

What a Good Agent Trace Looks Like: A Reference Schema

A practical spec for logging AI agents that answers "what did it see", "why did it do that", "what did it cost" and "who stepped in". Built on OpenTelemetry. Ships with the schema itself.



About this document

This whitepaper specifies a reference schema for tracing AI agents. It is written to be implemented: every field named here is intended to be emitted by a production agent runtime through an OpenTelemetry SDK, and every design rule is stated with the operational question it is meant to answer.

The document is organised in thirteen sections. Sections 1 to 3 establish the problem and the design rules. Sections 4 and 5 define the trace structure and the full field specification. Sections 6 to 10 cover the four additions the schema makes to conventional tracing: skipped options, human overrides, per-span cost, and per-span redaction and retention. Sections 11 to 13 provide a worked example, the mapping to the OpenTelemetry GenAI conventions, and a staged adoption path.

Document control

FIELD	VALUE
Title	What a Good Agent Trace Looks Like: A Reference Schema
Document type	Technical whitepaper — reference schema
Published by	Blue Data Consulting
Status	Released for implementation
Intended audience	Platform architects, platform engineering leads, agent developers, and observability and compliance stakeholders
Normative basis	OpenTelemetry GenAI semantic conventions; extension fields under the <code>agent.*</code> namespace

Notice

The OpenTelemetry GenAI semantic conventions referenced throughout this document are under active development. Where this schema adds a field, it does so under a dedicated `agent.*` namespace so that a concept graduating into the standard can be adopted by renaming a single field in a single span processor.

Retention periods given in Section 10 are stated as engineering defaults and as a floor, not as legal advice. Sector-specific and jurisdictional obligations may require longer windows, and should be confirmed with the relevant compliance function before implementation.

All identifiers, costs, timings and log values in the worked example in Section 11 are illustrative dummy data.



Contents

1. Executive summary.....	1
2. Why normal tracing isn't enough for agents	2
3. Six rules the schema follows.....	3
4. The shape of a trace	4
The nine span kinds.....	4
5. What goes on each span	6
Common to every span	6
Root span: invoke_agent.....	6
Model call: chat	7
Tool call: execute_tool	8
Retrieval and guardrail spans.....	9
6. Skipped options (rejected branches)	10
7. Human overrides.....	12
8. Cost per step.....	14
9. Keeping personal data out.....	15
10. How long to keep what	16
11. A full example: spans and logs	17
The same run, as a log stream.....	18
12. How it maps to OpenTelemetry.....	19
13. How to adopt it, step by step	20
The reference architecture.....	20
The three levels	20
Checklist	21



Figures and tables

FIGURES

Fig 1. Session, trace, span, event.....	4
Fig 2. Ran and skipped options as siblings under the plan span	10
Fig 3. Self cost on leaves, roll-ups on parents	14
Fig 4. Scrub before export, then route by retention class.....	15
Fig 5. The reference architecture	20

TABLES

Table 1. Agent properties and their effect on the trace (Section 2)
Table 2. The six design rules (Section 3)
Table 3. The nine span kinds (Section 4)
Table 4. Fields common to every span (Section 5)
Table 5. Root span field list (Section 5)
Table 6. Model call field list (Section 5)
Table 7. Tool call field list (Section 5)
Table 8. Retrieval and guardrail field lists (Section 5)
Table 9. Skipped-option reason codes (Section 6)
Table 10. Human override fields (Section 7)
Table 11. Human override reason codes (Section 7)
Table 12. Costing rules (Section 8)
Table 13. Content modes (Section 9)
Table 14. Retention classes (Section 10)
Table 15. Mapping to the OpenTelemetry GenAI conventions (Section 12)



SECTION 1

Executive summary

A trace is a familiar idea: a tree of spans showing what a request did and how long each part took. This schema keeps that, and adds four things that agents need and normal services don't.

THE FOUR ADDITIONS

- **Skipped options are spans too.** When the agent considers three tools and calls one, the other two show up in the trace with a reason they were skipped. So "why didn't it do X" is a filter, not an investigation.
- **Human overrides are spans with a required reason.** Who stepped in, what they changed, and a reason code from a short list. This distinguishes "the agent is drifting" from "the policy changed".
- **Cost lives on every span.** Not just the run total, which makes it possible to find the one retrieval step that made everything expensive.
- **Redaction and retention are set per span type.** Prompt content is kept for 30 days and scrubbed of personal data. Human decisions are kept for six months or more. Timings and counts are kept for a year.

Everything else follows the OpenTelemetry GenAI conventions, which already define spans for agent runs, model calls and tool calls. Where a standard name exists, we use it. Where it doesn't, we add an `agent.*` field. Teams already emitting OTel spans from their framework are at Level 1 of three (Section 13).

INTENDED AUDIENCE

This document is written for the architect deciding what an agent platform must log before the first workload lands; the platform lead who has inherited a running agent and a dashboard that answers the wrong questions; and the developer who gets paged when the agent does something odd and needs the trace to explain why, rather than leaving it to guesswork.



SECTION 2

Why normal tracing isn't enough for agents

Tracing was built for services whose behaviour is fixed by their code. Knowing the input and the version is enough to know what the service will do. So a trace only records what varies between requests: timings, IDs, status codes. Nobody logs the code, because it can be read in the repo.

An agent flips this. The code is small and stable. The behaviour is decided at run time by a model reading its context. That context, the prompt, the retrieved documents, the tool results, the earlier turns, is effectively the program. Five things follow, and each one is a place where a normal trace goes silent.

AGENT PROPERTY	WHAT IT DOES TO THE TRACE
The context is the program	If the prompt version, retrieved docs and tool schemas aren't on the span, this is the output of code that can't be read.
Same input, different path	A run can't be reproduced from the request alone; everything the model saw is needed.
It considers options and drops most	The path not taken is usually the one under review, and a normal trace has nowhere to put it.
Humans are part of the run	Approvals, edits and rejections happen inside the flow. Their reasons are the most valuable signal available.
Every step has a price and a privacy risk	Cost and redaction have to work per span, not per run.

IMPLEMENTATION NOTE

A normal service trace answers: where did the time go, which hop failed? An agent trace has to answer three more: what did the model see, why did it choose that, and what would it have cost to choose differently? If a current trace can't answer those, that's the gap this schema fills.



SECTION 3

Six rules the schema follows

RULE	WHAT IT MEANS IN PRACTICE
Standard names first, own namespace second	Use OTel GenAI span kinds and <code>gen_ai.*</code> attributes as-is. Everything extra goes under <code>agent.*</code> . When the standard catches up, renaming one field in one processor is all it takes.
Every span carries its own inputs, outputs, cost, and retention class	Nothing important lives only on the root. Totals are computed from children, not stored as the only record.
Absence is data	An option that was considered and skipped is a span with a status, not a missing row. A guardrail that blocked something is a span, not a log line.
Human decisions need a reason code	An override without a reason category is invalid. Free text is welcome; the code is what makes it queryable.
Redact before export, hash for grouping, keep structure	Content leaves the process only after a processor scrubs it. IDs become stable pseudonyms. Timings, sizes and order are never touched.
Retention follows the span, not the trace	One retention window per trace is either too short for audit or too long for privacy.



SECTION 4

The shape of a trace

IN PLAIN TERMS

A **session** is a conversation. A **trace** is one run of the agent inside that conversation. A **span** is one step in the run. An **event** is a blob of content (like the prompt text) attached to a span, kept separate so it can be scrubbed or dropped without losing the span itself.

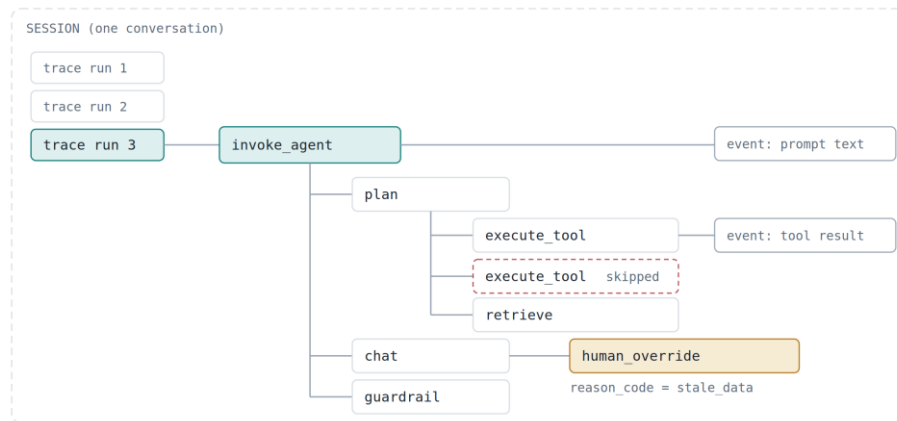


Fig 1. Session, trace, span, event. The skipped tool (dashed) and the human override (amber) are ordinary spans in the tree. Content hangs off spans as events so it can be scrubbed independently.

The nine span kinds

Five come straight from the OTel GenAI conventions. Four are additions.

SPAN KIND	FROM	WHAT IT REPRESENTS
<code>invoke_agent</code>	OTel	The root of one run. Which agent, which prompt version, the totals.
<code>invoke_workflow</code>	OTel	Orchestration across several agents (a graph or router). Only when it's the entry point.
<code>chat</code>	OTel	One model call. Model, settings, tokens, why it stopped, cost.
<code>execute_tool</code>	OTel	One tool call, whether it ran or was skipped. Args, result, side effects.
<code>retrieve</code>	OTel	A lookup from an index or memory. Query, sources, freshness.
<code>agent.plan</code>	Ours	A decision point. Options considered, which were chosen, why. Parent of the options.
<code>agent.guardrail</code>	Ours	A policy or safety check and its verdict. Blocks become skipped spans.
<code>agent.handoff</code>	Ours	Delegation to a sub-agent or a queue.
<code>agent.human_override</code>	Ours	A person's decision on something the agent proposed.



SECTION 5

What goes on each span

IN PLAIN TERMS

Every span gets six common fields (session, run, retention class, its own cost, content mode, and for parents a rolled-up cost). Then each span kind adds its own. The tables below are the full spec; the short version is in the "must have" box under each one.

Levels: **required** always; **conditional** required when the condition applies; **optional**. Names starting `gen_ai.` are OTEL unchanged; names starting `agent.` are ours.

Common to every span

FIELD	LEVEL	NOTES
<code>agent.session.id</code>	required	The conversation's key. Pseudonymous if it points at a person.
<code>agent.run.id</code>	required	Same as trace ID; repeated so it survives export to log stores.
<code>agent.retention.class</code>	required	<code>metadata</code> , <code>content</code> or <code>decision</code> . See Section 10.
<code>agent.cost.usd</code>	required	This span's own cost, not counting children. Zero is fine.
<code>agent.cost.rollup_usd</code>	conditional	Self plus all descendants. Required on root, plan and handoff spans.
<code>agent.content.mode</code>	required	<code>none</code> , <code>sanitized</code> or <code>full</code> . Tells the reader what content to expect. See Section 9.

Root span: `invoke_agent`

MUST HAVE

- Which agent, which prompt version (label + hash), which tool schemas (hash), which framework version.
- How the run ended, how many overrides, how many skipped options.

Full field list

FIELD	LEVEL	NOTES
<code>gen_ai.agent.name</code> , <code>gen_ai.agent.id</code>	required	Human name and stable ID of the agent definition.
<code>agent.prompt.version</code> , <code>agent.prompt.hash</code>	required	The hash catches edits nobody bumped the version for.



FIELD	LEVEL	NOTES
<code>agent.tools.schema_hash</code>	required	Hash of the tool definitions shown to the model. Renaming a parameter changes behaviour.
<code>agent.runtime.framework</code> , <code>agent.runtime.version</code>	required	Framework upgrades change behaviour too.
<code>agent.outcome</code>	required	<code>completed</code> , <code>overridden</code> , <code>escalated</code> , <code>abandoned</code> , <code>error</code> .
<code>agent.override.count</code> , <code>agent.branches.rejected</code>	required	Counts, so runs can be filtered without walking the tree.

IMPLEMENTATION NOTE

Someone edits the system prompt without bumping the version. Two days later behaviour shifts. Filtering traces by `agent.prompt.hash` surfaces the exact change point in seconds. Without the hash, it's git history diffed against a timestamp, and hoping.

Model call: chat

MUST HAVE

- Provider, model requested, model that actually answered.
- Tokens in and out (from the provider, not estimated), why it stopped, sampling settings.
- A list of references to everything in its context (which tool results, which retrieved chunks, which prior messages).

Full field list

FIELD	LEVEL	NOTES
<code>gen_ai.provider.name</code> , <code>gen_ai.request.model</code> , <code>gen_ai.response.model</code>	required	Response model can differ from request (aliases, routing). Log both.
<code>gen_ai.usage.input_tokens</code> , <code>gen_ai.usage.output_tokens</code>	required	From the response, not counted locally.
<code>agent.usage.cached_input_tokens</code>	conditional	When the provider reports prompt caching. Priced differently.
<code>gen_ai.response.finish_reasons</code>	required	<code>stop</code> , <code>tool_calls</code> , <code>length</code> , <code>content_filter</code> . Length means it got cut off.
<code>gen_ai.request.temperature</code> and other sampling params	required	Anything that affects the output.
<code>agent.context.sources</code>	required	Array of span IDs or content hashes for every input in this call's context.
<code>agent.pricing.version</code>	required	Which rate card priced this span.



FIELD	LEVEL	NOTES
events <code>gen_ai.input.messages</code> , <code>gen_ai.output.messages</code>	conditional	Present in <code>sanitized</code> or <code>full</code> mode. Always as events, never attributes.

Tool call: `execute_tool`

MUST HAVE

- Tool name and the call ID that links it to the model's request.
- Status: ran, skipped, or failed.
- Does it change the outside world? If yes, an idempotency key.
- How old the returned data is (if cached), and how many tokens the result added to the next model call.

Full field list

FIELD	LEVEL	NOTES
<code>gen_ai.tool.name</code> , <code>gen_ai.tool.call_id</code>	required	Call ID links to the preceding chat span.
<code>agent.tool.status</code>	required	<code>executed</code> , <code>not_executed</code> , <code>failed</code> .
<code>agent.tool.side_effect</code>	required	True for send, write, pay, delete.
<code>agent.tool.idempotency_key</code>	conditional	Required when <code>side_effect</code> is true. Tells "decided" from "did" from "did twice".
<code>agent.tool.args_hash</code> , <code>agent.tool.result_bytes</code>	required	Structure that survives redaction.
<code>agent.tool.result_age_s</code>	conditional	When the tool returns cached data. Stale reads are a top cause of "correct reasoning, wrong facts".
<code>agent.tool.tokens_added</code>	required	Tokens this result pushed into the next model call. The most useful cost field in practice.
<code>agent.tool.external_cost_usd</code>	conditional	For tools that call metered APIs.
events <code>gen_ai.tool.call.arguments</code> , <code>gen_ai.tool.call.result</code>	conditional	In <code>sanitized</code> or <code>full</code> mode.

**IMPLEMENTATION NOTE**

The reasoning log says "decided to email the customer". The mail provider shows two emails. Did the model call the tool twice, or did the tool retry? With an idempotency key on the span, the answer is one lookup. Without it, it's two systems' logs read side by side, guessing at the clock skew.

Retrieval and guardrail spans

retrieve

FIELD	LEVEL	NOTES
<code>agent.retrieve.index,</code> <code>agent.retrieve.top_k</code>	required	Which store, how many asked for.
<code>agent.retrieve.sources</code>	required	Doc IDs, versions, chunk sizes returned.
<code>agent.retrieve.freshness_s</code>	required	Age of the oldest chunk.
<code>agent.tool.tokens_added</code>	required	Same meaning as for tools.

agent.guardrail

FIELD	LEVEL	NOTES
<code>agent.guardrail.name,</code> <code>agent.guardrail.version</code>	required	Which policy, which version.
<code>agent.guardrail.verdict</code>	required	<code>pass</code> , <code>block</code> , <code>modify</code> , <code>escalate</code> .
<code>agent.guardrail.target</code>	required	Span ID of what was checked.
<code>agent.guardrail.categories</code>	conditional	Which categories fired, without the matched text.



SECTION 6

Skipped options (rejected branches)

IN PLAIN TERMS

Whenever the agent could have done something and didn't, that something becomes a span with status `not_executed` and a reason code. It sits next to the options that did run, under the plan span that made the choice. So the trace shows the decision, not just the winner.

What counts as a skipped option: a tool call the model asked for that the runtime declined to run; an option the planner listed and didn't pick; an action a guardrail blocked; a retry that was budgeted and never attempted; a handoff that timed out before dispatch. They all get the same representation, so "why didn't it" is a single query.

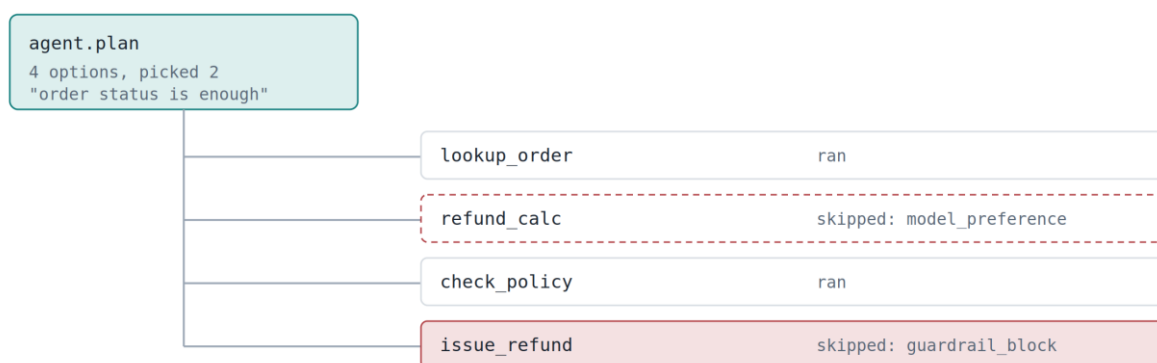


Fig 2. Ran and skipped options as siblings under the plan span. Reason codes turn "why didn't it" into a filter.

Reason codes (agent.branch.reason)

CODE	MEANING
<code>model_preference</code>	The model considered it and chose another path. Its one-line justification is on the plan span.
<code>guardrail_block</code>	A guardrail said no. Linked to the guardrail span.
<code>budget_exhausted</code>	Token, cost or step budget ran out first.
<code>timeout</code>	Prepared, but the deadline arrived.
<code>dependency_failed</code>	Something it needed had already failed.
<code>human_rejected</code>	Proposed to a person, declined. Linked to the override span.
<code>duplicate</code>	Deduplicated against an idempotency key.

**IMPLEMENTATION NOTE**

Product asks: "Why didn't it check the refund calculator?" The trace shows it. `refund_calc` is right there under the plan, skipped, reason `model_preference`, justification "order status is enough". Now the conversation is "was that rule right?" That's a prompt tweak, not a mystery.

Bonus: when a reviewer says a skipped option *should* have run, that span, with its full context, drops straight into the eval suite as a labelled failure. Free test cases.

Two implementation notes. Skipped spans should carry the same structural fields as ones that ran (tool name, args hash, side-effect flag), so they can be evaluated later. And they always carry zero cost; the whole point is that they didn't run.



SECTION 7

Human overrides

IN PLAIN TERMS

Any time a person changes what the agent would have done (approve, edit, reject, escalate), that's a span. It records who (pseudonymous), what action, what they changed, how long they took, and a reason code they pick from a short list. The reason code is required. It's the one field that turns "overrides went up" into something actionable.

FIELD	LEVEL	NOTES
<code>agent.override.actor</code>	required	Stable pseudonymous ID. Never a name or email in the trace store.
<code>agent.override.action</code>	required	<code>approve</code> , <code>edit</code> , <code>reject</code> , <code>escalate</code> .
<code>agent.override.target</code>	required	Span ID of what was reviewed.
<code>agent.override.reason_code</code>	required	From the list below. Free text doesn't substitute.
<code>agent.override.reason_text</code>	optional	The reviewer's words. Content mode applies.
<code>agent.override.replacement_hash</code>	conditional	For <code>edit</code> : hash of what shipped instead.
<code>agent.override.review_ms</code>	required	Time from shown to decided. A 2-second approve on a high-stakes action is its own signal.

Reason codes (`agent.override.reason_code`)

CODE	MEANING
<code>agent_wrong</code>	Wrong on the facts or the reasoning.
<code>stale_data</code>	Correct reasoning on out-of-date input. Points at a tool, not the model.
<code>policy_changed</code>	Correct against the agent's instructions, but the instructions are outdated.
<code>tone_or_style</code>	Substance fine, presentation not.
<code>risk_threshold</code>	Reviewer declined on judgement, not error. Too consequential to automate.
<code>customer_context</code>	Reviewer knew something the agent couldn't have.
<code>confirmed</code>	Approved, no change.

**IMPLEMENTATION NOTE**

Override rate climbs from 4% to 11% in two weeks. Grouping by reason code shows `agent_wrong` is flat. `policy_changed` tripled. Legal updated the refund policy; nobody updated the prompt. That's a one-line fix and a process gap, not a model regression. Without the reason code, a week would have gone into re-running evals for nothing.

Override rate by reason is the one chart that separates "the agent is drifting" from "the world moved". Total override rate can't. A fixed eval set can't either.

Reviewers will complain about the required category. In our experience it lasts about two weeks, then the chart becomes the most-watched thing on the dashboard. Keep the list short (under ten), stable, and versioned.



SECTION 8

Cost per step

IN PLAIN TERMS

Every span carries its own cost. Model calls price their tokens against a versioned rate card. Tools that call paid APIs carry that cost. Skipped spans and control spans (plan, guardrail, override) cost zero. Parents (root, plan, handoff) also carry a rolled-up total of everything beneath them.

RULE	WHY
Price at emission time; record the rate card version	Rates change. Re-pricing later while still knowing what it cost at the time depends on this.
Never estimate tokens the provider already reports	Estimates drift. If estimating is unavoidable, mark <code>agent.usage.estimated=true</code> .
Blame context growth on the span that caused it	<code>tokens_added</code> on tools and retrievals shows which step bloated every later model call.
Roll up through handoffs	A run that delegates to four sub-agents shows the delegated cost at the delegation point.
Put business dimensions on the root	Customer, feature, environment. Cost per customer is a join on the root, not on every child.

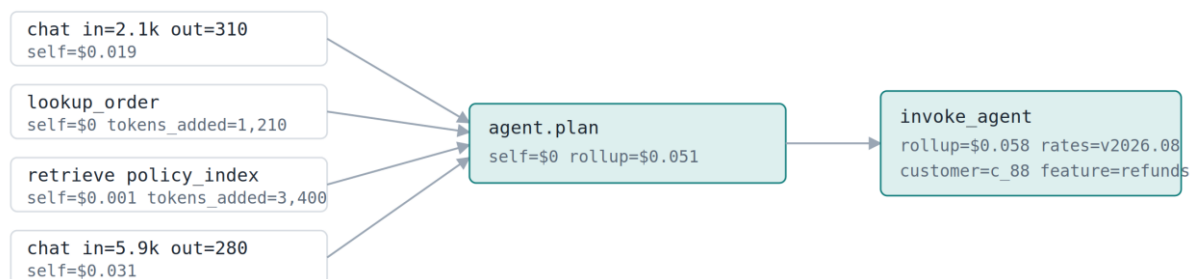


Fig 3. Self cost on leaves, roll-ups on parents. The retrieval's 3,400 `tokens_added` explains why the second chat call cost more than the first.

IMPLEMENTATION NOTE

Finance says agent spend tripled last week. Run totals confirm it and reveal nothing else. Sort tool spans by `tokens_added`: one retrieval now returns 8,000 tokens instead of 800, and every later model call re-reads all of it. Someone bumped `top_k` from 5 to 50. Reverting is enough, and the trace proves the cause.



SECTION 9

Keeping personal data out

IN PLAIN TERMS

Prompts, model outputs and tool results are the most useful things to log and the most likely to contain names, emails and account numbers. So: content travels as span events (never attributes), a processor scrubs it before it leaves the process, user IDs become keyed hashes so grouping by user is still possible, and timings and sequence are never touched. Every span records which mode was in effect.

MODE	WHAT'S CAPTURED	WHEN
<code>none</code>	No content events. Just hashes, sizes, timings, sequence, token counts.	Production in regulated contexts.
<code>sanitized</code>	Content events, after a processor replaced personal data with typed placeholders like <code>[EMAIL_1]</code> .	Production default elsewhere.
<code>full</code>	Content events unmodified.	Dev, consented shadow traffic, incident capture under a hold.

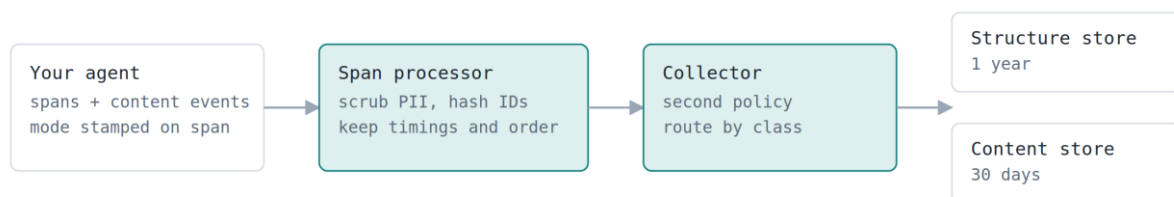


Fig 4. Scrub before export, again at the collector, and send content and structure to stores with different retention.

IMPLEMENTATION NOTE

Why the processor and not the app code? Because when security changes the redaction rule, cutting a release of every agent shouldn't be necessary. And because security wants one document that says what leaves the process. Putting the rules in the processor lets that document write itself.

Why hash IDs instead of dropping them? Because at 2 a.m. it's still necessary to say "show me every run for this user". A keyed hash provides that without storing who the user is.

One more rule: the redaction step logs itself. It emits which categories it detected, what it did, and its own version, without the matched text. So the redaction is auditable too.



SECTION 10

How long to keep what

IN PLAIN TERMS

Three classes, stamped on each span. Metadata (timings, counts, cost, hashes) for a year. Content for 30 days. Anything with a human decision or a side effect for at least six months, in storage that can't be quietly edited.

CLASS	COVERS	KEEP FOR	WHY
<code>metadata</code>	Span names, timings, status, tokens, cost, hashes, reason codes, roll-ups	12 months	Trends, capacity, cost review. No personal data.
<code>content</code>	Prompt, output, tool argument and result events	30 days	Debugging and eval sampling. Shortest window that serves both.
<code>decision</code>	Any span in a run with a human override, side-effect tool, guardrail block or escalation, plus its ancestors and the content of what was overridden	6 months minimum	Audit. Six months is the floor the EU AI Act sets for logs from high-risk systems. Treat it as a minimum, and confirm sector-specific rules separately.

IMPLEMENTATION NOTE

Six months after a refund incident, a compliance reviewer asks what happened. Content aged out at 30 days, so a per-run policy would leave nothing to show. But the override span was class `decision`, so it, its parent chain, and the overridden output are still there, exportable as plain JSON, and untouched since the day they were written. That's the difference between an audit trail and a log.



SECTION 11

A full example: spans and logs

A trimmed trace for a refund-support run in `sanitized` mode. Read the comments; the story is in them.

```
// ROOT. Which agent, which prompt, how it ended.
{ "name": "invoke_agent refunds-assistant", "span_id": "7f3a",
  "gen_ai.agent.name": "refunds-assistant",
  "agent.prompt.version": "2026.08.2", "agent.prompt.hash": "a91f...",
  "agent.tools.schema_hash": "6e0c...",
  "agent.content.mode": "sanitized", "agent.retention.class": "decision",
  "agent.outcome": "overridden", "agent.override.count": 1, "agent.branches.rejected": 2,
  "agent.cost.rollup_usd": 0.058, "customer.id": "c_88", "feature": "refunds" }

// PLAN. Four options, picked two, and said why.
{ "name": "agent.plan", "span_id": "b210", "parent": "7f3a",
  "agent.plan.candidates": ["lookup_order", "refund_calc", "check_policy", "issue_refund"],
  "agent.plan.chosen": ["lookup_order", "check_policy"],
  "agent.plan.justification": "order status sufficient; calculator only if amount disputed",
  "agent.cost.rollup_usd": 0.051 }

// TOOL THAT RAN. Note the age of the data it returned.
{ "name": "execute_tool lookup_order", "parent": "b210",
  "agent.tool.status": "executed", "agent.tool.side_effect": false,
  "agent.tool.result_age_s": 2460, "agent.tool.tokens_added": 1210,
  "events": [ { "name": "gen_ai.tool.call.result",
    "body": "{ order: ..., email: [EMAIL_1] ... }" } ] }

// SKIPPED OPTION. The model preferred not to.
{ "name": "execute_tool refund_calc", "parent": "b210",
  "agent.tool.status": "not_executed", "agent.branch.reason": "model_preference" }

// BLOCKED OPTION. A guardrail stopped the side-effect tool.
{ "name": "execute_tool issue_refund", "parent": "b210",
  "agent.tool.status": "not_executed", "agent.branch.reason": "guardrail_block",
  "agent.tool.side_effect": true, "agent.guardrail.ref": "g_55" }

// MODEL CALL that wrote the draft answer. Lists everything it read.
{ "name": "chat", "span_id": "9c02", "parent": "7f3a",
  "gen_ai.usage.input_tokens": 5900, "gen_ai.usage.output_tokens": 280,
  "agent.context.sources": ["tool:call_01", "retrieve:r_09", "msg:u_1"],
  "agent.cost.usd": 0.031,
  "events": [ { "name": "gen_ai.output.messages",
    "body": "...your refund has been approved..." } ] }

// HUMAN OVERRIDE. Who, what, and a reason code that points at the tool.
{ "name": "agent.human_override", "parent": "7f3a",
  "agent.override.actor": "u:hm-2f81", "agent.override.action": "reject",
  "agent.override.target": "9c02", "agent.override.reason_code": "stale_data",
  "agent.override.reason_text": "order updated 40 min ago, refund already reversed",
  "agent.override.review_ms": 41200, "agent.retention.class": "decision" }
```

**TRACE WALKTHROUGH**

Read top to bottom and the incident explains itself. The plan chose two tools out of four and said why. The order lookup returned data 41 minutes old. The model reasoned correctly on it and drafted "approved". The reviewer rejected it and picked `stale_data`, which points at the tool, not the model.

The fix is a cache policy on `lookup_order`. The eval suite gains a labelled case: this context, with this stale input, should have triggered a re-fetch. And the whole thing is kept for six months because it carries a human decision.

The same run, as a log stream**IN PLAIN TERMS**

The span JSON above is what the backend stores. The lines below are the same run rendered as a live log, the view a developer actually tails. The schema doesn't mandate a log format; this is one reasonable rendering of the same spans, shown to illustrate how the fields read in real time. All values are dummy data.

```
$ tail -f agent.log
10:41:02.114 INFO run=7f3a invoke_agent refunds-assistant
prompt=2026.08.2 (a91f) tools_schema=6e0c mode=sanitized
10:41:02.890 INFO run=7f3a plan: candidates=[lookup_order, refund_calc,
check_policy, issue_refund] chosen=[lookup_order, check_policy]
why="order status sufficient; calculator only if disputed"
10:41:03.104 WARN run=7f3a tool lookup_order ok (210ms) result_age=41m
tokens_added=1210 side_effect=no [stale: age > 15m threshold]
10:41:03.106 INFO run=7f3a tool refund_calc SKIPPED reason=model_preference
10:41:03.107 INFO run=7f3a tool issue_refund SKIPPED reason=guardrail_block
guardrail=side_effect_review verdict=block
10:41:03.290 INFO run=7f3a tool check_policy ok (180ms) tokens_added=640
cost=$0.004
10:41:08.201 INFO run=7f3a chat in=5.9k out=280 finish=stop cost=$0.031
ctx=[tool:call_01, retrieve:r_09, msg:u_1]
10:41:08.223 INFO run=7f3a draft ready, held for review (side-effect path)
10:41:49.410 WARN run=7f3a HUMAN OVERRIDE actor=u:hm-2f81 action=reject
target=chat:9c02 reason=stale_data review_took=41.2s
note="order updated 40 min ago, refund already reversed"
10:41:49.412 INFO run=7f3a outcome=overridden cost_total=$0.058
overrides=1 skipped=2 retention=decision
```

PRACTITIONER NOTE

The stale data announces itself at 10:41:03 as a WARN, five seconds before the model drafts the wrong answer, because `result_age_s` crossed a threshold the processor knows about. Alerting on the cause beats alerting on the symptom.

The two skipped options appear in the stream with reasons attached. "Why didn't it call `issue_refund`" is answered by a line that was already there, not by an investigation.

The override line alone (actor, target, reason, note) is a complete incident summary. Everything a post-mortem needs is one grep away: `grep "run=7f3a" agent.log`.



SECTION 12

How it maps to OpenTelemetry

The schema emits through any OTel SDK and lands in any OTLP backend. Where the GenAI conventions define a name, we use it. Where they're silent, we add an `agent.*` field. The conventions are still moving, so keep the extension namespace; when a concept graduates into the standard, the change is a rename in one processor.

CONCEPT	OTEL GENAI TODAY	THIS SCHEMA ADDS
Agent run	<code>invoke_agent</code> , <code>gen_ai.agent.*</code>	Prompt and tool-schema hashes, outcome, counts.
Model call	<code>chat</code> , <code>gen_ai.request.*</code> , <code>gen_ai.usage.*</code>	Context sources, cost, rate card version.
Tool call	<code>execute_tool</code> , <code>gen_ai.tool.*</code>	Status, side effect, idempotency, result age, tokens added.
Content	Events for messages and tool args/results; capture modes	<code>agent.content.mode</code> stamped on the span.
Skipped option	Not defined	<code>status=not_executed</code> plus <code>agent.branch.reason</code> , under <code>agent.plan</code> .
Human override	Not defined	<code>agent.human_override</code> span with actor, action, target, reason code.
Cost	Token counts only	<code>agent.cost.usd</code> , roll-ups, pricing version.
Retention	Not in scope	<code>agent.retention.class</code> on every span.



SECTION 13

How to adopt it, step by step

The reference architecture

IN PLAIN TERMS

Before the levels, here is the target picture: where each piece of the schema lives in an ordinary observability stack. Nothing here requires a new vendor. It requires putting redaction and retention in the right place in a pipeline that is probably already running.

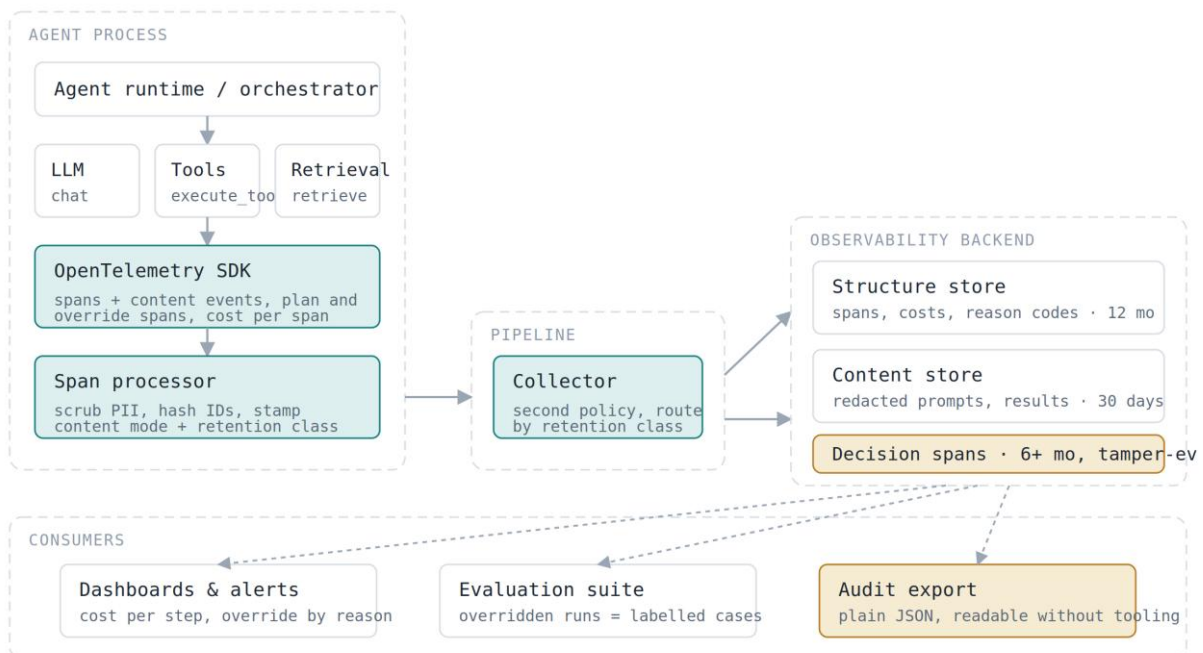


Fig 5. The reference architecture. The span processor sits inside the process boundary so nothing unscrubbed ever leaves. The collector routes by retention class, one pipeline serving three stores. The evaluation suite and audit export are consumers of the trace, not separate systems.

Three placement decisions carry all the weight. Redaction happens in the span processor, *inside* the process, so content is scrubbed before it crosses any network (Section 9). The collector routes by `agent.retention.class`, which is how one pipeline serves a 30-day content store, a 12-month structure store and a six-month tamper-evident decision store without duplication (Section 10). And the evaluation suite reads from the same backend as the dashboards, which is what turns every override and every wrongly skipped option into a labelled test case with zero extra plumbing (Sections 6 and 7).



The three levels

Not everything has to happen at once. Three levels; each is useful on its own and each answers questions the last one couldn't.

LEVEL 1

Standard spans, real inputs

Emit OTel GenAI agent, chat and tool spans from the framework already in use. Add prompt and tool-schema hashes, context sources, and content events in sanitized mode. This answers: what did the model see?

LEVEL 2

Decisions and cost

Add plan spans with skipped options and reason codes, cost on every span, side-effect flags and idempotency keys. This answers: why this and not that, and what did each step cost?

LEVEL 3

Human decisions and retention

Add override spans with required reason codes, retention classes, and tamper-evident storage for the decision class. This answers: is the agent drifting or the world, and can what happened be proven?

FROM DAY ONE

Redaction in the processor

Content leaves the process only through a processor that scrubs and hashes. This is the one item that is expensive to retrofit. Do it first.

ADOPTION GUIDANCE

Day 1: turn on the framework's OTel instrumentation and point it at any OTLP backend. Day 2: add the prompt hash and tool-schema hash to the root span. Day 3: add a span processor that scrubs emails and phone numbers and hashes user IDs. Day 4: add `tokens_added` to tool spans and cost to chat spans. Day 5: wrap the planner so skipped tools emit `not_executed` spans. This reaches Level 2, where most incidents explain themselves from the trace.



Checklist

- ☐ Every model call span records model, sampling settings, provider-reported tokens, and a reference to every input in its context.
- ☐ Prompt version and hash, and tool-schema hash, are on the root span of every run.
- ☐ Tool spans carry a side-effect flag and, when true, an idempotency key.
- ☐ Tool and retrieval spans record how many tokens they added to the next model call.
- ☐ The trace can show an option that was considered and skipped, with a reason code from a fixed list.
- ☐ Human overrides are spans with a pseudonymous actor, an action, a target, and a required reason code.
- ☐ Cost is on every span, priced against a versioned rate card, and rolled up at plan, handoff and root.
- ☐ Content is carried as span events, never attributes, and scrubbed in a span processor before export.
- ☐ Retention is set per span by class; decision-class spans are kept at least six months in tamper-evident, exportable storage.
- ☐ Extension fields live under one namespace, and the mapping to OTel GenAI is documented.



SECTION 14

References

The following standards, specifications, and regulatory instruments are cited in this document or inform its design decisions.

Standards and specifications

REFERENCE	DESCRIPTION
OpenTelemetry GenAI Semantic Conventions — open-telemetry/semantic-conventions-genai	The primary specification this schema builds on. Defines span kinds (<code>invoke_agent</code> , <code>chat</code> , <code>execute_tool</code> , <code>retrieve</code>), the <code>gen_ai.attribute namespace</code> , and <code>content capture events</code> . <i>Status: Development as of this writing. All extensions in this document are placed under the <code>agent</code> namespace so they migrate cleanly when upstream converges.</i>
OpenTelemetry Specification — open-telemetry/opentelemetry-specification	The core OTel specification defining trace, span, context propagation, and the OTLP wire protocol. The schema relies on standard span context (trace ID, span ID, parent span ID) and the OpenTelemetry SDK span processor interface.
OpenTelemetry Semantic Conventions — open-telemetry/semantic-conventions v1.40+	General semantic conventions repository. GenAI conventions were moved to the dedicated repository above at v1.40.0 (February 2026). Attribute naming conventions, stability definitions, and the experimental/development status model all follow this specification.
W3C Trace Context — W3C Recommendation	Defines the <code>traceparent</code> and <code>tracestate</code> HTTP headers used to propagate span context across process and service boundaries. Relevant for multi-agent runs and handoff spans that cross network boundaries.
OTLP — OpenTelemetry Protocol Specification	Wire protocol for exporting spans, metrics and logs to any OTLP-compliant backend. The schema is transport-agnostic; it emits standard OTLP and lands in any compliant backend (Grafana, Datadog, Honeycomb, Jaeger, and others).

Regulatory instruments

REFERENCE	RELEVANCE TO THIS SCHEMA
Regulation (EU) 2024/1689 — EU Artificial Intelligence Act, Article 12 (Record-Keeping)	Requires high-risk AI systems to include automatic, continuous logging over the lifetime of the system. The six-month minimum retention floor for the decision retention class in Section 10 is drawn from Article 19 of this regulation. Enforcement began August 2026 for high-risk systems under Annex III.
Regulation (EU) 2024/1689 — EU Artificial Intelligence Act, Article 19 (Retention of logs)	Sets the minimum retention period for automatically generated logs at six months, unless Union or national law requires longer. Cited directly in the decision retention class definition in Section 10.
EU General Data Protection Regulation (GDPR) — Regulation (EU) 2016/679	Informs the redaction and pseudonymisation design in Section 9. The requirement to scrub personal data in a span



REFERENCE	RELEVANCE TO THIS SCHEMA
	processor before export and to hash user IDs rather than drop them is consistent with GDPR Article 5 (data minimisation) and Article 25 (data protection by design).

Further reading

SOURCE	DESCRIPTION
How OpenTelemetry Traces LLM Calls, Agent Reasoning, and MCP Tools — greptime.com, May 2026	Detailed walkthrough of the six-layer structure of the OTel GenAI conventions: LLM client tracing, agent orchestration, MCP tool calling, content capture, metrics, and quality evaluation. Useful companion to Section 12 of this document.
Agent Tracing with OpenTelemetry (2026): How It Works and the Tools — morphllm.com, June 2026	Covers the span tree model for multi-step and sub-agent runs, and surveys the major tracing backends (Langfuse, Arize Phoenix, OpenLLMetry, Braintrust, LangSmith) and their adoption of the GenAI conventions.
EU AI Act Requires Automatic Logging for High-Risk AI — letsdatascience.com, 2026	Technical analysis of the Article 12 logging mandate: what "technical", "automatic", and "lifetime" mean as engineering constraints, and why manual exports do not satisfy the regulation.
Article 12 and the Logging Mandate — securityboulevard.com, April 2026	Practical guide to what Article 12 requires from engineering teams, including the distinction between an audit trail and a standard application log, and the argument for treating logging as a functional requirement from day one.



ABOUT THE AUTHORS



Ujjawal Rana

MAIN AUTHOR

Ujjawal Rana led the research, design, and writing of this whitepaper. His work focuses on AI agent observability, production tracing infrastructure, and building practical tooling for teams deploying large language model systems at enterprise scale.



Snehal

CO-AUTHOR

Snehal contributed to the technical research, schema validation, and development of the worked examples and field-level specifications in this document.



Aashish Pandey

REVIEWER

Aashish Pandey conducted the technical review of this document, validating schema correctness, implementation guidance, and the accuracy of regulatory references.



Blue Data Consulting

Blue Data Consulting is a B2B enterprise AI and IT consulting firm operating across India, Malaysia, and the UAE. The firm helps organisations in the telecom, semiconductor, and healthcare sectors build and deploy enterprise-grade AI systems, from infrastructure design and model integration to governance frameworks and workforce enablement.

Blue Data's two practice areas are Enterprise IT Solutions, covering AI platform architecture, systems integration, and production deployment; and Enterprise Upskilling and Corporate Training, helping technical and business teams develop the skills to operate AI systems responsibly and effectively.

This whitepaper is part of Blue Data's ongoing technical publishing programme. Blue Data publishes research and reference specifications at the intersection of AI engineering and enterprise operations, with a focus on artefacts that practitioners can implement directly.

bluedataconsulting.co