Updater V2 Design

Based on review of `crates/catchup_worker` as it stands on the `updater` branch.

Summary of what works and what doesn't

The catchup control plane is in good shape: durable segments, cursor-based resume, dead-letter tracking, structured retries, and reasonable test coverage. The realtime path is a stub. `SyncService::realtime_update` spawns legacy `firebase_worker` tasks in `WorkerMode::Updater`, drops the handles immediately, and the whole thing runs sequentially after catchup in `main.rs`. It will not survive a production workload.

Issue 1: MaxID is not a freshness cursor

`/maxitem.json` tells you the highest ID ever assigned. It catches new items. It does not catch:

→ Score changes on existing stories (the primary mutation HN items experience)
→ `descendants` counter updates as comments are added
→ `dead` / `deleted` flag changes
→ New `kids` entries added to an existing story or comment

The Firebase SSE stream at `/updates.json` pushes batches of item IDs whenever *any* of these mutations occur. This is the right mechanism — it's designed for exactly this purpose. The old brute-force (pull last 1M items) was compensating for not having the SSE path working correctly. With a working SSE path the question becomes: what do we do about mutations missed during SSE downtime?

The disconnect problem: when the SSE connection drops, we lose a window of update events. We know the time span of the outage but not which items were mutated. The only backstop is a periodic rescan of items likely to still be changing.

Do we still need the 1M-item brute force? No, but we do need a periodic rescan. The 1M-item window is the wrong unit. At current HN velocity (~130k items/day), 1M items is about 7-8 days. Most item mutation happens in the first 24-48 hours after creation (front page lifetime). An empirically-tuned time-based window is both cheaper and more principled.

Proposed approach: rescan items where `items.time >= now() - interval 'N days'`, not "last N IDs". N should be derived from data. Until we have data, 3 days is a reasonable conservative default.

Measuring the right window: add `last_fetched_at TIMESTAMPTZ` to the `items` table. When the realtime update consumer processes an event for item X, record `now() - to_timestamp(X.time)` as the update age. A histogram of this over a week of operation directly gives you the empirical p95/p99 mutation window. Set N accordingly. This is cheap to add now and will answer the question correctly rather than guessing.

Issue 2: Catchup and realtime must be concurrent

The current `main.rs` flow is:

```
catchup() → wait for completion → start realtime listeners
```

This is wrong. On a fresh install, catchup takes ~5 hours. During those 5 hours: - New items are being created at ~150/hour - Existing items are being scored and commented on - The SSE stream is not being consumed at all

The fix is straightforward: run catchup and the SSE listener as concurrent tokio tasks under a shared `CancellationToken`. Both share the global rate limiter so they don't stack additively against Firebase's limits.

Priority: give realtime update workers slightly higher token priority than catchup workers, or simply let them share the limiter equally and accept that catchup will be slightly slower when updates are bursty. The simpler approach (equal sharing) is fine for a personal archive.

Issue 3: Process architecture

Keep `catchup_only` as-is. It is a clean, well-wired entrypoint with good CLI ergonomics and test coverage. It serves the bulk-ingest use case. Don't merge it into the service binary.

Fix `main.rs` to be the updater service. The service binary should do exactly four things concurrently:

1. **SSE listener loop** — connects, consumes update events, pushes item IDs to a bounded channel, handles reconnection with gap repair.
2. **Update ingest loop** — drains the update channel, fetches items, upserts. Should use `IngestWorker` (not the legacy `firebase_worker`).
3. **Incremental catchup loop** — continuously runs `CatchupOrchestrator::sync()` against `[frontier+1, maxitem]` with a short sleep when no new segments are found. This handles new items appearing while the service runs.

4. **Periodic rescan loop** — on a configurable interval (default: every 6 hours), scans `items.time >= now() - N days` and re-upserts everything in the window. This is the correctness backstop for missed SSE events.

All four loops share one `CancellationToken` for shutdown. All four share the global rate limiter. The service exits if any loop dies with a non-retryable error; systemd restarts it.

The `catchup_only` vs service entrypoint split is not a hack — it's the right model. `catchup_only` is a job. The service binary is a daemon. They can and should run simultaneously during initial ingest: `catchup_only` brings the historical corpus up to date while the service handles the live edge.

Issue 4: Multi-worker

For a single-server personal archive: don't build multi-worker in v2.

Reasoning: - The database and the worker are co-located. An outage that takes one down takes both down. Multi-box failover doesn't buy much here. - The correctness model (periodic rescan + SSE gap repair) already handles the worker dying and restarting. A brief outage just widens the rescan window. - Operational complexity of running two workers correctly (split-brain on segment claims, duplicate upserts, leader election) is not worth it for a mirror workload where "slightly stale" is acceptable.

The path to multi-worker is one SQL change. Currently `claim_next_pending_segment` does:

```
UPDATE ingest_segments SET status = 'in_progress' ...
WHERE segment_id = (SELECT segment_id FROM ingest_segments WHERE status = 'pending' ... LIMI
```

Adding `FOR UPDATE SKIP LOCKED` to the subquery makes this safe for concurrent claimants without any other changes. Do this now even if multi-worker isn't deployed — it costs nothing and future-proofs the schema.

Redis is unnecessary. The `ingest_segments` table *is* the work queue. Postgres advisory locks (`pg_try_advisory_lock`) are sufficient for single-active semantics if you ever want a hot standby.

SSE reconnection design

The current `listen_to_updates` exits with `Ok(())` when the stream ends. This is the core of WP4 from v1-gaps.md and it needs to be fixed.

The reconnection loop needs to do three things correctly:

1. Reconnect with backoff. Wrap the connection loop in an outer retry loop. Use exponential backoff (500ms → 1s → 2s → 4s → cap at 30s) on repeated failures. Reset on successful event receipt.

2. Gap detection on reconnect. Before connecting, call `get_max_id()` and store it as `max_id_at_connect`. Before disconnecting (or immediately after reconnect), call `get_max_id()` again. If `max_id_post_reconnect > max_id_at_connect`, the range `(max_id_at_connect, max_id_post_reconnect]` contains items that were created during the blind window. Enqueue these IDs directly to the update channel (or create catchup segments for them if the range is large). Note that this only covers *new* items, not mutations to existing items — that's what the rescan loop handles.

3. Last-event tracking. Record a `last_event_at` timestamp on every successful SSE event. This drives the `realtime_last_event_age_seconds` metric (see monitoring section). A stale last-event timestamp is the primary signal that the listener is broken.

Update channel design

Replace `flume::unbounded` with `flume::bounded(4096)` or similar.

When the channel is full, the SSE listener will block on `send_async`. This is intentional backpressure — if the consumer can't keep up, the listener slows down. The risk is that Firebase times out the SSE connection if we stop reading from it. If that happens, we reconnect, gap detection fires, and the missed items get queued as a catchup range. This is acceptable.

Do not silently drop items from the channel. If you drop, you permanently miss that update until the next rescan cycle.

The update ingest loop should deduplicate in-flight IDs. The SSE stream can push the same item ID multiple times in rapid succession (score changes cluster). A `HashSet<i64>` tracking IDs currently in the channel or in-flight in workers is sufficient. If an ID is already queued or being processed, skip it. This is an optimization, not a correctness requirement (upserts are idempotent).

Realtime ingest worker

Replace the legacy `firebase_worker::worker(mode=Updater)` with something that uses `IngestWorker` primitives. The legacy path: - Has the double-count metric bug (`records_pulled` incremented twice per item) - Does no batching (uploads every item individually) - Has no

retry logic (errors propagate and kill the task) - Is not supervised (handles are dropped)

For realtime, the semantics are slightly different from catchup: - Items come one at a time from the SSE stream, not in sequential ID ranges - There is no segment state machine to update - Retries should be per-item, bounded (3-5 attempts), with short backoff

A simple supervised loop per worker is sufficient:

```
loop {
    id = recv from channel
    fetch with retry (3 attempts, 200ms/500ms/1s backoff)
    upsert
    emit metric
}
```

Spawn N workers (configurable, default 8 is plenty — updates are not CPU-bound, the bottleneck is Firebase API rate limits). Hold the `JoinHandle`s. If a worker exits with error, log and restart it after a short delay. If all workers die, exit the process.

Periodic rescan loop

This loop provides the correctness backstop.

Trigger: every 6 hours (configurable). Also trigger immediately after a long SSE outage (>5 minutes).

Window: items where `to_timestamp(items.time) >= now() - interval '3 days'` (configurable N). This is a time-based filter, not an ID-count filter.

Mechanism: query `SELECT id FROM items WHERE time >= ...` and feed the IDs into the update ingest path (not through the SSE channel — use a separate ingest path or create catchup segments for the range). Re-using `CatchupOrchestrator` for this is clean: compute the ID range corresponding to the time window (lowest and highest ID with `time >= cutoff`), enqueue as catchup segments with `force_replay_window=true`. The existing segment machinery handles batching, retries, and rate limiting correctly.

What this catches: - SSE events missed during disconnects - Items that were changing at the moment of initial catchup ingest (score was lower, some comments hadn't appeared yet) - Items where the SSE event was delivered but the worker failed to process it

What this does not catch: mutations to items older than N days that HN is still pushing SSE events for. This is the long tail. The empirical measurement (via `last_fetched_at`) will tell you if N=3 is sufficient or if you need N=7.

Monitoring

These are the signals you actually need. Everything else is noise.

Service health (page on these)

Metric	Type	Alert threshold	What it means
`realtime_listener_connected`	Gauge (0/1)	< 1 for > 60s	SSE connection is down
`realtime_last_event_age_seconds`	Gauge	> 300s	Keepalives stopped; connection may be zombie
`realtime_worker_alive_count`	Gauge	< 1	All update workers are dead
`catchup_dead_letter_segments_total`	Counter	rate > 0	Fatal failures in catchup/rescan

Lag / correctness (ticket on these)

Metric	Type	Alert threshold	What it means
`catchup_frontier_lag` = `maxitem - frontier_id`	Gauge	> 50000	Falling behind on new items
`realtime_queue_depth`	Gauge	> 2000	Update consumer can't keep up
`realtime_reconnects_total`	Counter	rate > 6/hour	SSE stability problem
`rescan_last_run_age_seconds`	Gauge	> 43200 (12h)	Rescan isn't running

Diagnostics (no alert, just dashboards)

→ `realtime_items_updated_total` — throughput of realtime path

→ `realtime_update_age_seconds` histogram — empirical p50/p95/p99 of SSE event age (drives rescan window sizing)

→ `rescan_items_processed_total` — volume of rescan work

→ `rescan_items_changed_count` — items that differed from what we had (rescan effectiveness indicator)

→ `catchup_progress_percent` — how far through historical ingest we are

The `rescan_items_changed_count` metric is the key signal for tuning N. If this number is near zero, N is too large (wasted work). If it's consistently nonzero but you're not seeing corrections, N may be too small. Compare against `realtime_update_age_seconds` p95 to calibrate.

Implementation order

These are roughly sequential, each buildable in isolation:

1. **Fix the SSE listener** (`firebase_listener/listener.rs`): outer reconnect loop, backoff, gap detection, `last_event_at` tracking. No downstream changes needed; existing caller interface can stay the same for now.
2. **Fix update worker supervision** (`sync_service/mod.rs`): hold handles, restart on exit, replace `flume::unbounded` with `flume::bounded`. Don't touch the legacy worker logic yet — just wrap it properly.
3. **Concurrent catchup + realtime in main.rs**: run both as `tokio::spawn` tasks under shared `CancellationToken`. This is mostly a `main.rs` restructure.
4. **Port update workers to** `IngestWorker`: replace legacy `firebase_worker(mode=Updater)` with a per-item retry loop using the existing `IngestWorker` primitives. Fix the double-count metric.
5. **Incremental catchup loop**: add a `loop { sleep(15s); orchestrator.sync(...) }` task that keeps the frontier advancing. This replaces the one-shot catchup call in `main.rs`.
6. **Periodic rescan loop**: add the scheduled rescan task using the existing segment orchestrator with `force_replay_window=true` over the N-day time window.
7. **Observability**: add the metrics above. Add `last_fetched_at` column to `items`. Wire up rescan effectiveness tracking.
8. **FOR UPDATE SKIP LOCKED**: small change to `claim_next_pending_segment`. Do this early (step 1 or 2) since it's a correctness fix, not a feature.

Items 1-3 fix the most critical correctness and supervision gaps. Items 4-6 bring full correctness. Items 7-8 are polish.

What not to build in v2

→ **Redis**: unnecessary. `ingest_segments` + Postgres is the queue.

→ **Multi-active workers on separate hosts**: the availability gain is marginal for co-located DB + worker. The `FOR UPDATE SKIP LOCKED` fix (step 8) is the preparation needed if this changes.

→ **Run-level audit table** (`ingest_runs`): a `started_at` on segments is enough lineage. Add later if you actually need it.

→ **Proactive staleness scanning beyond the rescan window**: if an item is 30 days old and gets an unexpected update, the next daily rescan won't catch it. This is an acceptable

miss for a personal archive. The SSE stream should fire for it anyway.

→ **Leader election / advisory locks**: if you're running one process on one box, this complexity buys nothing. Add it if you ever add a warm standby.