SLICK ENGINEERING & CONSULTING, INC. · FREE FIELD GUIDE

The Self-Hosted Evernote Replacement Playbook

How I moved 12,690 notes to Joplin on a Mac mini, built a local AI that answers questions about them, proved nothing was lost — and cancelled a $250/yr subscription.

Corey Slick

RESEARCH ENGINEER · SLICK-ENGINEERING.COM · JUNE 2026

CONTENTS

What's in this playbook

A note on scope: this is a field guide, not a copy-paste install script. It documents the decisions, the failure modes, and the architecture of a real single-user system I run every day — enough that you could rebuild it, adapt it, or decide it's not worth it. Security specifics (hostnames, tokens, ports) are deliberately kept at the architecture level.

§ 01 · MOTIVATION

Why I left Evernote

Fifteen years of lock-in, a price that only moved one direction, and a search box that never worked.

For about fifteen years, my working memory lived in Evernote: research notes, a daily journal, scanned documents, medication logs, clipped articles. By the end it was **12,690 notes across 277 notebooks** — roughly 60,000 attachments and well over 10 GB. That kind of archive is not something you idly migrate, and the pricing reflected exactly that.

I was on the Advanced tier (formerly "Professional"). When Bending Spoons restructured the plans in late 2025, my tier jumped roughly **$80/year**, landing me near **$250/year** — with nothing about the product improved for me. The trajectory was clear: more gating, higher prices, and a search box that had disappointed me for years.

So I decided to leave before the archive got any bigger. But I had hard requirements, and they shaped every decision that followed:

- **Apps on iOS, macOS, and Android.** I capture on the phone, write on the Mac, and read everywhere.
- **AI search that actually works — on every device, including the iPhone.** This was my number-one need. "Ask my notes a question and get a real answer" simply didn't exist in Evernote.
- **Preserve my colored highlights, 100%.** I color-code heavily; losing the highlights would have made years of notes harder to use. Non-negotiable.
- **PDFs, voice notes with transcription, and a Home/scratchpad surface** like the one Evernote put front and center.
- **Keep my existing daily-journal pipeline** — an automation that emails entries into a notebook every morning — working against whatever replaced Evernote.

Everything in this playbook is downstream of that list. The interesting engineering wasn't "set up a note app." It was satisfying those five constraints — especially AI search on iOS and 100% highlight fidelity on a 15-year archive — without signing up for a second subscription or a permanent maintenance burden.

§ 02 · STRATEGY

Build vs. buy: the actual decision

Adopt the hard 90% that already exists. Build only the high-leverage 10% that doesn't.

I weighed three paths:

1. **Buy a lifetime app** (UpNote, ~$40 one-time). Cheap and done — but I'd be trading one closed box for another, with no AI story.
2. **Self-host open-source Joplin.** Own the data and the backend, $0 recurring, but I'd have to build the parts Joplin lacks.
3. **Build the whole thing from scratch.** Total control, and a multi-month detour to reinvent an editor and a sync engine that already exist and are battle-tested.

I chose the middle path, with a principle that shaped everything after: **adopt vanilla Joplin for the hard 90%, and build only the high-leverage 10%.** Joplin already nails the genuinely hard part — a real Markdown editor, a robust sync engine, offline-first behavior, and three mature client apps. Reinventing that would be foolish. What it doesn't do well is exactly the 10% I cared about most.

Why not just fork the Joplin apps?

The obvious move is to fork Joplin's clients and bolt your features on. I rejected that fast. Joplin ships two to three releases a month touching the editor, sync, and import paths; maintaining a fork across three platforms is a permanent tax — re-signing and re-releasing the iOS build on every one of those releases, forever.

There was one hard wall pushing me toward forking anyway: **the stock iOS app only installs Joplin-team "recommended" plugins, so the community AI-search plugin can't run on the iPhone** — the device I carry everywhere, for the feature I wanted most.

> THE INSIGHT THAT DISSOLVED THE PROBLEM
>
> The AI work — embedding a 10 GB corpus, running a model — wants to run on an always-on machine *regardless*. So instead of forking anything, I built the AI as **one small server-side companion service** on the Mac mini, reachable from every device (including the iPhone) over my private network. That sidesteps the iOS plugin gate entirely, leaves all three Joplin apps stock and auto-updating, and gives me a single implementation to maintain instead of three forks.

That decision — vanilla clients everywhere, plus one companion service that owns the custom 10% — is the spine of the whole system. Keep it in mind through the rest of the playbook; almost every "how did you do X on iOS without a plugin?" answer is "the companion service does it server-side."

§ 03 · INFRASTRUCTURE

The architecture

Deliberately boring infrastructure — which is exactly what you want underneath fifteen years of notes.

The backend runs on a Mac mini I already keep on 24/7. Four pieces do the work:

```
# the shape of it
iPhone / Android / Macs  ──sync──▶  Joplin Server (Docker + Postgres)
                                          ▲   on the always-on Mac mini
every device  ──▶  Companion service ─────┘   (reads Joplin's Data API)
                     ├─ ask-my-notes (local embeddings + Claude)
                     ├─ voice transcription (local Whisper)
                     ├─ Home / scratchpad / quick-capture
                     └─ large-attachment streaming
```

1 · Joplin Server (the sync hub)

`joplin/server` + `postgres:16` run in Docker (via colima) and start themselves on boot. All three stock clients — iPhone, Android, and the Macs — sync to it. Sync is a *core* Joplin feature, not a plugin, so there's no desktop-only limitation: every client syncs to the same server.

2 · Tailscale for remote access (with a real certificate)

Remote access goes through Tailscale, which issues a genuine, auto-renewing Let's Encrypt certificate for the server. This is not a nicety. **iOS won't trust a self-signed certificate and gives you no "ignore TLS errors" escape hatch**, and Android's equivalent toggle is buggy with Joplin Server. A real public-CA certificate was the only reliable path to the phones — and Tailscale provides one with no public attack surface at all.

> SECURITY POSTURE
>
> The system is **private-network-only**: nothing is exposed to the public internet, no ports are forwarded, the database is never published, and the only way in is my own tailnet. At rest, the mini runs FileVault. Off-site backups are client-side encrypted before they leave the machine (§08). SSH to the mini is key-only.

3 · The always-on desktop app (OCR + the Data API)

One Joplin *desktop* app on the mini runs as the workhorse. It does two jobs the server can't: it generates **OCR** (a desktop-only feature — it rebuilds a searchable text layer over scanned PDFs and images, since Evernote's text layer isn't portable), and it exposes Joplin's local **Data API**, which the companion service reads from. Everything custom is built on top of that read-only Data API.

4 · Offline-first as a safety net

Because Joplin is offline-first, every device holds a complete local copy. If the mini reboots, sync just pauses — no data is ever at risk on the clients. On the phones I keep attachment download on *manual* so they never pull the full multi-gigabyte library; the large files Joplin's mobile apps refuse to sync are served on demand instead (§07).

§ 04 · MIGRATION

The migration nobody warns you about

Moving 12,690 notes with fidelity was the hardest engineering in the whole project.

Evernote exports to ENEX files; Joplin imports ENEX. In theory, done. In practice, the off-the-shelf path quietly destroys data:

- **Joplin's native ENEX importer aborted on malformed anchor tags and silently lost ~11% of my notes** in testing. An 11% data-loss rate on an irreplaceable archive is a non-starter.
- It also **mangled highlight color spans** — and preserving highlights was a hard requirement.
- The popular third-party converter (Yarle) **drops highlight colors too**, and produced title collisions.

So I wrote my own **ENEX → Markdown converter**. Four things made it work:

- **A tolerant streaming parser.** Some per-notebook ENEX files are large enough to blow past Node's ~512 MB single-string limit, so the converter streams rather than reading whole files into memory.
- **Highlight detection.** It finds Evernote's highlight markers and emits inline colored spans (`<span style="background-color:#hex">`) that render correctly on every platform — all five highlight colors intact.
- **Date preservation.** Real created and modified timestamps carry over, not the import date. (Sorting fifteen years of notes by "when I actually wrote this" is the difference between a usable archive and a pile.)
- **Graceful fallback.** On any conversion edge case it preserves the raw HTML, so nothing is ever dropped for content it can't cleanly convert.

The result:

12,690	277	~60k
NOTES IMPORTED · 0 ERRORS	NOTEBOOKS · 26 STACKS	ATTACHMENTS

5 / 5
HIGHLIGHT COLORS PRESERVED

Colored highlights were preserved across all 20 highlighted notebooks, and created/modified dates came through intact. After import, OCR re-runs on the mini to rebuild the searchable text layer over scanned documents.

> HARD-WON LESSON
>
> **Run destructive import tests in a throwaway profile.** An early test against my real sync polluted it with ~50,000 deletion records and forced a clean server reset. A 53,000-resource library does not forgive mistakes quickly — sandbox the experiment.

§ 05 · VERIFICATION

Proving nothing was lost

Migrating an archive is one thing. Trusting it enough to cancel the subscription is another.

Before I cancelled anything, I audited the migration. I wrote a script that compared **all 12,690 notes** — the original Evernote content, decompressed straight out of Evernote's own backup database, against what actually landed in Joplin — using a normalized word-and-sentence diff. The results:

- **Zero notes lost.** Joplin actually ended up holding three *more* than Evernote; the handful of apparent title "misses" were just notes I'd renamed or moved over the years.
- **607 notes showed content differences — almost entirely web clips and saved emails.** Converting a captured web page's HTML into clean Markdown is inherently lossy (comment threads, navigation chrome, ad boilerplate). The pristine originals are preserved in the off-site archive regardless.
- **Exactly two hand-typed notes had real text drops.** Both traced to two bugs in my converter's handling of deeply nested lists. I fixed both and repaired the two notes back to 100%.

Two notes out of 12,690, both recovered. Combined with a full export of the original Evernote data archived to encrypted off-site storage, that was enough confidence to pull the plug.

> THE TWO CONVERTER BUGS (AND THE RULE THEY TAUGHT)
>
> Both bugs *silently dropped text*, which is the worst kind. (1) Evernote encodes a nested list as a *sibling* of its parent `<li>` — invalid HTML that the Markdown converter skips. (2) A custom rule that returned raw HTML caused the converter to mis-serialize unrelated blocks in large documents. The durable lesson: **never return raw HTML from a Turndown rule — emit a placeholder and restore it after conversion finishes.** A handful of 4–5-level-deep nested lines that the converter still couldn't place inline were appended verbatim under a labeled section, so no text was ever lost.

> IF YOU TAKE ONE THING FROM THIS CHAPTER
>
> **Export and verify before you cancel.** Keep the original export archived off-line, and diff the migrated copy against the source *programmatically* — eyeballing a few notes is not verification of fifteen years.

§ 06 · THE 10% I BUILT

The AI companion: "Ask My Notes"

The centerpiece — and the reason the whole project was worth doing.

The companion is a small service on the mini. The pipeline is simple to describe and does a lot:

```
notes ──chunk──▶ Ollama (nomic-embed-text) ──▶ sqlite-vec  (local vector store)
query ──▶ semantic search + SQL-over-metadata + full-note read ──▶ Claude (ZDR) ──▶ cited answer
```

- It pulls notes through the read-only Data API, chunks them, and **embeds them locally** with Ollama (`nomic-embed-text`), storing vectors in a single `sqlite-vec` file. The indexer is **incremental and OCR-aware** — it only re-embeds what changed.
- **Embeddings run locally on purpose.** The corpus includes a personal journal with medication logs; that content never leaves my own hardware to be indexed.
- Answers come from **Claude through a zero-data-retention endpoint**, and **always link back to the source notes** they're drawn from.
- It's **agentic, not just similarity search.** The model has three tools: semantic search, read-only SQL over a notes-metadata table, and full-note read. That combination means it answers content questions *and* arbitrary aggregations ("how many notes are in my daily journal?" → 158) *and* counts inside a single note ("how many injections did I log in May?" → 25). A plain vector search can't do the last two.

The front end is three native apps, not a web page

One shared SwiftUI codebase targets both iOS and macOS; a Jetpack Compose app covers Android. All verified on real devices. The pieces that matter:

- **Ask is a real multi-turn chat.** The client owns the conversation transcript and posts the message history with each turn, so the server stays completely stateless.
- **Tappable citations** open an in-app note reader that renders the note's HTML in a WebView — so the migrated colored highlights actually display in color.
- **Client-side sorting** on the *real* Evernote created/modified dates (relevance, modified, created, title).
- **Bearer-token auth**, with the token kept in the Keychain / encrypted storage.

> BONUS: THE SYNCED FAVORITES BAR JOPLIN NEVER HAD
>
> Joplin has no native synced or mobile favorites — the desktop plugin is per-machine and doesn't sync. I migrated my exact 54 Evernote shortcuts and built a **Shortcuts tab** into the apps that reads *and writes* a synced "⭐ Shortcuts" note through the Data API: add, remove, and reorder notes *and* notebooks on any device, syncing everywhere. The "make the companion own a synced note, and let every client edit it" pattern turned out to be a clean way to add cross-device features Joplin itself lacks.

§ 07 · ERGONOMICS

Voice notes, a real Home, and the files phones choke on

The small things that make a tool feel finished instead of merely functional.

Voice notes that transcribe themselves

A background worker on the mini watches for new audio notes and transcribes them with a local Whisper model (`mlx-whisper`, `large-v3-turbo`, on the Mac's GPU). It appends a labeled `🎙️ Transcript` section into the note body, so the text appears everywhere the note does *and* gets picked up by the same search index — meaning I can find a voice memo by what I actually said in it. It runs locally for the same reason the embeddings do (some recordings are personal), and it transcribed all 153 of my existing voice notes on its first run. It's device-agnostic: it doesn't matter which phone or Mac recorded the audio.

A Home that actually feels like Evernote

Joplin has no Home surface on mobile, which is exactly where I missed it most. The companion app got a native **Home tab**: a synced scratchpad that autosaves as you type, one-tap quick-capture into an auto-created `📥 Inbox` notebook, and a live "recent notes" list. It's the cross-platform landing pad Evernote always had and stock Joplin never did.

The big files phones can't sync

Joplin's mobile apps refuse to sync any single attachment over 100 MB, so a handful of large PDFs and a couple of videos were simply invisible on my phone. The companion serves them on demand instead: a per-note attachments list plus a global "large files" browser, backed by a range-aware streaming proxy on the mini. Tap a file and it downloads and opens in the native viewer (QuickLook on iOS and macOS, the system viewer on Android). Nothing big ever downloads unless I ask for it.

§ 08 · RESILIENCE

Backups, and what happens if the mini dies

A self-hosted system you can't restore is a liability, not an upgrade.

Every night, the Joplin database is dumped and pushed to **Cloudflare R2 through restic**, **client-side encrypted before it ever leaves the machine** (~11 GiB stored), with 7-day / 4-week / 6-month retention and a weekly integrity check. The job is **self-healing**: it probes the destination first and repairs a stale network tunnel before giving up, rather than silently failing at 3 AM. And — the step most backup setups skip — **I actually performed a restore and verified it.**

Separately, the complete original Evernote export (every ENEX file) is archived to the same encrypted off-site store — 14,530 files, ~27 GiB — so the source of truth survives independently of Joplin.

> THE RECOVERY STORY, END TO END
>
> Combined with Joplin's offline-first model, total loss of the mini means "restore from R2," not "lose fifteen years of notes." Every client still holds its own full local copy in the meantime, so a hardware failure is a rebuild, never a data-loss event.

This is the part of a self-hosting project that's easy to defer and dangerous to skip. If you build nothing else from this guide, build the backup — and then *test the restore*, because an untested backup is a hope, not a plan.

§ 09 · THE LEDGER

Economics & honest trade-offs

$250/year → ~$0/year recurring. Here's the real cost, and what you give up.

The headline: it runs on a mini that's already on for other work, the embeddings and voice transcription both run locally, and the only metered cost is Claude API calls for answers — cents, not dollars. Recurring cost is effectively zero.

But self-hosting isn't free in every sense, and it's worth being honest about the trade-offs:

WHAT YOU GAIN	WHAT YOU GIVE UP
Your data on hardware you own	Sharing, collaboration, real-time co-editing (this is single-user by design)
AI search that actually works, locally and privately	Handwriting / ink capture, and in-app PDF annotation
Encrypted backups you've actually test-restored	Single-host availability — if the mini is down, sync pauses (offline-first softens this; there's no 99.99% SLA behind it)
~$0/year recurring	You are now your own sysadmin

> THE REAL COST
>
> The recurring bill went to zero, but the project cost **build time**. Adopting Joplin for the hard 90% is the only reason that was tractable — if I'd tried to build the editor and sync engine too, I'd never have finished. The whole strategy is to spend your finite effort only on the slice that doesn't already exist.

§ 10 · DECISION

Should you do this?

An honest framework, because the answer is genuinely "it depends."

This is a good fit if…

- You're a **single user** (or a household), not a team that needs shared workspaces.
- You already run an always-on machine, or are comfortable doing so.
- You value **owning your data and keeping a sensitive corpus private** enough to invest some build time up front.
- You're comfortable being your own sysadmin — or willing to learn — and you'll actually test your backups.

Reconsider if…

- You need **sharing, collaboration, or team features**. This architecture is deliberately single-user.
- You rely on **handwriting/ink** or **in-app PDF annotation** — neither has a clean path here.
- You want zero maintenance and a vendor to call. Self-hosting trades a subscription for responsibility.

If you land on "yes," the order that worked for me: stand up Joplin Server and get all your clients syncing first; migrate *one* notebook and verify fidelity before the full import; **do not cancel your old subscription until you've audited the migration and archived the original export**; then layer on the companion features one at a time. The AI companion is the most rewarding part, but it's worth nothing if the migration underneath it lost your notes.

APPENDIX A

Hard-won gotchas

Each of these cost me a debugging cycle. May they cost you none.

Joplin Data API

- The `limit` parameter maxes out at **100** (200 returns HTTP 400). Page through results.
- A multi-field `fields=id,title` request can return 400 in some cases — fetch a note body on its own.
- Folder-delete does *not* cascade to trashed notes, and **the Trash syncs** — a deleted notebook keeps downloading to your phones until you empty the trash and sync.

Import & conversion

- Joplin's native ENEX importer is unusable for a large, highlight-heavy archive (it aborts on malformed anchor tags and mangles highlight spans). Yarle drops highlight colors and collides titles. A custom converter was the only path to 100% fidelity.
- **Never return raw HTML from a Markdown-converter rule.** Emit a placeholder and restore it after the conversion pass, or you'll silently corrupt unrelated blocks in large documents.
- Evernote's own backup database is often a *fuller* source than the per-notebook ENEX export — prefer it when repairing individual notes.

Multi-client sync

- **Data API writes only hit the local client's database.** They reach other devices only after that client syncs up to the server and the others sync down. Verify the server actually has a change before blaming a device.
- **Run OCR on one machine only.** Leaving OCR on across multiple desktops churns tens of thousands of resources every cycle and starves sync.
- **Don't click "Synchronise" while a sync is running — it cancels it.** On a 53,000-resource library the initial indexing phase takes several minutes; cancel it before the download phase and you get a death-loop where nothing pulls. Let one sync run uninterrupted.
- An open note's editor caches its content. After a sync pulls a newer version, click away and back to refresh the view.

APPENDIX B

The stack at a glance

Every component, and the one reason it's there.

LAYER	CHOICE	WHY
Sync hub	Joplin Server + Postgres (Docker / colima)	Core sync, every client; not a plugin
Remote access	Tailscale Serve + Let's Encrypt	Real cert for iOS; no public attack surface
OCR + automation host	Always-on Joplin desktop on the mini	Desktop-only OCR + the local Data API
Companion service	FastAPI on the mini	Owns the custom 10%, reachable by every device
Embeddings	Ollama · nomic-embed-text	Local, private, $0; corpus has a personal journal
Vector store	sqlite-vec (single file)	Incremental, OCR-aware, trivial to back up
Answers	Claude (zero-data-retention)	Best at synthesizing across retrieved notes; cites sources
Transcription	mlx-whisper · large-v3-turbo	Local Apple-GPU; writes transcripts back into notes
Clients	SwiftUI (iOS + macOS) · Compose (Android)	Native, shared codebase where possible
Backups	restic → Cloudflare R2	Client-side encrypted, retention, verified restore
At-rest	FileVault + SSH key-only	Defense in depth on the always-on host

That's the whole system. None of the individual pieces are exotic; the engineering was in choosing which 10% to build, getting the migration to 100% fidelity, and wiring the AI layer so it works on the device I actually carry.

THE BOTTOM LINE

You can own this.

Replacing a polished commercial product with self-hosted open source plus a focused slice of custom code isn't a downgrade. Done right, you come out with *more* than you started with: AI search that actually works, automatic voice transcription, your data on your own hardware, encrypted backups you've genuinely test-restored — and no annual bill.

The whole trick was refusing to rebuild the 90% that already exists, and spending every bit of the effort on the 10% that doesn't. That — finding the high-leverage 10% and building exactly it — is most of what I do.

Read the original write-up, or get in touch:

`slick-engineering.com/blog/2026-06-19-self-hosted-evernote-replacement-joplin-ai-search`

`slick-engineering.com/contact`