edge cases for inactive users.

4. Guarantee Atomicity

Use distributed locks for all write operations to ensure data integrity, especially for critical data like unread counts and sequence IDs.

The Session as a Contract

Ultimately, the Session Model is more than a data structure; it's a contract between the client and the server.