How I Write Incident Notes That Hindsight Can Actually Recall

Most of the work in building an agent with memory isn't the agent. It's deciding what a memory looks like when you write it.

I spent more time on the format of a single incident record than on any endpoint in RecallOps, and I'd make that trade again. Here's what I settled on, and why retrieval quality is mostly decided at write time.

Retrieval is a write-time problem

When recall goes badly, the instinct is to tune the retrieval side: adjust the query, change how many results come back, rewrite the prompt. Sometimes that helps. But in my experience the bigger lever sits earlier. If the record that went in is vague, nothing downstream can rescue it.

An incident agent is a good stress test for this, because the source material is famously bad. Real postmortems are long, chronological, full of chat excerpts, and light on the one thing you want later: *what to do when this happens again.*

What RecallOps does

You describe a live incident, and RecallOps recalls similar past incidents from Hindsight, an open-source agent memory engine, then returns a specific answer. That answer covers the likely cause, recommended steps, fixes that already failed (with incident IDs), and who resolved something similar before.

The stack is FastAPI, a plain HTML/JS page, and Groq for the LLM. Everything is stored in one Hindsight memory bank, `recallops-incidents`. The interesting design work is in `memory.py`, which wraps `retain`, `recall`, and `reflect`, and in the shape of the records I feed it.

The record format

Here's what one retained incident looks like:

```
Incident INC-245 (2026-03-12) on payment-api. Severity HIGH.
Symptoms: HTTP 500 errors, p99 latency 8s, "FATAL: remaining connection slots are reserved".
Root cause: deploy #458 reduced DB max pool size from 100 to 20.
What worked: rolled back deploy #458; errors cleared in 4 minutes.
What failed: restarting payment-api (no effect); raising request timeout to 30s (made it worse).
Resolved by: Priya Sharma (DB team). Time to resolve: 72 minutes.
```

Every choice in there is deliberate.

One incident, one memory. I don't chunk the record. A memory of "root cause: deploy reduced the pool" is useless without the symptoms that let you match it, and a memory of the symptoms is

useless without the fix. They travel together.

Labeled prose, not JSON. The reader of this text is a language model, and the recall side is semantic. Natural language with consistent labels (`Symptoms:`, `Root cause:`, `What worked:`, `What failed:`) reads well for both humans and models. I didn't need a schema for the memory itself.

Symptoms use the words people actually type. The log line, `remaining connection slots are reserved`, is quoted verbatim. When someone describes a new incident at 2 AM, they paste the error. Matching that is far easier when the stored record contains the same string.

Failures get their own line. This is the field I'd defend hardest. Almost nobody writes down what didn't work, and it's the highest-value part of the record.

A named human and a time to resolve. "Resolved by: Priya Sharma (DB team)" is what powers the "who fixed this before" answer, and the duration gives a sense of how bad the incident was.

Testing recall against distractors

A perfect format doesn't prove anything if the bank only contains records that match. So my seed set, a fictional online store called ShopSwift, is built to make recall work for its living.

There are about fifteen incidents across six services. Three are connection-pool problems, each fixed by a rollback with a failed restart. The rest are unrelated on purpose: certificate expiry, Redis memory, search index trouble, Kafka lag, DNS.

If you seed only the interesting cases, every retrieval looks like a success. The distractors are what turn "it returned something" into "it returned the right thing."

The seed data lives in `data/seed_incidents.json` and is loaded through `POST /api/seed`, which is idempotent, so running it twice doesn't duplicate memories. That mattered more than I expected, because I hit that button constantly while iterating on the format.

> **[Insert image here]** Loading sample history and adding a new incident
> Screenshot idea: the memory count after seeding, and the add-new-case form.

Writing is also a product feature

The format isn't only for seed data. When an engineer finishes an incident, they mark it resolved and fill out a form with the cause, what worked, what failed, and who fixed it. That form posts to `/api/resolve`, which builds a record in exactly the same shape and retains it immediately.

This is a quiet design win. Because the form mirrors the labels, the *tool itself* pushes people toward writing good memories. I'm not asking anyone to remember a template. The "What failed" box is just there, and it's hard to leave blank when it's staring at you.

The next similar question benefits right away. There's no reindexing step and no nightly job.

What the result looks like

With that structure in place, describing a new incident with the same symptoms produces a response like this:

```
{
  "likely_cause": "...",
  "steps": ["...", "..."],
  "dont_try": [{ "action": "...", "evidence": "INC-102, INC-245" }],
  "who_fixed": "Priya Sharma (DB team), 2 of 3 similar incidents",
  "similar_incidents": ["INC-245", "INC-188"]
}
```

Notice how much of that is only possible because of the record format. `dont_try` exists because there's a `What failed:` line. `who_fixed` exists because there's a `Resolved by:` line. The response schema is a mirror of the write schema.

Limits

The format is only as good as the person filling it in. A record that says "restarted things, eventually fixed" is retained faithfully and recalled uselessly. The form can nudge, but it can't force good writing.

I also don't version records. If a root cause is later found to be wrong, there's no built-in way to mark the old memory as superseded. And there's no import from existing ticketing systems yet, so history has to be entered through the form or the seed file.

Lessons learned

1. **Design the memory before the retrieval.** Most recall problems are write problems.
2. **Keep related facts in one record.** Symptoms without the fix, or the fix without the symptoms, are both dead weight.
3. **Quote the strings people will search with.** Verbatim error messages beat paraphrases.
4. **Give failures their own field.** They're the most valuable and least recorded information you have.
5. **Test with distractors.** A memory bank of only relevant items proves nothing.
6. **Make the product enforce the format.** A form with the right labels beats a wiki page with a template.

If you're new to this, an introduction to agent memory explains the concepts, and the Hindsight docs cover the retain and recall calls in detail.