> **TL;DR** Agent transcripts are useful evidence, but they are not a knowledge base by themselves. The practical pattern is a promotion ladder: raw logs become session cards, session cards become a searchable index, repeated lessons become learning candidates, and only reviewed rules become team knowledge.

The problem: agent work disappears

If you use Claude Code, Codex, Cursor, or any serious coding agent every day, you have probably seen this failure mode:

- an agent spends two hours debugging a tool quirk
- the fix works
- the reasoning stays inside the chat transcript
- another session repeats the same mistake a week later

The team did not forget because nobody cared. It forgot because the knowledge never left the transcript.

That is the real problem. Modern coding agents leave behind a lot of raw evidence: messages, tool calls, commands, file edits, errors, summaries, retries, and decisions. Claude Code stores local session transcripts as JSONL files under project-specific directories. Codex has its own session and thread state. Other agent tools have similar traces.

But a pile of logs is not memory.

Memory is what happens after the team can answer:

- what did we learn
- where is the evidence
- when should this change future behaviour
- which rule should the next agent actually follow

Why raw transcript search is not enough

The obvious first attempt is to grep old transcripts. That works for emergencies, but it does not scale as a daily workflow.

Raw transcripts are noisy:

- tool output dominates the useful conversation
- failed commands and retries create duplicate surface area
- temporary theories appear next to final conclusions
- long logs make small decisions hard to find
- local paths, tokens, and environment details can create privacy risk

They are also hard to trust.

If a transcript says "this is fixed", that is not the same as a durable rule. Maybe the agent was wrong. Maybe the user corrected it later. Maybe the fix was only true for one repo, one date, or one provider version.

The goal is not to preserve every word. The goal is to preserve the minimum useful evidence that helps the next session avoid repeating work.

The five-layer knowledge ladder

A practical agent knowledge base has five layers.

1. Raw logs

Raw logs are the evidence layer.

They answer:

- what was said
- what tools ran
- what files changed
- what failed
- what output was observed

Keep raw logs local and treat them as sensitive. Do not casually paste them into new model contexts. Do not upload them wholesale into a third-party vector database unless your privacy model explicitly allows that.

Raw logs are for audit and reconstruction, not for everyday retrieval.

2. Session cards

A session card is one compact summary of a work session.

It should be readable in under a minute.

Good fields:

```
{
  "date": "2026-05-10",
  "project": "example-nextjs",
  "task": "Debug missing SEO analytics data",
  "outcome": "GSC domain property worked; URL-prefix property returned 403",
  "files_touched": ["docs/runbooks/seo/playbook.md", "AGENTS.md"],
  "tools_used": ["GSC MCP", "GA4 MCP", "git"],
  "decisions": ["Use sc-domain property for future GSC calls"],
  "failure_modes": [
    "URL-prefix property returned 403 despite domain property access"
  ],
  "evidence": [
    "GSC query against sc-domain:example.com succeeded",
    "GSC query against https://example.com failed with 403"
  ]
}
```

This is the first useful compression step. The transcript remains available, but the next agent should start from the card.

3. Search index

Once you have session cards, search the cards first.

That can be simple:

- keyword search over JSONL
- SQLite FTS
- DuckDB over JSONL
- local embeddings over session summaries

Do not start by embedding full transcripts. Embed the distilled session cards first.

Most future questions are not:

> Show me every token from the old conversation.

They are:

> Have we seen this failure before?

or:

> Which session decided this property ID?

Session cards answer those questions faster and with less privacy exposure.

4. Learning candidates

Some session cards contain a reusable lesson. Most do not.

A learning candidate is a proposed rule that might deserve reuse:

- a tool failure pattern
- a workflow anti-pattern
- a security rule
- a provider-specific gotcha
- a repeated review finding
- a deployment or analytics trap

For example:

> If a GSC connector returns 403 for a URL-prefix property, test the domain property before concluding the connector is broken.

That is not just a session summary. It is a candidate operating rule.

Candidates should carry:

- summary
- scope
- confidence
- evidence links
- date
- owner or source
- review status

They should not automatically become policy.

5. Promoted knowledge

Promoted knowledge is what the next agent should actually follow.

This belongs in:

- runbooks
- repo docs

- framework instructions
- checklists
- test fixtures
- lint rules
- typed configuration

Promotion is a judgment step.

A lesson should be promoted only when it is:

- repeatable
- specific
- verified
- likely to save future work
- safe to generalize

That is the difference between "the agent said something" and "the team should now operate differently".

A concrete example: the analytics property trap

Here is the kind of lesson that should survive a transcript.

An agent tries to pull Search Console data for a site. It uses:

`https://example.com`

The API returns `403`.

The easy but wrong conclusion:

> The connector does not have access.

The better investigation:

- test the domain property
- test the URL-prefix property
- compare which one the connector can read
- document the working property string
- update the SEO runbook

The durable lesson:

> For this site, use `sc-domain:example.com` for GSC calls. The URL-prefix property returns `403` because the connector account has domain-property access, not URL-prefix access.

That is a good promoted rule because it is:

- concrete
- easy to verify
- likely to recur
- short enough to place in a runbook
- useful to both humans and agents

The raw transcript is no longer the primary interface to the lesson. The runbook is.

What a session card should include

The card should be boring on purpose.

Useful fields:

Field	Why it matters
Task	Helps future search match intent, not only filenames.
Repo or project	Prevents a local rule from leaking into every codebase.
Date	Makes stale provider behaviour easier to detect.
Tools used	Surfaces broken MCPs, flaky browsers, and repeat failure zones.
Files touched	Connects reasoning to actual artifacts.
Outcome	States what changed or what was learned.
Decisions	Captures judgment, not just activity.
Failure modes	Prevents repeated dead ends.
Evidence	Keeps the card auditable.

Keep the card short. If it needs three pages, it is probably a report, not a card.

What not to index

Do not treat agent memory as an excuse to retain everything.

Avoid indexing:

- .env values
- token stores
- API keys
- full credential configs
- private customer data
- large raw command outputs
- full stack traces unless they are the artifact being studied
- encrypted or hidden reasoning payloads
- temporary chain-of-thought-like scratch material

The safest default is:

- index summaries
- store evidence pointers
- keep raw logs local
- retrieve transcript snippets only when needed

That keeps the knowledge base useful without turning it into a liability.

How to extract useful knowledge

Use three passes.

Pass 1: summarize the session

Create one short card per session.

Extract:

- task
- outcome
- files touched
- tools used
- important errors
- final status

This pass should be conservative. It is allowed to say "no useful lesson".

Pass 2: classify operational signals

Tool calls often carry the best reusable evidence.

Classify:

- failed tool calls
- retry loops
- edit failures
- missing permissions
- provider errors
- broken configuration
- commands that fixed the problem

This turns "the agent struggled" into a concrete pattern:

> The Search Console URL-prefix property fails, but the domain property works.

or:

> The screenshot tool was not the issue. The page had not finished hydrating.

Pass 3: extract candidate lessons

Only extract a learning candidate when there is evidence.

Good candidates sound like this:

- "When X happens, test Y before concluding Z."
- "Do not do X unless condition Y is true."
- "For this repo, use X as the canonical source of truth."
- "This provider returns error X when configuration Y is missing."

Weak candidates sound like:

- "Be careful."
- "Improve docs."
- "Use better testing."
- "The agent should remember this."

Specific beats broad.

Promotion rules

Most extracted lessons should remain candidates.

Promote a lesson only when it passes four tests.

1. Is it repeatable?

One odd failure is not automatically a rule. Repeated failures, reproducible behaviour, or provider documentation make a stronger case.

2. Is it scoped?

A rule must say where it applies.

Good:

> For this repo's GSC connector, use the domain property.

Bad:

> GSC uses domain properties.

3. Is it verified?

Evidence can be:

- a passing test
- a successful API call
- a failing API call with exact error
- a commit
- a browser check
- a report
- an official doc

Do not promote naked opinion.

4. Is it worth interrupting future agents?

Every promoted rule adds cognitive load.

If the lesson is rare, keep it searchable. If the lesson is common and expensive to rediscover, put it in the runbook.

A minimal implementation

You can build a useful version without a graph database.

Start here:

1. Keep raw agent logs local.
2. Create one session card per completed session.
3. Store cards as JSONL.
4. Search cards before reading transcripts.
5. Keep durable rules in Markdown runbooks.
6. Add evidence links back to commits, reports, or transcript spans.

That is enough for a small team.

A slightly stronger version adds:

- local embeddings over session cards
- a tool-call digest
- a simple status field for learning candidates
- a weekly review where candidates are promoted, rejected, or left alone

Only add a knowledge graph when relationships become the bottleneck:

- which sessions produced this rule
- which rule superseded another rule
- which repo owns this decision
- which source contradicted it

Do not start with the graph. Start with cards.

Common traps

Summarizing the summarizer

If your summarizer is itself an agent, it may create new sessions. If your pipeline ingests those sessions, it can start summarizing its own summaries.

That creates noisy, recursive junk.

Fix it with a simple guard:

- tag automated summarizer sessions
- exclude them from normal learning extraction
- keep generated summaries separate from human work sessions

Indexing secrets

Agent logs often contain command output. Command output sometimes contains secrets.

Redact before indexing. Do not rely on "we will be careful later".

Treating every agent conclusion as truth

Agents write confident sentences. Confidence is not evidence.

Promote only lessons with proof.

Creating duplicate docs

If a lesson belongs in a runbook, put it there. If it belongs in repo instructions, add a pointer to the runbook.

Do not copy the same rule into five places unless those places have different audiences and you can keep them synchronized.

Missing provenance

Every promoted lesson should answer:

- where did this come from
- who or what verified it
- when was it last checked
- what evidence would overturn it

Without provenance, a knowledge base becomes folklore.

Letting stale docs outrank fresh evidence

Docs age. Provider behaviour changes. Tooling changes.

When a runbook and a fresh reproduction disagree, investigate before repeating the old rule.

Where this fits in an AI production stack

This is not just engineering housekeeping.

AI production work depends on repeated judgment:

- which model failed and why
- which provider setting matters
- which QA gate caught the defect
- which analytics property is canonical
- which deployment check actually proves the fix

If those lessons live only in transcripts, the team keeps paying for the same debugging.

If they move through a promotion ladder, the workflow improves over time.

That is the difference between an agent that helps once and an agent system that gets operationally sharper.

For adjacent production patterns, see:

- How to Run an AI Video Model Bakeoff Without Turning It Into Vibes
- Hardening Agents in Production
- OpenAI Codex CLI /goal Command Guide

Sources checked

- Claude Code docs: How Claude Code works
- Claude Code docs: Hooks reference
- Claude Code docs: Session storage
- OpenAI Help Center: OpenAI Codex CLI getting started