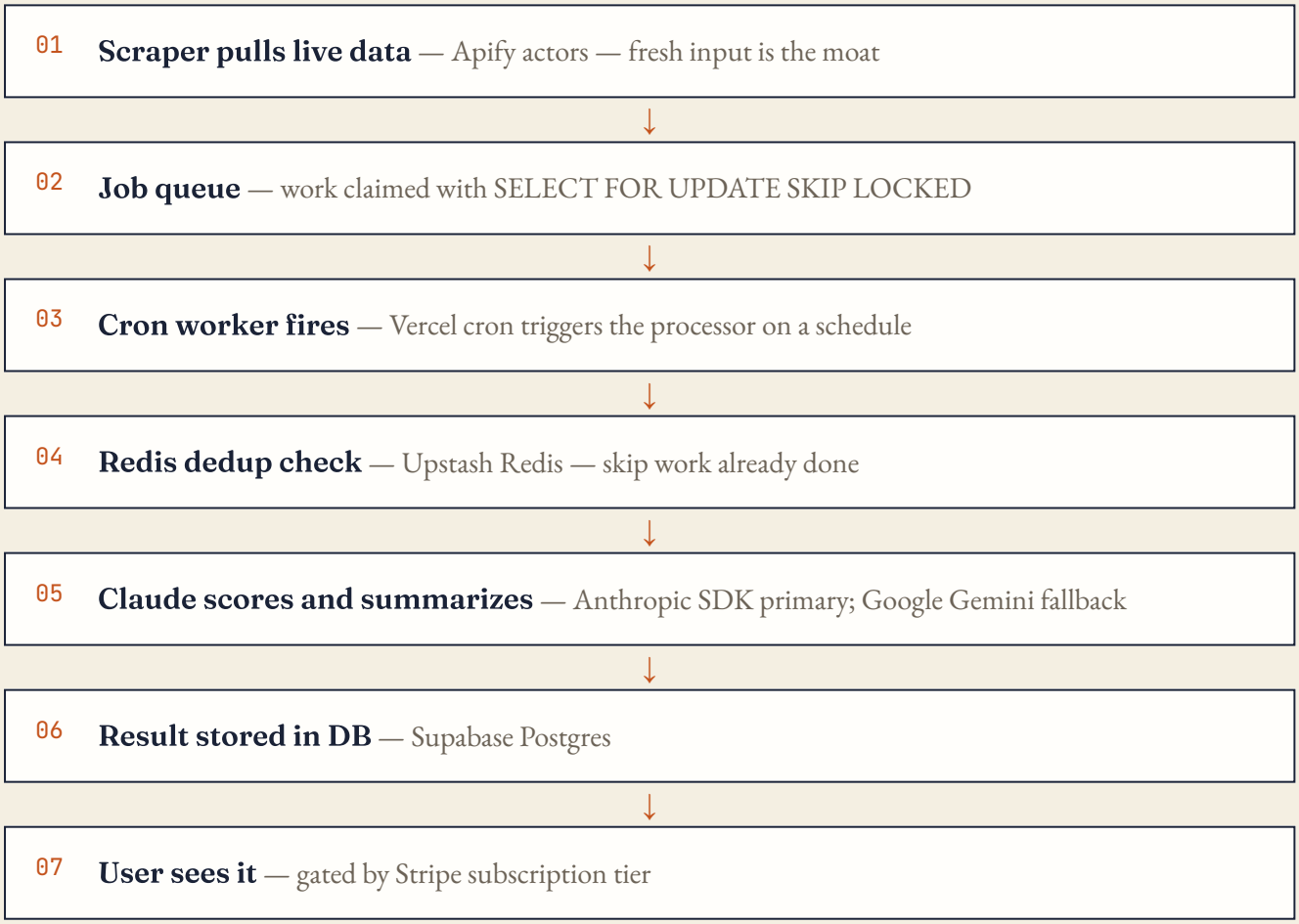
The AI Tool (Micro-SaaS) Playbook

How to ship a small AI tool people pay for monthly: a real data source, an AI that turns it into an answer, and a subscription around it.

A micro-SaaS (a small, focused software product with a subscription) built on AI has one core job: take a real data source, run it through a model, and give the user an answer they'd otherwise have to dig for themselves. A live tool doing real revenue uses exactly this stack. The tricky parts aren't the AI — they're the plumbing that makes the work reliable and the cost manageable.

Here's how the data actually flows once it's running:



The Build Sequence

1. The idea — a narrow, repeated job

The premise of micro-SaaS is simple: find one thing people need done on a regular basis, then do it for them reliably. The job has to be narrow enough that a focused tool can do it well, and repeated enough that monthly billing makes sense.

AI earns its place here when the job involves synthesizing a lot of information into a judgment that would take a human time to reach. Not "generate random content" — something specific, useful, and tied to real data.

2. Framework + hosting — Next.js on Vercel

Next.js handles the UI (the dashboard where users see their results), the API routes (auth, job submission, result retrieval), and the cron triggers. Vercel deploys it all from a git push. One repo, one platform.

The reason Vercel matters beyond convenience: it runs scheduled cron jobs (`vercel.json` cron config) that trigger your worker on a schedule. No separate queue infrastructure, no external cron service. The cron fires a serverless function, which does the work.

3. Auth + data — Supabase (magic-link login + Postgres)

Supabase provides both the database (Postgres) and authentication. Magic-link login: a user enters their email, gets a one-time link, clicks it, gets a session. No passwords to forget or reset.

Everything the tool produces lives in Postgres: user records, niche configurations, queued jobs, processed results, cached outputs. Supabase's JavaScript SDK (`@supabase/ssr`) handles session management in the Next.js middleware layer so authenticated routes are protected before the page renders.

4. The real input — live data via scraper (Apify)

This is the moat. Anyone can prompt Claude with a static dataset. A tool that pulls fresh, live data from the real world gives users something they can't replicate by opening ChatGPT.

The live build uses Apify actors — cloud-hosted scrapers — to pull public data on a regular cadence. `apify-client` is the Node SDK for triggering runs and retrieving results. The scraped data lands in the job queue, tagged to the niche the user configured.

Why Apify: it handles the infrastructure of running scrapers at scale (proxies, retries, storage) so your code just calls an API. You're not managing headless browsers on a server.

5. The AI core — Claude with a fallback

The scraped data goes to Claude (Anthropic SDK) for classification, scoring, and summarization. The model reads the raw input and writes a structured result the user can act on.

The live build has a fallback: if Claude is unavailable or rate-limited, it retries with Google's Gemini (`@google/genai`). This is the pattern for any production AI tool — never depend on a single model provider for uptime.

Why Claude for the primary: quality of reasoning on classification and summarization tasks. Why have a fallback at all: API outages happen; a fallback keeps the product working when they do.

6. Do the work in the background — job queue + cron worker

You don't want AI + scraping blocking a user's page load. Instead:

1. When there's work to do, write a job record to the database.
2. A Vercel cron job fires on a schedule and runs a worker.
3. The worker picks up jobs with a Postgres pattern: `SELECT ... FOR UPDATE SKIP LOCKED`. This is a database-level lock that lets multiple workers run in parallel without two of them grabbing the same job. It's simpler than a dedicated queue service and works fine at micro-SaaS scale.
4. Upstash Redis (serverless key-value store, accessed via `@upstash/redis`) handles deduplication — before processing, check if you've already processed this exact input recently. If yes, skip it.

This "background processing" pattern is why the tool can handle a batch of users without slowing down the UI or burning through AI budget on repeated work.

7. Charge for it — Stripe subscriptions with tiers

Stripe manages subscriptions: free tier, paid tiers, payment methods, invoices. Each tier has a cap on what the user can do (e.g., how many niches, how many results per day). Before the worker processes a job, it checks the user's current Stripe subscription to confirm they're within their tier's limits.

The live build uses Stripe's `stripe` Node SDK for creating checkout sessions and `@supabase/supabase-js` to look up the user's tier from a Supabase table synced via Stripe webhooks.

Why enforce limits at the worker level (not just the UI): a determined user could bypass a UI check. The worker is the right place to enforce entitlements because it runs on your server.

8. Keep it from bankrupting you — cost cap + error monitoring

AI and scraping both cost money at scale. The live build has a daily cost-cap kill switch: a flag in the database that halts the processing pipeline when it's set. A cron job checks aggregate AI spend and flips the flag if daily costs exceed the threshold.

Sentry (`@sentry/nextjs`) catches errors in production and surfaces them with full stack traces. Without it, you find out about failures when users email you or churn.

Why build the kill switch early: AI costs can spike unexpectedly when a batch job misfires or a scraper returns much more data than expected. A kill switch is cheap to build and expensive to not have.

9. Ship carefully — allowlist first, then widen

Don't launch to everyone at once. Open to a small allowlist — 20 to 50 users — and watch what breaks. Fix the failure modes before you widen access. The live build used this pattern before a general launch.

Why: problems that look minor with 5 users look catastrophic at 500. Find the broken paths while they're cheap to fix.

What it costs to run

This is the one playbook where you need to budget for real operating costs. AI and scraping aren't free at any meaningful scale.

To prototype: Near $0. Vercel Hobby free, Supabase free, Upstash Redis free tier, Sentry free tier. Stripe only charges per transaction. Claude costs a few cents per run. A weekend of testing might cost $5–10 in AI calls.

When you have real users: Costs that actually show up:

- **Apify** — the free tier is limited; the Starter plan (~$49/mo) is the realistic entry point for any regular scraping workload.
- **AI spend** — this scales directly with usage. Set a daily budget cap in your Anthropic account and build the kill switch (Step 8) before you launch.
- **Vercel Pro (~$20/mo)** — cron jobs are limited on the Hobby tier; Pro unlocks more frequent cron triggers and more compute.
- **Supabase Pro (~$25/mo)** — once you're storing meaningful job history and result data, you'll hit the free tier's limits.

The first subscriptions you sell should cover Apify and Vercel Pro. Everything else scales with usage.

If you want to work through this stack with people who are shipping the same kind of thing, that's what the Workshop is built for.

KEEP GOING

Pick one step. Do it today. Ship the thing — then build the next one.

MORE PLAYBOOKS — FREE

The Tools I Use	paulguardado.com/playbooks/the-tools-i-use
The Niche Content Site Playbook	paulguardado.com/playbooks/niche-content-site-playbook
The Creator Platform Playbook	paulguardado.com/playbooks/creator-platform-playbook

— Paul