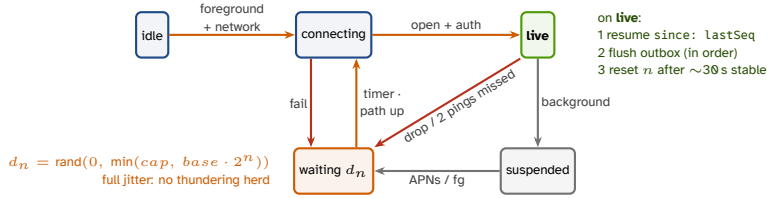
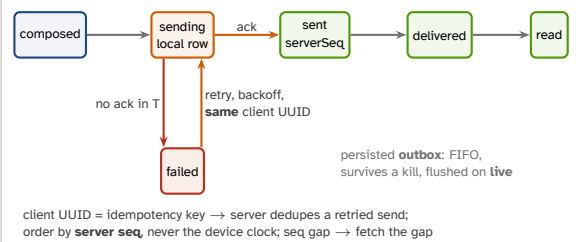


In one line: After the boxes-and-arrows, the interviewer picks one box and asks “how exactly?”. Score by naming the **mechanism**: the cache **key** and **eviction**, the reconnect **state machine**, the **idempotency key**, the **ordering authority**, the **conflict policy** and how you *detect* a conflict, the **cursor**, the flag’s **default when nothing was fetched**, and what leaves the device **privately**.

WebSocket lifecycle + reconnect



Message delivery + outbox



1 · Image cache

Tiers: memory NSCache of *decoded, downsampled* images → disk (bytes in Caches/) → network. Hits are hoisted up. **Key:** URL + target *pixel* size (+ transform); disk filename = SHA-256(URL) — never hashValue (seeded per run). **Eviction:** totalCostLimit with cost = w·h·4 bytes; NSCache also purges under memory pressure but is *not* strict LRU. Disk: byte cap + age, LRU sweep off-main; never Documents/ (backed up). **Downsample** with ImageIO to points × scale: a 4000 × 3000 photo is **48 MB** decoded, whatever its JPEG size. **Reuse:** cancel in prepareForReuse *and* check the cell still wants that URL. **Dedupe** in-flight by key; cancel the download only when its last observer leaves. Cap decode concurrency; prefetch at low priority, cancellable. **trap:** URLCache alone caches encoded bytes, not the decoded bitmap — decode still hits every scroll.

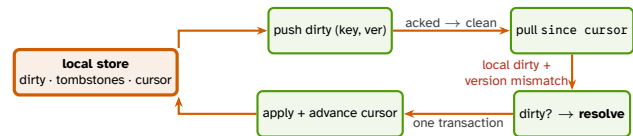
```
actor ImageLoader {
    private let memory = NSCache<NSString, UIImage>()
    private var inflight: [String: Task<UIImage, Error>] = [:]
    func image(_ url: URL, px: Int) async throws -> UIImage {
        let key = "\(url.absoluteString)@(\(px))"
        if let hit = memory.object(forKey: key as NSString) { return hit }
        if let running = inflight[key] { return try await running.value }
        let task = Task { try await diskOrNetwork(url, downsampleTo: px) }
        inflight[key] = task; defer { inflight[key] = nil }
        let img = try await task.value
        memory.setObject(img, forKey: key as NSString, cost: px * px * 4)
        return img
    }
}
```

2 · Chat / realtime

Transport: URLSessionWebSocketTask in the foreground — re-arm receive() after every message, sendPing every 25 s (cellular leaves half-open TCP), reconnect on NWPathMonitor. iOS kills sockets in the background → **APNs** wakes / notifies; reconnect + catch up on foreground. **Ordering:** per-conversation **server seq**; pending rows sort locally until acked. **Idempotency:** client UUID per message; server returns the existing row on a repeat; client dedupes incoming by id. **Outbox:** persisted FIFO (survives kill), flushed in order on **live**; after N failures → **failed** + tap to retry (no poison-pill blocking the queue). **Receipts:** events keyed by message id, batched per screen. **Typing:** ephemeral, debounced start/stop, TTL 5 s, never stored. **trap:** “sent” only proves the ack — a missing since cursor on reconnect is the usual lost-message bug.

3 · Offline-first sync

Local store is truth; UI observes it; writes are local + optimistic. Client-generated UUIDs → create offline. **Change tracking:** dirty flag + updatedAt (simple, loses intent) or an **oplog** (ordered ops, richer merges). **Cycle:** push dirty ops (idempotent: set with expected version, not *increment*) → clear dirty per **acked record** → pull changes?since=cursor → apply → advance cursor in the same transaction. Apply rule: absent → insert · local clean → fast-forward · **local dirty** → **conflict**. **Detect** with a server version / ETag (If-Match → 412), or version vectors — not a clock. **Resolve:** *server-wins* (simplest, drops local edit) · *LWW* (silently drops the loser; skewed clock always wins) · *field-level merge* · *keep both* + *prompt* · *CRDT* (collab text, converges in any order). **Deletes = tombstones**, GC after a window (a device offline longer can resurrect). **Clock skew:** device time is a display hint; order by server revision or a hybrid logical clock. Triggers: foreground, path restored, BGAppRefreshTask, silent push; short + resumable.



4 · Pagination

offset ?page=3
live inserts shift rows → **dupes / gaps**
jump to page yes
DB cost scans OFFSET rows
cursor ?after=opaque
stable (keyset WHERE id < c)
no — sequential only
index seek

Prefetch threshold: fetch when a row ~5–10 from the end appears (or prefetchItemsAt); guard !isLoading, nextCursor != nil; dedupe by id; nextCursor == nil = end. Chat needs *both* directions (before: history, after: catch-up). Expired cursor → refetch from the top.

5 · Feature flags + remote config

Resolve in one provider: debug override → targeting → remote (**cached last-known-good** on disk) → **baked default**. First launch / offline = the default, so every flag’s default must be safe. Fetch on launch but **activate at a safe point** (next launch / screen entry) — no mid-screen flips. **Kill switch:** remote, no App Review; its cached value must persist so an unreachable server does not revive the broken path. **Rollout:** hash(userID + flagKey) % 100 < pct — stable bucket, ramp 1→10→100 % watching crash-free rate. Client flag = UX, **not** security; entitlements are enforced server-side. Test *both* branches; every flag gets an owner + expiry.

6 · Analytics / logging pipeline

Typed events (enum, schema-versioned) + context (session, app version, event UUID) → in-memory buffer → **persisted queue** (SQLite / file) → **batch** upload every N events or T seconds and on background (beginBackgroundTask), gzip, backoff; server dedupes by event UUID. Cap the queue, drop oldest; sample high-volume events. **Privacy:** no PII in events (hash / drop), consent gate before sending, ATT only for cross-app tracking, declare data in PrivacyInfo.xcprivacy; Logger interpolations are .private by default for dynamic strings. Crashes + hangs: MetricKit.

Interview traps

- ! Reconnect with a fixed delay — a server blip becomes a synchronized reconnect storm.
- ! Ordering, or LWW, on device timestamps.
- ! Batch sync all-or-nothing — one bad record blocks the queue forever.
- ! Deleting rows locally instead of tombstoning — the delete never syncs.
- ! Offset pagination on a live list; loading only when the last row is visible.

Remember: key + eviction · state machine + jittered backoff · UUID idempotency · server seq · cursor · version *detects*, policy *resolves* · safe default.

Likely questions

1. Wrong image in a cell? — late completion after reuse: cancel + URL check.
2. Message sent twice after a timeout? — lost ack; client UUID, server dedupe.
3. Two devices edit the same note offline? — version mismatch → named policy → both converge.
4. Flag value at first launch? — the baked default (then the cached one).