CASE STUDY | AI PRODUCT

# A Job Tracker That Learns

How a static list of job leads became a self-improving, two-agent daily system — the problem, the architecture, the product decisions, and what broke along the way.

| Type | Cadence | Agents | Pattern |
| --- | --- | --- | --- |
| Personal project | Daily, unattended | 2 (worker + critic) | Stateless + external state |

## Executive summary

I built a scheduled AI system that curates about 15 fitting job openings every morning, never re-suggests a role I've already handled, learns my taste from what I apply to versus skip, and has a second, independent agent review its work each day. The engineering was simple. The value was in the architecture: a **stateless worker agent plus an independent critic agent operating over a shared external memory** — chosen deliberately over heavier options like fine-tuning a model or running a fleet of parallel agents. The core lesson: knowing how little to build.

## 1. The problem

Job hunting is a daily search problem with a memory problem underneath it. Every morning means re-running the same queries across a dozen sites, re-reading roles already seen, and mentally filtering for the two things that actually gate the search: **experience level** and, on a STEM-OPT visa, **whether the employer can even hire the candidate** (E-Verify enrollment matters more than H-1B sponsorship for the first few years). A static list solves none of this — it is stale the next day and has no idea what has already been applied to.

The real requirement was never 'a list of jobs.' It was a system that shows up daily, remembers what it already showed, learns what the user actually pursues, and checks its own work — with the user doing nothing but tagging what they apply to.

## 2. What I built

Two scheduled agents that run back-to-back each morning, unattended, over a shared cloud database, publishing to a single live dashboard:

### The worker agent (11:00 AM)

Reads what prior runs learned, searches a broad set of sources, drops anything already handled, ranks the rest toward the user's revealed preferences, publishes the dashboard, and records what it learned.

### The critic agent (11:20 AM)

A separate, independent run that reviews that day's board against the rules and preferences, flags what is off (a role that should have been excluded, a weak fit ranked high, a missing direct link), and records its critique for the next worker run to fix. It advises; it does not edit the board itself.

## 3. The daily choreography

| | |
|---|---|
| **11:00** | **Worker runs (~8 min).** Reads reflections + critique + applied history, searches, excludes handled roles, ranks to taste, publishes ~15 roles, then overwrites its reflection note (this clears the old critique). |
| **11:20** | **Critic runs.** Independently reviews the fresh board, writes its critique back into the shared store. |
| **Next day** | **Worker reads both** the reflection and the critic's critique first thing, fixes what was flagged, and the cycle repeats. |

That read-then-overwrite-then-critic-rewrite order is why the critique always survives to the next morning: the worker reads it before clearing it, and the critic re-adds a fresh one 20 minutes later.

# 4. How it works — five ideas

### Stateless compute

Each run is a brand-new session with no memory of the last. That is a feature: cheap, disposable, and impossible to corrupt with stale in-process state. The model can be swapped or upgraded at any time without losing history, because no history lives inside the agent.

### External state

Continuity is delegated to a cloud database beside the agents — the roles applied to or skipped, the worker's reflections, and the critic's reviews. Stateless workers behave as if they remember because the memory is in the data, not the model. It is also inspectable and fixable: the database was opened and de-duplicated by hand once — something impossible to do to a model's internal memory.

### Preference learning

Every run contrasts what the user applies to against what they skip, and biases the day's ranking accordingly — a recommender-style loop that needs no model training, just the user's own signals read back at runtime.

### Reflection loop

At the end of each run the worker records what it learned and what to adjust; the next run reads that first. The agent does not self-modify — the system around it improves because its inputs get richer each day.

### Evaluator / critic pattern

Self-review has a blind spot: an agent grading its own work tends to rationalize its choices. So a separate critic agent — fresh context, not invested in the output — independently audits each board and feeds its critique back in. Worker proposes, critic disposes. This is the one place a second agent earned its keep; note it runs in sequence after the worker, not in parallel.

# 5. The data model

All shared state lives in the dashboard artifact's cloud database (not in the page's code, not in the browser). Two collections:

| | |
|---|---|
| **applications** | One document per role the user has tagged. Fields: company, role, location, salary, url, status (Applied / Skip / Interested), updatedAt. This is what powers exclusion and preference learning. |
| **reflections** | A 'latest' document plus dated history. Fields: prefs (the worker's inferred read of the user's taste), notes (adjustments for next run), critique (the independent critic's findings), plus timestamps. The dashboard surfaces all three in a 'What it has learned' panel. |

A subtle but important lesson lived here: keying records on company + role *text* caused duplicates when a name drifted between runs (‘Tint’ versus ‘Tint AI’). Identity and schema design matter for agents exactly as they do in any data system.

## 6. Key product decisions

The build was straightforward. The product work was choosing the lightest tool at each fork.

### Fine-tune a model on my preferences?  No.

Retrieval plus a feedback loop delivers the same behavioral outcome at a fraction of the cost — no labeled dataset, no training pipeline, and it never goes stale. Fine-tuning earns its place for format-at-scale or latency, none of which applied here. Knowing when not to train was the point.

### How many agents, and in what shape?  Two, in sequence.

A ‘lead plus parallel searchers’ design would widen coverage but adds cost and coordination-failure risk for little gain at this scale. The multi-agent pattern that did earn its place was an evaluator/critic: worker then critic, sequentially. Different problem, different pattern.

### Store state in the page, or a real database?  Database.

Browser storage is per-device and invisible to the agents. Only a shared, server-side store lets the user’s clicks and both agents read the same source of truth — which is the entire reason the system stops re-suggesting handled jobs and can carry a critique from one agent to another.

## 7. What broke, and what I learned

- **Silent write drops.** Rapid tagging tripped a rate limit and some saves failed quietly. Fix: a queue that spaces and retries writes, plus a visible ‘synced’ state. Observability matters more for agents than for apps, because they fail where no one is watching.
- **Identity drift.** Keying records on company + role text created duplicates when names shifted between runs. Lesson: agents produce and consume data; a stable identity key matters as much as in any system.
- **A silently open loop.** After adding the critic, a verification pass caught that the worker’s instructions never told it to read the critique — so the critic was reviewing into a channel no one consumed. It looked done and was not. Always trace the full data path end to end; ‘it ran’ is not ‘it worked.’
- **Freshness versus volume.** A strict 7-day window plus a growing exclusion list can leave fewer than 15 fresh roles on a slow day. The honest fix is to fill with the freshest available and say so, not to pad with stale or off-target listings.

## 8. What’s next

- **Outcome tracking:** record what happens after applying (call / reject / silence) and weight the sources and role-types that actually convert.
- **Parallel searchers:** promote to a lead-plus-searchers design if source coverage becomes the bottleneck.
- **Self-editing reflections:** let the loop propose its own instruction tweaks for human review — a step toward genuine self-improvement, with a person in the loop.

**The one-line version:** the intelligence is a fixed, shared model; everything that makes this feel smart, personal, and self-correcting lives in the design around it — stateless runs, external memory, a preference loop, and an independent critic. The scarce skill was not building the agent. It was choosing how little to build.

## Appendix: glossary

| | |
|---|---|
| **Agent** | A model plus tools plus a goal, running in a loop until the goal is met. |
| **Stateless** | Retains nothing between runs; each run’s behavior depends only on that run’s inputs. |
| **External state** | Memory kept in a store beside the agent (here, a cloud database) rather than inside it. |
| **Reflection loop** | The agent writes notes at the end of a run that the next run reads — behavioral learning without retraining. |
| **Evaluator / critic** | A second, independent agent that reviews the first agent’s output and feeds back corrections. |

| | |
|---|---|
| **Fine-tuning** | Changing a model's weights by training on new data — deliberately avoided here in favor of retrieval + feedback. |