
Agent Session Record:

an open format for recorded agent sessions

A white paper and call for review

VERSION

vo.1 draft

DATE

October 2026

AUTHORS

Wills Manley

ABSTRACT

Agents now do work that people need to check later. Coding agents plan, edit, test and commit. Personal assistants on messaging apps answer group chats, act on schedules, send replies and keep notes about the people they work for. Some apps run other agents and add their own queues, approvals and delegated tasks. Each harness, the software that runs the agent, stores the session privately in its own format. Transcripts are deleted by retention defaults, rewritten in place, copied wholesale on fork or resume, and almost never linked to the commits they produced. Many of these stores cannot say which person in a group chat asked for something, what started a turn that nobody typed, who approved an action, or where a remembered fact came from. Tools that search, review, attribute, evaluate or meter agent work must reverse-engineer each private format, and they still cannot tell what is missing from what never happened.

Agent Session Record (ASR) is a proposed vendor-neutral, append-only JSON Lines (JSONL) format for keeping a lasting record of an agent session. It records what was asked and by whom, what started each turn, what the model received and produced, which tools ran and what they returned, which approvals and questions went to a person and how they were answered, what the agent wrote to memory, which replies were delivered, which files changed, how sessions spawned, forked, continued or ran one another, how context was compacted, what it cost, and which commits resulted. Version 0.1 is fitted first to four tools: Claude Code, T3 Code, Codex and OpenClaw. ASR is designed as the archival counterpart of the Agent Client Protocol (ACP), much as SARIF is the archival counterpart of LSP diagnostics. Existing logs can be converted today without vendor help. We invite review before the format reaches Preview.

Contents

1	The problem	3
2	Who needs a common record	5
3	What exists today, and why it is not enough	6
4	What MCP and ACP taught us	8
5	Design principles	9
6	A short tour of the format	10
7	Key design decisions	14
7.1	Identity	14
7.2	Immutable records and revisions	15
7.3	Order, branches and the context view	15
7.4	Lineage as independent facets	16
7.5	Inference and usage	16
7.6	File changes as evidence, not proof	17
7.7	Capture states, gaps and tombstones	17
7.8	Privacy and security	18
7.9	Profiles: what a producer promises to record	18
7.10	Extensibility and versioning	18
7.11	ACP alignment	19
7.12	Senders, messages between agents, and triggers	19
7.13	Approvals, questions and memory	20
7.14	What enters the standard	20
8	Bridges and companions	21
9	Adoption	22
9.1	For harnesses	22
9.2	For tool builders	22
9.3	What we ask of vendors	23
10	Governance and roadmap	23
11	Call for review	24
	References	26

1 The problem

An agent session is no longer a chat log. One session can read a repository, run tests, edit dozens of files, start subagents, open a pull request and spend money on inference. Another can run for months behind a messaging app: it answers several people in a group chat, wakes on a schedule, asks for approval before it acts, sends replies and writes down what it learned. When a reviewer asks why a change was made, when a team asks what its agents cost, when someone asks who told the assistant to send a message, or when an evaluation team wants realistic trajectories, the session is the primary evidence: the request, the plan, the failed attempts and the approach that was finally taken.

Yet agent sessions have no common format. Source code has git. Test and analysis results have JUnit XML and SARIF. Traces have OpenTelemetry. Agent sessions live in each harness's private store (JSONL files, SQLite databases, compressed blobs, protobuf caches), each with its own idea of identity, order and completeness. We surveyed the on-disk formats of more than forty harnesses, mostly through their public source code, documentation and changelogs, and read the stores of open-source personal assistants such as OpenClaw [1] and Hermes Agent [2]. The differences change what a consumer can safely conclude. Twelve failure modes recur. The first eight appear across coding agents; the last four matter most for assistants and for apps that run other agents.

Retention deletes history by default. Some harnesses sweep old local transcripts on a schedule. Claude Code, for example, removes local transcripts after a configurable cleanup period, 30 days by default [3], and Gemini CLI documents a retention policy with a maximum age [4]. That is a sensible default for files that can contain secrets, but the record of how code was written then disappears unless something else keeps it, and nothing marks what was removed.

Stores are mutable, and some are destructive. Several harnesses rewrite a whole document on every save: Zed stores each thread as one compressed blob that is replaced as the thread changes [5], and Cline-family extensions rewrite their task files whole [6]. Goose deletes and reinserts a conversation's rows when replacing it, and truncation deletes rows [7]. Gemini CLI's JSONL is append-only on disk, but a later line with the same ID replaces an earlier one, rewind markers cut the logical history, and stored tool results can be replaced by their masked form after the model saw them [8]. Codex compresses and moves infrequently accessed session log files, so a session's path is not stable [9]. An importer that assumes append-only behavior re-imports everything or silently loses rewound and deleted turns.

History is copied on fork and resume. Forking often copies the parent's records into the child. Claude Code stamps copied records with a `forkedFrom` pointer. Codex's older fork mode copies items into the child session log, while its newer paginated mode references the initial part of the parent's log instead [9]. Pi and Goose also copy [10, 7]. A consumer that does not know which records are copies double-counts tokens, cost and edits, returns duplicate search results, and credits the child with the parent's work.

Parent pointers are ambiguous. One harness uses a single parent column for four relations: delegation, branching, and continuation after compaction or after a reset. Another writes subagent transcripts that carry the parent's session ID. Another stores a parent reference that is sometimes an ID and sometimes a file path. Inside tree-shaped transcripts, most sibling records come from parallel tool calls, attachments and retries, not real branches. A consumer that reads every parent pointer as a fork invents branches; one that ignores them loses real rewinds.

Reasoning is often encrypted or omitted. Providers return reasoning as plain text, summaries, signatures with the text omitted, encrypted blobs or redacted blocks. Codex, for instance, keeps reasoning as encrypted content whose summaries are often empty, and remote compaction summaries are encrypted too [9]. A record must preserve content it cannot interpret, without indexing it as text or presenting a summary as the full reasoning.

Tool output is truncated, masked or moved elsewhere. Large outputs are written to side files that retention later deletes, replaced by masked versions, or stored only in the truncated form the model saw. Some formats keep a command's exit code only inside the model-facing output text. Consumers then treat previews as complete.

Nothing links edits to commits. No harness we surveyed records which commit included which agent edit. The nearest are Cursor's separate AI code-tracking store, which maps AI-written lines to commits outside the transcript, Claude Code's pull-request link records, Codex's git state at session start, and Zed's snapshot of the uncommitted diff when a thread begins. Anyone else who wants attribution must infer it later by matching diffs.

Time is missing, coarse or out of order. Some stores keep only a thread-level update time. One transcript carries minute-precision local time as text inside the user's prompt. Others record whole seconds. In some JSONL logs, timestamps run backwards relative to file order around attachments and copied records. Only a few harnesses write an explicit sequence number.

Who spoke is often lost. In a group chat, several people talk to the same agent, and other agents may send it messages too. Many formats record only that a message came from "the user". One assistant we read keeps a sender's name only as a prefix inside the message text. Another stores a sender ID on each message and says how that identity was established. A reader that sees only "user" cannot tell which person asked for an action.

Turns start without a prompt. Agents wake on schedules, heartbeats, webhooks and finished background tasks, and they receive messages from other agents. When an app runs another agent, the backing agent's transcript shows the app's scheduled prompts, wake notices and messages from other threads as ordinary user prompts. Only the app's own records say that no person typed them. A reader cannot tell "the user asked" from "a job fired at 8 a.m."

Approvals are not always kept. Agents ask before they run commands, send messages or change shared data, and some let a model review the request. Codex does not write approval requests or decisions to its session log [9]. OpenClaw records who resolved each approval (a device, a chat channel or the runtime), when it expires and what a model reviewer concluded [1]. Without a record of the approval, an action looks the same whether a person allowed it, a rule allowed it or nobody was asked.

Memory changes leave no trail. Coding agents keep memory files between sessions, and assistants keep notes about the people they work for. Memory is written by file edits, by memory tools and by background runs that have no transcript of their own. A record that shows only which memory was loaded cannot tell a user where a remembered fact came from or when it was written.

None of these are bugs in the usual sense. Each is a reasonable choice for a tool whose job is the live session. They become problems when someone needs the session after it ends, and by then the information needed to read it correctly is often gone.

Much of the information needed for a lasting record already exists somewhere. Codex writes a sequence number on each record of a session log and records Model Context Protocol (MCP) server and tool names as separate fields. Claude Code captures environment snapshots, instruction files and permission decisions, and marks copied records. Gemini CLI separates model-facing from display content and records token breakdowns per message. OpenHands records the full system prompt and tool definitions, and models compaction as an overlay that hides events without deleting them. OpenCode distinguishes queued input from steering (instructions sent while the agent is working on a turn). Copilot CLI's rewind restores only the files Copilot changed [11]. Goose records where each cost figure came from. Claude Code marks whether a turn was started by a person, a schedule, a finished background task or another agent. T3 Code, an app that runs Claude Code and Codex sessions, records which prompts a person typed and which it injected for a schedule, a finished task or another thread [12]. OpenClaw records whether each reply was delivered, suppressed or failed. The problem is not that harnesses record too little. It is that every consumer must learn each harness's choices separately, and nothing tells it when a choice changes.

A note on authorship. The authors also maintain ctx [13], an open-source tool that searches agent sessions across harnesses, and ctx is one converter implementation for this format. ASR does not depend on ctx. Its 1.0 criteria require two independent implementations, and we want maintainers from other organizations.

2 Who needs a common record

Seven uses drive the design. Each is something teams already build against private formats, one harness at a time.

1. **Search and memory across harnesses.** People move between agents within a day. Finding “the session where we decided to drop the v1 API” or “the chat where the assistant booked the flight” should not depend on which agent was open. Memory tools that feed past sessions into new ones also need to mark retrieved text so it does not become its own evidence.
2. **Portable handoff.** Seed a new session, in any harness, from a prior record. This is weaker than resuming a session in any harness. A record does not restore processes, credentials, sandboxes or provider-private continuation state, and replaying a record for display never re-executes tools. Resuming execution in the native harness is an adapter capability, qualified per harness.

3. **Review and audit.** Reviewers want to see what the agent was asked and by whom, what started each turn, what it tried, which commands it ran with which exit codes, who or what approved each action, which replies it sent and what it wrote to memory, in a form that outlives the harness’s retention window.
4. **Attribution and blame evidence.** Answering “Which session wrote this line?” needs edits recorded with enough identifying detail to match commits, and lineage recorded accurately enough that a fork is not credited with its parent’s work.
5. **Evaluation and training export.** Trajectories with exact tool calls, results and token accounting, with copied context excluded.
6. **Cost and usage analytics.** Per-call token counts and cost, with an explanation of how cached tokens are counted, without double-counting subagents, copies or cumulative snapshots.
7. **Replay viewers.** Displaying a session accurately requires the active branch, compaction boundaries, and both the content shown to the model and the content shown to the user.

Four tools first. Version 0.1 is fitted to four tools that between them cover these uses across different kinds of agent: Claude Code and Codex, two coding agents; T3 Code, which runs other agents and adds its own queues, approvals and delegated tasks; and OpenClaw, a personal assistant that answers on messaging apps. Our evidence is a field-by-field reading of what each tool writes, from its source code or installed program and, for three of them, from local session data. Everything these four tools record has a place in ASR.

One rule keeps the core small: **a field or record type is standard when one of these uses needs it and at least two of the four tools record it.** The few exceptions are fields that are the only way to keep a concept ASR already has, each with a written justification. Every other fact a tool records gets a documented destination in that tool’s registry entry: a vendor `_meta` key on the nearest record, or a vendor record type. There is no experimental tier. A feature that is not ready is left out and listed as future work (Section 10).

3 What exists today, and why it is not enough

Several existing formats address related problems. Each was built for a different job and does that job well. None was built to keep a lasting session record across harnesses.

Agent Client Protocol (ACP). ACP, developed by Zed and JetBrains, is the live JSON-RPC protocol between an editor and an agent [14]. Its content model (MCP content blocks, typed tool calls, structured diffs, plans, permissions) is the closest prior art for what a session contains, and the v2 draft makes IDs mandatory and opens every enum [15]. But ACP is a live view: no per-entry timestamps, no lineage between entries, whole-session fork only, no links to commits, and live-only messages may be left out of replay. ASR is designed as ACP’s archival counterpart, not a competitor.

ATIF. The Agent Trajectory Interchange Format from the Harbor framework, home of Terminal-Bench, is the most widely adopted trajectory format for evaluation and training [16, 17]. Its token accounting is explicit, its `is_copied_context` flag handles copied history, and v1.7 added

embedded subagent trajectories. It is one JSON document with linear steps, so it cannot be appended to as a session runs, and it has no fork tree, file-edit model or commit linkage. It is well suited as an export target.

GitHub Unified Agent Session (UAS). UAS specifies how GitHub Agentic Workflows normalizes logs from several engines [18]. Its normalization rules are the best we found: absence is not zero, never fabricate IDs or timestamps, array position defines order, and dangling or orphan tool calls stay explicit. ASR adopts several of them nearly word for word. UAS is tied to one CI runtime’s vocabulary and has no file-edit, commit or fork model.

OpenTelemetry GenAI semantic conventions. OTel defines spans for agent invocation, inference and tool execution, and a message-part vocabulary that is the closest thing to neutral content types across providers [19]. It is telemetry: content capture is off by default, data may be sampled, and export batches are unordered. A session record should emit OTel spans, not replace them.

Agent Trace and Git AI. Both describe line attribution: which ranges at a revision came from which conversation [20, 21]. Git AI runs in production, specifies how attribution survives rebase, squash and cherry-pick, and keeps transcripts out of pushed git notes [22]. Agent Trace’s 0.1.0 RFC from January 2026 defines a clean per-revision record. Both are attribution views, and both need a session record to point at.

Entire. Entire’s CLI stores sessions alongside commits [23]. Its design is the closest to ours: it keeps the native transcript next to a normalized one, uses an append-only serialized format whose records are combined to reconstruct state, and links commits with an Entire-Checkpoint trailer. Its normalized transcript is defined and versioned by one product, and it drops reasoning blocks [24].

Format	Built for	Shape	Lineage and branches	Code linkage	Status (Oct 2026)
ACP v2	Live editor↔agent protocol	JSON-RPC update stream	Subagent sessions (proposed); whole-session fork	Structured diffs; no commits	v1 stable; v2 draft
ATIF	Evaluation and training trajectories	One JSON document	Embedded subagent trajectories; linear steps	None	Active, v1.8
GitHub UAS	Normalizing CI agent logs	JSONL trace plus merged artifact	Parent IDs kept, not interpreted	Workflow commit and ref	Project spec, draft
OTel GenAI	Telemetry	Spans and events	Span tree (call tree)	None	Development
Agent Trace	Line attribution	One record per revision	None	Line ranges at a revision	RFC 0.1.0
Git AI	Line attribution in git	One git note per commit	None	Line ranges; rewrite rules	v3.0.0, in production
Entire	Sessions stored with commits	Native plus normalized transcript	Subagent tasks	Checkpoint trailer	Versioned with the product
ASR	Lasting session record	Append-only JSONL per session, plus bundles	Lineage facets, branches, heads	File changes with object IDs; commit records	Draft

Lessons from outside AI. Two analogies shaped the design. The first is LSP and SARIF. LSP diagnostics provide a lightweight, live channel to an editor. SARIF is a larger format for stored results exchanged by analysis tools and GitHub code scanning [25, 26]. The two were never merged, and both are better for it. ACP and ASR divide the work the same way. The second is LSIF and SCIP, two formats for stored code indexes. LSIF was a persisted dump of LSP-style data, stored as a graph of numeric vertices and edges. Sourcegraph replaced it with SCIP because the graph was hard to debug, one wrong ID could break navigation across a repository, and the output was large [27]. ASR therefore uses self-contained records with readable native IDs rather than mirroring a wire protocol’s event graph.

4 What MCP and ACP taught us

MCP and ACP are the two agent protocols this ecosystem has developed in the open. Both changed course on points that matter for a record format. We read their changelogs and design documents and took six lessons.

Sessions need defined semantics, or they mean nothing portable. MCP shipped protocol-level sessions whose scope each implementer chose: per tool call, per application launch, per page load. They had not converged after more than a year, and the 2026-07-28 revision removed them in favor of explicit handles [28, 29]. ASR defines its container precisely: one stream per session, a key built from the native ID, and lineage stated as named facts rather than a bare parent pointer.

IDs added late are expensive. ACP launched without message IDs. Clients could “only infer message boundaries by detecting a change in the `sessionUpdate` type”, and implementations worked around this in `_meta` [30]. Message IDs became optional in v1 in 2026 and required in v2, along with stable IDs for tool calls, terminals, plans and compactions. ASR requires a record ID on every record from the start, carries native entity IDs verbatim, and marks any ID a converter had to derive.

Closed enums break validators. SDKs and validators generated from ACP’s schema could reject a message only because a newer version had added an enum value. ACP v2 opens every enum and reserves `_`-prefixed values for implementations [31]. Every ASR enum is open, unknown values are preserved, and vendor values use ACP’s `_` prefix, so ACP values pass through a recorder untouched.

Experimental features do not belong in core. MCP made JSON-RPC batching mandatory in one revision and removed it in the next [32]. It shipped tasks in core as experimental, then moved them to an extension after implementers hit design problems [33]. A formal deprecation lifecycle came only afterwards [34]. ASR has no experimental tier. A feature is in the format only when at least two of the four tools it is fitted to record it; a feature that is not ready stays out and is listed as future work. Deprecation is defined before 1.0. Because records are archival, “removed” means writers stop emitting a feature, never that readers stop accepting it.

The turn is not the unit of work. ACP v1 tied work to the prompt request and its response. Background subagents, queued and steered input, and output the agent starts on its own forced v2 to report work as independent state updates. The prompt response then means only that the message was inserted [35]. ASR models a session as an ordered timeline of typed records. A turn is an optional grouping defined by the source, and state changes (`queued`, `running`, `idle`, `requires_action`, `ended`) are records of their own. What started a turn (a person, a schedule, an event, another agent or a finished background task) is a `trigger` on the turn's first record.

Diffs need structure, not old and new text. ACP v1's `{path, oldText, newText}` could not tell a deleted file from an emptied one or express moves, copies and binary files, and many agents filled it with edit snippets that cannot produce a patch. v2 replaced it with an authoritative `changes[]` list plus an optional git patch [36]. ASR adopts that shape and adds information needed for historical records: paths relative to declared workspace roots, optional before and after object IDs, and a required `precision` field. The field identifies snippet-only edits so they are not presented as patches.

Two more choices follow. ACP v2 defines a uniform patch rule for live updates; ASR stores full replacement records instead, so a reader never needs a patch engine. MCP now requires a conformance scenario before a specification change is final [37]; ASR will ship its validator, schemas and golden corpus before Preview.

5 Design principles

1. **Record what happened.** The record holds literal facts: this call exited with code 101, this commit was observed. Whether a session “produced” a commit or “owns” a line is a claim for downstream tools to derive.
2. **Keep native IDs.** The harness's own session, message and call IDs are copied verbatim, and a derived ID says so. Paths, import times and content hashes are never identity.
3. **State what is missing.** An absent value means unknown. A truncated output is labeled truncated, and missing history becomes an explicit gap. Zero, false and empty are facts, distinct from absence.
4. **Append-only and crash-safe.** Records never change once written. An incomplete final line is ignored; a corrupt line in the middle produces a diagnostic and a gap.
5. **A small required core, with capability-based profiles.** The core is a header, messages, tool calls and results, ordering and capture states. Coding, lineage, usage and messaging are profiles a producer claims.
6. **Aligned with the live protocol.** ASR reuses ACP v2's entity names, content blocks, tool kinds, diff shape and enum conventions, and the mapping declares its losses.
7. **Links need evidence.** No lineage, copy or link is inferred from similar text or timing.
8. **Designed for redaction.** Disclosure levels, redaction records, redactable workspace roots and safe-reader rules are part of the format.

9. **Preserve what you don't understand.** Tools that store or rewrite a stream keep unknown fields, record types and enum values.
10. **Ship with tools.** A validator, converters and a corpus of sanitized real sessions are part of the specification.

6 A short tour of the format

An ASR stream is one session: UTF-8 JSON Lines, one record per line, no pretty-printing, written by exactly one writer. Line 1 is the `session` header. Every record has a `type`, an `id` unique within the stream, and a `seq`, a strictly increasing integer that defines observation order. Timestamps are optional, never fabricated, and carry their precision and source. Machine-readable identifiers use `agent-session` (for example the media type `application/vnd.agent-session+jsonl`); “ASR” appears only in prose.

Area	Record types
Session	<code>session</code> (header, with participants and conversations for group chats), <code>session_info</code> , <code>declaration</code>
Conversation	<code>message</code> , <code>tool_call</code> , <code>tool_result</code> , <code>inference</code> , <code>plan</code>
Approvals and questions	<code>permission</code> , <code>input_request</code>
Processes and hooks	<code>terminal</code> (background processes and monitors), <code>hook</code>
Structure	<code>subagent</code> , <code>compaction</code> , <code>rewind</code>
State	<code>state</code> (queued, running, idle, requires_action, error, resumed, ended)
Context and memory	<code>config_change</code> , <code>instructions</code> , <code>memory_change</code>
Messaging	<code>delivery</code>
Coding	<code>file_change</code> , <code>vcs</code> , <code>workspace_state</code> , <code>checkpoint</code>
Missing history	<code>gap</code> , <code>tombstone</code>
Usage	<code>usage_summary</code> , plus per-call usage and the exact request (<code>inference.context</code>) on <code>inference</code>

A turn's first record can carry a `trigger` that says what started it. A message from another agent or session is an ordinary `message` with `origin: inter_session` and a `from` that names the sender.

The example below follows a short coding session. It was converted from a fictional harness's native log to illustrate the conversion process available today. Records are wrapped for print, hashes are abbreviated, and `...` marks omitted values; in a real stream each record is a single line.

The header names the harness (`agent`) and, separately, the tool that wrote the stream (`producer`). Here a converter read the native log and declares in `lossy` that the source's permission decisions could not be carried over. `lineage.root: true` is a positive claim: a

missing `lineage` means unknown, never root. Workspace roots let every later path be written as `{root, path}`, and a root can be redacted on its own. `profiles` say what the producer promises to record (Section 7.9).

```
{
  "type": "session",
  "seq": 0,
  "id": "h",
  "formatVersion": "0.1",
  "agent": {
    "name": "com.example.agent",
    "version": "3.2.0"
  },
  "sessionId": "sess_5f0c9a12",
  "producer": {
    "name": "asr-convert",
    "version": "0.1.0",
    "method": "converter",
    "instance": "conv-01",
    "source": {
      "format": "example-agent-jsonl"
    },
    "lossy": ["permission"]
  },
  "authority": "local",
  "startedAt": "2026-10-26T14:02:11.204Z",
  "lineage": {
    "root": true,
    "scope": "primary"
  },
  "workspace": {
    "roots": [
      {
        "id": "r1",
        "path": "/home/dev/src/app",
        "vcs": {
          "type": "git",
          "remote": "https://github.com/acme/app",
          "head": "5d1c0e9...",
          "branch": "main",
          "dirty": false
        }
      }
    ]
  },
  "profiles": ["core", "coding", "lineage", "usage"],
  "contentLevel": "full"
}
```

A prompt, a model call and a reply. The user message separates who wrote it (`author: human`) from how it entered the session (`origin: prompt`; other values include `queued`, `steer`, `injected` and `retrieval`) and who it was for (`audience`). The `inference` record is one model call, with its own ID scope (here the provider's response ID). It is written only when the source identifies a model call, never synthesized from message boundaries. Its usage counts only that call, with `cached` and `uncached` tokens identified separately, and its cost includes the source of the figure. The assistant message points back to the call through `inferenceId`. Its reasoning was encrypted by the provider, so it is kept as an opaque blob, excluded from text indexing.

```
{
  "type": "message",
  "seq": 1,
  "id": "r1",
  "messageId": "msg_01",
  "ts": "2026-10-26T14:02:11.950Z",
  "role": "user",
  "author": "human",
  "origin": "prompt",
  "audience": "both",
  "content": [
    {
      "type": "text",
      "text": "Fix the failing parser test."
    }
  ]
},
{
  "type": "inference",
  "seq": 2,
  "id": "r2",
  "inferenceId": "resp_7a1",
  "purpose": "turn",
  "model": {
    "provider": "example",
    "id": "example-model-5",
    "requested": "default"
  },
  "usage": {
    "input": {
      "uncached": 1204,
      "cacheRead": 22976
    },
    "output": {
      "total": 412,
      "reasoning": 160
    }
  },
  "cost": {
    "amount": 0.0213,
    "currency": "USD",
    "source": "harness_estimated"
  },
  "latency": {
    "ttftMs": 640,
    "durationMs": 5120,
    "stopReason": "tool_use"
  }
},
{
  "type": "message",
  "seq": 3,
  "id": "r3",
  "messageId": "msg_02",
  "inferenceId": "resp_7a1",
  "role": "assistant",
  "author": "model",
  "audience": "both",
  "content": [
    {
      "type": "reasoning",
      "visibility": "encrypted",
      "opaque": {
        "blob": "sha256:9be1...",
        "size": 2210
      }
    },
    {
      "type": "text",
      "text": "Running the parser tests first."
    }
  ]
}
```

A tool call and its result. A `tool_call` means the model requested a call; the result records what happened. The call keeps its exact name, plus an optional display title and an ACP tool kind. The result's `content` is what reached the model. Its capture state says the content was truncated, how large the original was, and where the full output is kept: a content-addressed blob in the same bundle. `exec` records the command, the working directory and the exit code as structured data. `linkage: exact` says the pairing came from a native call ID. A call with no result stays dangling, and duplicate native call IDs are never paired by order.

```
{
  "type": "tool_call", "seq": 4, "id": "r4", "callId": "call_01", "callIdSource": "native",
  "inferenceId": "resp_7a1", "name": "shell", "kind": "execute", "title": "Run parser tests",
  "input": { "command": "cargo test -p parser" }
}
{
  "type": "tool_result", "seq": 5, "id": "r5", "callId": "call_01", "linkage": "exact",
  "status": "failed", "content": [ { "type": "text", "text": "test parse_empty ... FAILED ..." } ],
  "capture": { "content": { "state": "truncated", "originalBytes": 48213,
    "blob": { "blob": "sha256:41ad...", "size": 48213 } } },
  "exec": { "command": "cargo test -p parser", "cwd": { "root": "r1", "path": "." },
    "exitCode": 101 },
  "startedAt": "2026-10-26T14:02:17.410Z", "durationMs": 5210
}
```

A subagent. The agent delegates a search. The parent stream announces the child with a subagent record that names the child session and the call that created it. The child has its own stream, and its header records the lineage claim and the evidence for it. `inherits.mode: none` says none of the child's records were copied from the parent, so every record in the child is its own work. The session reference `com.example.agent:sess_5f0c9a12` follows the core grammar `<agent>:[<namespace>:]<sessionId>[#<recordRef>]`.

```
{
  "type": "tool_call", "seq": 8, "id": "r8", "callId": "call_02", "inferenceId": "resp_7a2",
  "name": "spawn_agent", "kind": "other",
  "input": { "role": "explorer", "task": "Find where the tokenizer handles empty input" }
}
{
  "type": "subagent", "seq": 9, "id": "r9", "childSession": "com.example.agent:sess_77b2e0",
  "callId": "call_02", "async": false
}
```

```
{
  "type": "session", "seq": 0, "id": "h", "formatVersion": "0.1",
  "agent": { "name": "com.example.agent", "version": "3.2.0" }, "sessionId": "sess_77b2e0",
  "lineage": { "spawnedBy": { "session": "com.example.agent:sess_5f0c9a12",
    "record": "call_02", "evidence": "native_field" },
    "inherits": { "mode": "none", "role": "explorer" },
    "scope": "subagent", ... }
}
```

A file change and a commit. The `file_change` names its cause (the edit call), its status and its precision. Each change follows ACP v2's shape and adds git blob object IDs for the file before and after the edit. The blob ID for the file after the edit is the hash git stores in a commit tree, so a tool can later match this edit to the commit that contains it. That match provides strong evidence for attribution but does not prove causation. The `vcs` record says the harness observed the commit being created (evidence: `observed`), rather than parsing it from output or seeing it mentioned. The `state` record closes the turn.

```
{
  "type": "file_change", "seq": 14, "id": "r14", "cause": { "callId": "call_03" },
  "status": "applied", "precision": "exact",
  "changes": [ { "op": "modify", "root": "r1", "path": "src/tokenizer.rs",
    "before": { "objectId": "8c3f2b1...", "objectIdType": "git-blob-sha1" },
    "after": { "objectId": "e41a907...", "objectIdType": "git-blob-sha1" } } ],
  "patch": { "format": "git_patch", "blob": { "blob": "sha256:c07d...", "size": 612 } }
}
{
  "type": "vcs", "seq": 21, "id": "r21", "action": "commit", "root": "r1",
  "commit": "a9e4d27...", "parents": [ "5d1c0e9..." ], "branch": "main",
  "cause": { "callId": "call_05", "evidence": "observed" }
}
{
  "type": "state", "seq": 22, "id": "r22", "state": "idle", "stopReason": "end_turn"
}
```

A compaction, much later. Compaction is its own record, never a message, so a summary is never indexed as something the user said. It names what it replaces (a range here, or an explicit ID set for harnesses that hide individual events), the token counts before and after, and whether the earlier history was kept. If the source destroyed it, `historyRetained` is `false` and a gap record follows.

```
{
  "type": "compaction",
  "seq": 120,
  "id": "r120",
  "compactionId": "cmp_1",
  "kind": "summary",
  "trigger": "auto",
  "summary": [
    {
      "type": "text",
      "text": "Earlier: fixed empty input ..."
    }
  ],
  "replaces": {
    "throughId": "r118",
    "tokens": {
      "before": 182000,
      "after": 21000
    }
  },
  "historyRetained": true
}
```

The commit that resulted can also point back to the session with a trailer, defined in the integration companion (Section 8):

```
Agent-Session: com.example.agent:sess_5f0c9a12
```

An assistant in a group chat. The same records describe a personal assistant. In this second fictional example, an assistant sits in a family group on a messaging app. The header declares the people in the chat as participants and the chat itself as a conversation. `conversationKey` is the source's own key for the chat, which stays the same when the assistant starts a new session for it. The `messaging` profile promises that, whenever the source records them, each inbound message names its sender and conversation and each reply the source sent has a `delivery` record.

```
{
  "type": "session",
  "seq": 0,
  "id": "h",
  "formatVersion": "0.1",
  "agent": {
    "name": "com.example.assistant",
    "version": "1.4.0",
    "sessionId": "win_4c1d",
    "conversationKey": "family-group",
    "scope": "primary",
    "participants": [
      {
        "id": "p1",
        "kind": "human",
        "role": "owner",
        "displayName": "Dana"
      },
      {
        "id": "p2",
        "kind": "human",
        "role": "member",
        "displayName": "Sam"
      }
    ],
    "conversations": [
      {
        "id": "c1",
        "channel": "whatsapp",
        "kind": "group"
      }
    ],
    "profiles": [
      "core",
      "messaging",
      "usage",
      ...
    ]
  }
}
```

Sam asks for a change. Changing the shared calendar needs the owner's approval, and the `permission` record names Dana as the person who approved it (`decider`) and says why approval was needed (`reason`). The reply is a `message`, but a message alone does not mean anyone received it, so the `delivery` record says it was sent. The tool result is omitted here.

```
{
  "type": "message", "seq": 40, "id": "r40", "messageId": "m_118", "role": "user",
  "author": "human", "origin": "prompt", "from": { "participant": "p2" },
  "conversation": "c1",
  "content": [ { "type": "text", "text": "Can you move Friday's dinner to 7?" } ]
}, {
  "type": "tool_call", "seq": 42, "id": "r42", "callId": "call_31", "inferenceId": "resp_c1",
  "name": "calendar_update", "input": { "event": "evt_88", "start": "19:00" }
}, {
  "type": "permission", "seq": 43, "id": "r43", "requestId": "apr_12",
  "subject": { "callId": "call_31" }, "reason": "Changes a shared calendar",
  "status": "resolved", "decision": "allow_once", "decidedBy": "user",
  "decider": { "participant": "p1" }
}, {
  "type": "message", "seq": 46, "id": "r46", "messageId": "m_119", "inferenceId": "resp_c2",
  "role": "assistant", "author": "model", "conversation": "c1", "phase": "final",
  "content": [ { "type": "text", "text": "Done: Friday's dinner is now at 7." } ]
}, {
  "type": "delivery", "seq": 47, "id": "r47", "message": "messageId=m_119",
  "conversation": "c1", "status": "sent", "platformMessageIds": [ "pm_5521" ]
}
```

Other assistant facts use the same pattern. A turn that no person started carries a `trigger` on its first record, such as `{ "kind": "schedule", "id": "job_brief", "runId": "fire_0412" }` for a job that runs each morning. A question the agent puts to a person is an `input_request` with its options and the answer; secret answers are never recorded. A note the agent saves about Dana is a `memory_change` that names the tool call that wrote it, so a user can trace a remembered fact back to the session that created it.

Streams travel in **bundles**: a directory or archive whose `bundle.json` manifest lists each session's key, path, digest, record count and status (present, or missing with a reason such as `deleted_by_retention`). Large content lives under `blobs/sha256/`. Original native files may sit under `native/`, but never in a shareable bundle by default.

7 Key design decisions

Each decision below lists what we chose, the main alternatives, and why.

7.1 Identity

Decision. A session key is the agent name, an optional namespace and the native `sessionId`, verbatim. Every record has a record ID; records that describe a logical entity also carry its native key (`messageId`, `callId`, `inferenceId` and so on). A revision gets a new record ID and keeps the entity key. Derived IDs are marked `idSource: "derived"`, and positional IDs from sources rewritten as whole snapshots are marked `valid` only within one capture. A public agent identity registry documents each harness's rules, such as how a subagent's stream is identified.

Alternatives. Consumer-minted UUIDs; content-addressed IDs, which make a file self-verifying; path-based identity.

Why. Citations must survive re-import, tool upgrades and moved files. IDs derived by a consumer drift when the derivation changes; our own implementation broke old citations exactly this way. Content hashes change whenever a redaction or correction changes bytes, and they

never match the ID a harness shows its users, such as the one passed to a resume command. The namespace exists because some native IDs are local counters or timestamps.

7.2 Immutable records and revisions

Decision. A record never changes once written. A correction, or a later change to a fact that was already final, is a new full record with `supersedes: <recordId>`. Converters combine streamed chunks and partial rows into one record and indicate that in `assembly`. When a source keeps a revision of a final fact, such as a changed tool status or masking applied to a completed result, converters should keep the revision as a `supersedes` chain. Combining those revisions into one record is a declared loss. Snapshot-only sources record what was seen and never manufacture past transitions.

Alternatives. Store live-protocol patches; keep every observed revision, chunks included; always keep only the final state.

Why. Patches make every reader implement a patch engine and make a record's meaning depend on everything after it. Keeping every chunk increases the number of records as token count grows and makes the session harder to read. Keeping only the final state destroys evidence, for example that the model saw an output before it was masked. Full replacements keep reading simple and auditing possible.

7.3 Order, branches and the context view

Decision. `seq` defines order. Messages, tool calls, tool results and compactions take part in the model's context. Every other record is kept only for auditing and is never the context predecessor of another record. A context record's predecessor is the previous context record, or the target of a `rewind`. `branchesFrom` appears only at a real branch point in a tree-stored source, and the specification publishes a normative rule for telling mechanical siblings (parallel tool calls, attachments, retries) from real branches. `session_info.heads` allows several current leaves. The context view, over the active path, applies rewinds and then compactions in `seq` order. Off-path records remain audit facts: commits made and money spent on an abandoned branch still happened.

Alternatives. An explicit predecessor pointer on every record; ordering by timestamp.

Why. Most sessions are linear, and most parent pointers in tree-shaped logs do not mark real branches, so requiring every reader to walk a tree for the rare real branch would add work for all of them. Timestamps are missing, coarse or out of order in too many sources. The context view is reconstructed from recorded evidence and does not prove what the exact request contained. A recorded exact request goes in `inference.context`, which can name an earlier request as its prefix (`prefixFrom`) so that a long session does not repeat its whole history in every model call. An explicit predecessor pointer on every record remains a reasonable alternative and an open question (Section 11).

7.4 Lineage as independent facets

Decision. The header's lineage is a set of optional facets: `spawnedBy` (the delegation that created this session), `forkedFrom` (a divergence point, by record or ordinal), `continues` (with a reason such as compaction, reset or resume), `inherits` (whether ancestor records are physically present: copied, referenced, rendered as a summary, none or unknown), `executedBy` (the sessions of another agent that actually ran this session's turns) and `relatedUnknown` (a native parent pointer of unknown meaning). Each facet carries its own evidence. The distinction between `primary` and `subagent` is separate from these facets. These claims are recorded in the session they describe (the child, for a spawn or fork), so a missing parent never invalidates the child. A copied record carries `copiedFrom` with a proof kind. Copies stay searchable, but consumers never count their usage, cost, edits or commits as the child's work.

Alternatives. One relation enum (delegated, forked, resumed and so on); a general graph of parent edges.

Why. Real sessions combine relations. A subagent started with a copy of its parent's context is both a delegation and a fork, and the history it inherits can come from a different session than the one that delegated to it. A single enum forces converters to pick one and misstate the rest. Each facet is asserted only when the source proves it. A missing `copiedFrom` proves a record is original only when `inherits.mode` is `none`.

`executedBy` covers a case that none of the other relations describe. T3 Code runs Claude Code and Codex sessions, and OpenClaw can run its sessions through Codex [1]. Both sides then keep a transcript of the same work. This is not a fork or a spawn, and merging the two streams would lose who wrote each prompt, which only the outer app knows: the backing transcript shows the app's injected prompts as ordinary user prompts. So each side keeps its own stream. The app's stream names the backing sessions in `executedBy`, optionally turn by turn, and its records that copy the backing session carry `copiedFrom`, so model calls, cost and edits are counted once.

7.5 Inference and usage

Decision. Per-call usage is recorded on `inference` records and always counts only that call. The uncached input count is used only when the source's cache rule is unambiguous; otherwise the record gives `input.total` and `includesCache: true | false | null`. Reasoning tokens are a subset of output tokens. Cost carries a currency and a source (`provider_reported`, `harness_estimated` or `converter_estimated`), and absent cost means unknown, never zero. Native aggregates go in `usage_summary`, which states scope (turn, session or tree), whether values are cumulative, whether children are included, and coverage. Consumers use either an aggregate that covers the relevant records or its components, never both, and never add parent and child totals of unknown scope.

Alternatives. One normalized token schema with a mandatory cache breakdown; per-call usage reconstructed from aggregates; dropping tree-wide totals.

Why. Existing formats disagree on whether input counts include cache: ATIF counts cached tokens inside prompt tokens, Entire excludes them, and UAS carries a flag. Harnesses also repeat usage across split lines or report cumulative counters, and mislabeling these is the usual cause of large overcounts. A mandatory breakdown would make converters guess, and dropping native tree totals would force them to mislabel or discard real data. Fresh charges for reprocessing inherited context are real new costs, distinct from copied historical accounting.

7.6 File changes as evidence, not proof

Decision. A `file_change` records its cause (a call, an observed human edit or a restore), its status and ACP v2's `changes[]`, with paths as `{root, path}` in POSIX form plus an optional native spelling. Before and after states may carry object IDs (`git-blob-sha1`, `git-blob-sha256` or `sha256`); they are recommended, never required. `precision` is required: `exact`, `hunks`, `snippet` or `unknown`. A producer must never hash the current file and label it as the file's state after an earlier edit unless that state was observed at the time of the edit. `vcs` records commits, pushes, merges and pull requests, each with its evidence: `observed`, `parsed from output`, or `reported`.

Alternatives. Require full before and after images; require object IDs; promise exact attribution without heuristics.

Why. Many harnesses only have patch hunks or snippets, so requiring full file contents before and after an edit would push converters to fabricate them. Git blob IDs are cheap to compute and match commit trees directly. A matching blob is strong evidence, but the same content can arise independently, so ASR stops at evidence and leaves attribution claims to tools. Likewise, a session that mentions a commit did not necessarily produce it.

7.7 Capture states, gaps and tombstones

Decision. Every content-bearing field may carry a capture state: `complete`, `truncated`, `masked`, `summarized`, `external`, `omitted`, `redacted` or `absent`, with optional original size, digest, blob reference and reason. Completeness is relative to what the source captured, and a bounded copy is never labeled complete. A `gap` record marks missing history, with reasons from `not_retained` and `deleted_by_retention` to `encrypted`, `unparseable` and `conversion_loss`. A `tombstone` means upstream deletion or withdrawal only, never a correction. There are no file-wide trust or fidelity labels.

Alternatives. A file-wide fidelity or trust label; silent omission; tombstones that double as corrections.

Why. A label like “fidelity: full” cannot be checked, and in our experience it is the first thing to go stale. Per-field states are concrete and testable. Provenance comes from how the stream was produced (natively, by a converter, decoder, recorder or observer), from capture states and from per-facet evidence. Missing history is not evidence that something never happened, as ACP's subagent design also states [38].

7.8 Privacy and security

Decision. `contentLevel` (`full`, `redacted` or `metadata`) describes disclosure, not safety, and individual records may carry stricter `redactions[]`. Workspace roots can be redacted on their own. Redaction produces a new stream that records its source digest and policy. Redaction must cover blobs, accompanying native files, unknown extensions, identifiers and embedded paths. When redaction conflicts with preservation, redaction takes precedence. Conformance does not make a file safe to share, so exporters that prepare files for sharing must apply a redaction policy and record which one. Readers must extract archives safely, limit the resources used for parsing, never fetch URIs or run discovery commands automatically, display transcript content without executing it, and ask before publishing session references into git history.

Alternatives. Leave privacy to each tool; “safe to share” labels; automatic secret scrubbing as part of conformance.

Why. Transcripts contain secrets, private paths, customer names and internal reasoning. HAR, the browser’s archival format for HTTP sessions, is the cautionary example: HAR files routinely capture cookies and tokens, and shared HAR files have been used to hijack sessions. Git AI removed transcripts from git notes so they never leak through pushed refs [22]. Labels that imply safety mislead, which is why we retired them in our own implementation.

7.9 Profiles: what a producer promises to record

Decision. There are five profiles: `core`, `coding`, `lineage`, `usage` and `messaging`. Claiming a profile means that whenever this kind of event happens, the producer emits the record or an explicit capture state. It does not mean that such records are present: a session with no commits can still claim `coding`. `messaging` promises, whenever the source records them, the sender and conversation of each inbound message and a `delivery` for each reply the source sent. Memory writes, approvals, questions and triggers need no profile of their own: a producer that records them follows their rules. Because streams are written live, a `declaration` record can later add profiles, extensions and declared losses, or tighten the content level for later records. Declarations only ever add or tighten.

Alternatives. Presence claims; numbered conformance levels; one large required schema; separate profiles for assistants, memory and actions in the world.

Why. These promises answer a question every consumer has: whether a missing commit record means “no commit happened” or “commits are not captured”. And a live writer cannot know at line 1 what the session will contain. Coding agents and assistants both write memory, ask for approval and start turns without a prompt, so those facts need no separate profile. Only group chats and reply delivery are specific enough to need their own promise.

7.10 Extensibility and versioning

Decision. `formatVersion` is `MAJOR.MINOR`; minor versions only add optional fields, types and enum values, and this draft is `0.1`. The specification document is versioned separately. There is one schema, with no experimental tier (Section 7.14). Vendor fields go under `_meta` with reverse-

DNS keys, vendor record types are reverse-DNS names with / (such as `dev.zed/agent_switched`), and vendor enum values use ACP's `_` prefix. Producers must not invent root fields. Readers declare the major versions they support and fail explicitly on other major versions; storage tools keep streams with unsupported major versions as uninterpreted bytes. Deprecation runs Active, Deprecated (at least twelve months), Removed.

Alternatives. A closed schema with one extension bag; date-based versions; every reader supporting every major forever.

Why. These are the lessons from Section 4, plus one from UAS: keep the document version separate from the format version, so editorial changes cannot be mistaken for format changes.

7.11 ACP alignment

Decision. ASR reuses ACP v2's content blocks (pinned to a stated revision, with deviations listed), tool kinds and statuses, the `diff changes[]` shape, permission options, state values and the `_` enum convention. A reasoning-only message keeps its native ID, matching ACP's agent thoughts. The ACP v2 recorder mapping is the one normative bridge in the core specification (Section 8). Replay promises to preserve meaning, with declared losses. It promises identical bytes only when an optional capture of the original protocol traffic is retained. Where ACP has no field (timestamps, lineage, git, the model behind each entry, cost scope), ASR adds one.

Alternatives. Archive raw ACP update streams; design a content model independent of ACP.

Why. Raw update streams lack timestamps, lineage and commits, and must be replayed through patch rules to be read. An independent content model would split the ecosystem. We would rather propose to ACP the features a history record needs, such as fork at a message and per-turn usage, and keep the two specifications converging.

7.12 Senders, messages between agents, and triggers

Decision. One actor reference, `{participant | session | address | device}`, says who sent a message, who resolved an approval, who answered a question and what fired a trigger. A group chat declares its participants and conversations in the header; each message can name its `from` and its `conversation`, and a separate `delivery` record says whether a reply was queued, sent, suppressed or failed. A message from another agent or session is an ordinary message with `origin: inter_session` and a `from` that names the sending session; the sending side is the sender's own tool call. A `trigger` on the first record of a turn says what started it: a schedule, a heartbeat, an event or webhook, a finished task, another agent, an automatic continuation, recovery or a manual run, with the job and run IDs the source keeps.

Alternatives. A separate `session_message` record type for messages between sessions (an earlier version of this draft); a general `effect` record for every action in the world; new `origin` values for each kind of trigger.

Why. The receiving agent takes a message from another session into its context like any other input, so a separate audit-only record type would leave part of the context outside the context view. The four tools send these messages in different ways (Claude Code to named teammates, Codex between agent threads, T3 Code between threads, OpenClaw between sessions), but in every case the receiver gets a message. A reply and its delivery are separate facts: a reply can be generated and never sent, or sent in several parts. A general effect record would have to cover payments, bookings and calls that none of the four tools records yet. `origin` says how text entered the context and `trigger` says why the turn ran; a scheduled prompt needs both.

7.13 Approvals, questions and memory

Decision. A `permission` records a request for approval with its `status` (pending, held, expired or resolved), its decision (including `allow_session`, for the rest of the session), who decided (`user`, `policy`, `hook`, `auto_mode`, `agent`, or `classifier` for a model reviewer), the actor who resolved it, separate reviews by models or agents, and the reason approval was needed. A decision made by a rule without showing a prompt is recorded in compact form on the tool result instead. A question the agent asks a person is an `input_request`: the questions, their options, whether the agent waited, and the answers. Secret answers are never recorded, in any form. A write to memory is a `memory_change` that names the store, the operation, the target and its cause, a tool call or an actor. The agent registry says, for each agent, which file writes are memory writes.

Alternatives. Questions as tool calls only; one `permission` record for every call; memory writes as ordinary file changes.

Why. In most harnesses a question to the user is a tool call with a harness-specific name. Consumers need to find questions, delegation and plans without knowing each tool's name, so they find them by record type, not by tool kind. Claude Code makes almost all of its permission decisions automatically, without a prompt, so one full record per call would add many records that only repeat an automatic decision. Approvals now come from people on other devices, from parent agents and from model reviewers, and an audit needs to tell them apart. A file change alone does not say that the file is memory, and some memory is not a file.

7.14 What enters the standard

Decision. Version 0.1 is fitted to four tools, in this order of priority: Claude Code, T3 Code, Codex and OpenClaw. A field or record type is standard when at least two of them record it, or when it is the only way to keep a concept ASR already has; each such exception has a written justification in the specification. Every other fact these tools record gets a documented destination in the tool's registry entry: a `_meta` key on the nearest record (preferred) or a vendor record type, citing the native name. Nothing recorded is dropped, although redaction and the exclusion of secrets still apply. There is no experimental tier. A feature that is not ready is left out of the schema and listed as future work.

Alternatives. An experimental tier beside the standard schema, with a feature promoted once two independent producers emit it (an earlier version of this draft); admitting anything one tool records; designing first for every assistant product on the market.

Why. An experimental tier made it easy to add features that no tool records yet, while the features it held back were ones the most important tools already record: in our reading, Claude Code uses every record type that the earlier draft marked experimental. Requiring two of four tools keeps one product's habits out of the core. Registry destinations mean a converter never has to drop a fact to stay conformant, and they show which facts are candidates for the next version. Fitting real tools first gives the corpus and the converters something concrete to test.

8 Bridges and companions

ASR is meant to be the primary record that other formats are generated from, so harnesses need not capture the same session several times.

Normative: the ACP v2 recorder. A client or proxy that implements ACP v2 can record a live session in ASR. The mapping is part of the core specification, pinned to an ACP schema revision, with every field marked as carried, transformed or lost. It covers permission requests and questions to the user, and it uses the `terminal` record, now part of the format, for long-running processes.

Companion documents, versioned separately so changes to them do not affect the core:

- **ATIF export:** one trajectory per root-to-leaf path. Copied records become `is_copied_context` steps, and subagents become `subagent_trajectories`.
- **OpenTelemetry GenAI export:** `invoke_agent` spans for sessions and subagents, `chat` spans for inference records and `execute_tool` spans for tool calls, with the session as `gen_ai.conversation.id`, emitted only where timing exists.
- **GitHub UAS import and export**, mapping the core event types one to one.
- **Entire import**, accepting a native transcript and Entire's normalized transcript as a pair.
- **Agent Trace and Git AI derivation**, generating attribution records from file changes and commit records, including Git AI's join key.

Integration companion. Commit linkage is kept out of the core format's conformance:

- **Trailer.** `Agent-Session: <session reference>`, one per session, with no URL appended. A locator, if wanted, goes in a separate `Agent-Session-Url` trailer.
- **Aliases.** `Agent-Logs-Url` [39], `Copilot-Session` and `Claude-Session` are treated as direct session aliases, each with a documented ID scope (`Copilot-Session`, for example, carries a cloud session ID rather than the local resume ID [40]). `Entire-Checkpoint` resolves to a session through a checkpoint. `Co-authored-by`, `Assisted-by` [41] and `Made-with` [42] name actors or tools, not sessions, so they are not aliases.

- **Notes.** `refs/notes/agent-sessions` carries references and digests only. Transcripts never go into pushed git refs by default.
- **Export command and discovery file.** `<agent> sessions export --format agent-session`, plus a small JSON file under `$XDG_CONFIG_HOME/agent-sessions/sources/` declaring the export command, native paths, format versions and whether a deletion hook exists.

9 Adoption

9.1 For harnesses

There are three adoption levels, starting with one that requires no vendor involvement.

- **L0: third-party converters read native logs.** This works today. A converter declares its losses, and the agent identity registry records how it reads each harness.
- **L1: the harness ships an export command and a discovery file.** The export command can initially call an existing converter. The harness team knows what its logs mean and takes responsibility for the mapping.
- **L2: the harness's native log is a conformant stream:** the ASR envelope, with vendor-prefixed record types for anything the harness does not want to normalize. It is not a second log written alongside the first.

Harnesses that already write append-only JSONL with native IDs can adopt L2 with limited changes. The common record fields are `type`, `seq`, `id` and an optional `ts`; other record types can start as vendor types and move into core types over time. For the four tools ASR is fitted to, the registry entry says where each recorded fact goes, including the facts that become vendor `_meta` keys.

The same levels apply to apps that run other agents and to assistants. An app such as T3 Code gets a stream of its own, converted from its own records and linked to the converted streams of the agents it ran (Section 7.4).

9.2 For tool builders

To interpret records correctly across harnesses, a reader follows these rules:

- Treat missing values as unknown. A missing `trigger` does not mean a person started the turn.
- Order by `seq`, not timestamps.
- Never count copied records as the child's work, and never count an app's copies of the sessions it ran as new work.
- Use either a usage aggregate that covers the relevant records or its components, never both.
- Pair tool results only by declared linkage.
- Find questions, delegation and plans by record type, not by tool name or kind.

- Treat a reply as delivered only when a `delivery` record says so.
- Display transcript content without executing it.
- Keep unknown fields, types and enum values when storing or rewriting a stream.

9.3 What we ask of vendors

Each request below requires little effort, and several require no code.

1. **Check our reading of your logs.** Review the identity registry entry and converter notes for your harness, and tell us where we are wrong.
2. **Document your ID scopes and your fork, resume and copy behavior.** A paragraph in your docs saves every converter author from guessing.
3. **Mark copies.** Stamp copied records, or declare an inherited prefix, as some harnesses already do.
4. **Record outcomes when they happen.** Exit codes, the commit a `git commit` created, the pull request a command opened, each tool's working directory, and whether a reply was delivered, as structured fields rather than output text.
5. **Record who and why.** Who sent each message in a group chat, what started each turn, who or what decided each approval, and which tool call or person wrote each memory, as structured fields rather than text inside the prompt.
6. **Make deletion visible.** When retention or truncation removes history, leave a marker, or offer a deletion hook through the discovery file.
7. **Ship an L1 export** once the format reaches Preview.
8. **Contribute sanitized fixtures** from real sessions to the golden corpus.
9. **Nominate a reviewer or maintainer.** The format should not be owned by any one company, including ours.

10 Governance and roadmap

ASR is an independent project, published at agentsessionrecord.org [43], and uses a proposal process modeled on ACP's RFDs (Requests for Dialog) [44]. A change starts as a Draft, reaches Preview once it has a working implementation, and is Completed once it has conformance tests. A new field or record type also has to meet the admission rule in Section 7.14. We want maintainers from more than one organization, with ACP's maintainers as a formal liaison so the two specifications converge rather than drift [45]. After 1.0, once two independent implementations and the golden corpus exist, we intend to propose ASR to the Linux Foundation's Agentic AI Foundation, which already hosts MCP and AGENTS.md [46].

Before Preview. Five things must be done before the format is declared Preview:

1. Write the normative algorithms (the mechanical-sibling rule, the active path and the context view), each with fixtures.

2. Build the golden corpus: sanitized real sessions from at least ten harnesses, plus adversarial fixtures for the active path and context view, revision chains, terminals, declarations, copies, every compaction kind, incomplete final lines and unknown extensions.
3. Ship the validator, the JSON Schemas and the agent identity registry.
4. Publish the ACP v2 recorder support matrix.
5. Show two independent converters for the same five harnesses agreeing on the corpus.

Before 1.0. Every requirement is numbered and mapped to a validator check, with separate checks for syntax, semantics and unsupported features. Conformance classes cover producers, converters (which must declare losses), recorders, readers and exporters. Every example in the specification is validated in CI, and 1.0 requires the fixtures plus two independent implementations.

Not in 0.1 (future work). These are left out of the schema: principal, tenant and on-behalf-of identity; payments; a person taking over the agent's browser; tamper-evidence seals; guardrail verdicts; export coverage, retention policies and legal holds; provenance verified by the provider as distinct from content the client asserts; agent-to-agent tasks across vendors (A2A); live voice sessions; streams of a memory store on its own; records of resource access on their own; patches of uncommitted changes on workspace roots (`dirtyPatch`); and general effect and task records. Each can join later under the same admission rule as everything else.

11 Call for review

This is a draft, and it will change. We are asking for review now, while the core is still easy to fix. The most useful feedback is specific:

- **Harness teams:** Is our reading of your native format correct, especially identity, fork and copy behavior, compaction and retention? What would an L1 export cost you?
- **Assistant and app teams:** Do participants, conversations, triggers and deliveries describe your sessions the way your product means them? Which facts end up only in vendor keys, and should any of them be standard?
- **ACP maintainers and implementers:** Does the recorder mapping work as intended? Which features needed for historical records, such as timestamps, forking at a message and per-turn usage, belong in ACP?
- **Evaluation and training teams:** Does the ATIF export keep what you need, and is the copy model strong enough to keep copied context out of training data?
- **Observability and OTel GenAI contributors:** Is the span mapping right, and should the content vocabularies converge further?
- **Attribution and review tools:** Are file-change object IDs and commit records enough to build on, without overclaiming?

Several design questions are open, and we would especially value arguments on them:

1. **Branch representation.** A linear default with `branchesFrom` at real branch points, or an explicit context predecessor on every record?
2. **Vendor enum syntax.** ACP's `_` prefix, or reverse-DNS names like record types?
3. **Trailer value.** May the `Agent-Session` value carry a URL, or should locators stay in a separate trailer?
4. **Usage.** Are the cache-inclusion flag and the tree-scoped aggregate rules sufficient for your accounting?
5. **Admission.** Is “at least two of four priority tools” the right rule for a standard feature, and which tools should be added to the priority list next?
6. **Future work.** Which item on the “Not in 0.1” list (Section 10) do your users need first?

How to engage. Write to feedback@agentsessionrecord.org, or propose a change as an RFD. The specification, the companion specifications and this paper are at <https://agentsessionrecord.org>. If you maintain a harness, the most valuable single contribution is a sanitized fixture and a correction to our notes on your format.

References

1. OpenClaw. *OpenClaw* (source). <https://github.com/openclaw/openclaw>
2. Nous Research. *Hermes Agent* (source). <https://github.com/NousResearch/hermes-agent>
3. Anthropic. *Claude Code changelog*. <https://github.com/anthropics/claude-code/blob/main/CHANGELOG.md>
4. Google. *Gemini CLI: session management*. <https://github.com/google-gemini/gemini-cli/blob/main/docs/cli/session-management.md>
5. Zed Industries. *Zed agent thread database* (source). <https://github.com/zed-industries/zed/blob/main/crates/agent/src/db.rs>
6. Cline. *Cline* (source). <https://github.com/cline/cline>
7. Block. *goose session manager* (source). https://github.com/block/goose/blob/main/crates/goose/src/session/session_manager.rs
8. Google. *Gemini CLI chat recording service* (source). <https://github.com/google-gemini/gemini-cli/blob/main/packages/core/src/services/chatRecordingService.ts>
9. OpenAI. *Codex CLI* (source). <https://github.com/openai/codex>
10. Pi. *pi-mono* (source). <https://github.com/badlogic/pi-mono>
11. GitHub. *GitHub Copilot CLI changelog*. <https://github.com/github/copilot-cli/blob/main/changelog.md>
12. T3 Tools. *T3 Code* (source). <https://github.com/pingdotgg/t3code>
13. ctx (source). <https://github.com/ctxrs/ctx>
14. Agent Client Protocol. *Documentation and specification*. <https://agentclientprotocol.com>
15. Agent Client Protocol. *ACP v2 draft* (announcement), July 2026. <https://agentclientprotocol.com/announcements/acp-v2-draft>
16. Harbor. *RFC 0001: Agent Trajectory Interchange Format (ATIF)*. <https://github.com/harbor-framework/harbor/blob/main/rfcs/0001-trajectory-format.md>
17. Harbor. *ATIF documentation*. <https://docs.harborframework.com/core-concepts/agents/atif>
18. GitHub. *Unified Agent Session Specification* (GitHub Agentic Workflows). <https://github.github.com/gh-aw/specs/unified-agent-session-specification/>
19. OpenTelemetry. *Semantic conventions for generative AI*. <https://github.com/open-telemetry/semantic-conventions-genai>
20. *Agent Trace*, version 0.1.0 (RFC), January 2026. <https://agent-trace.dev/>
21. Git AI. *Git AI Standard v3.0.0*. https://github.com/git-ai-project/git-ai/blob/main/specs/git_ai_standard_v3.0.0.md
22. Git AI. *Pull request #1185*, April 2026. <https://github.com/git-ai-project/git-ai/pull/1185>
23. Entire. *Sessions and checkpoints* (architecture documentation). <https://github.com/entireio/cli/blob/main/docs/architecture/sessions-and-checkpoints.md>
24. Entire. *Entire Transcript Format*. <https://github.com/entireio/cli/blob/main/transcript/README.md>
25. OASIS. *Static Analysis Results Interchange Format (SARIF) Version 2.1.0*, OASIS Standard, March 2020. <https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html>
26. GitHub Docs. *SARIF support for code scanning*. <https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning>
27. Sourcegraph. *Announcing SCIP*, 2022. <https://sourcegraph.com/blog/announcing-scip>

28. Model Context Protocol. *SEP-2567: Sessionless MCP*. <https://modelcontextprotocol.io/seps/2567-sessionless-mcp>
29. Model Context Protocol. *The 2026-07-28 specification release, July 2026*. <https://blog.modelcontextprotocol.io/posts/2026-07-28/>
30. Agent Client Protocol. *RFD: Message ID*. <https://agentclientprotocol.com/rfds/message-id>
31. Agent Client Protocol. *RFD (v2): Enum variant extension*. <https://agentclientprotocol.com/rfds/v2/enum-variant-extension>
32. Model Context Protocol. *Specification 2025-06-18: key changes*. <https://modelcontextprotocol.io/specification/2025-06-18/changelog>
33. Model Context Protocol. *SEP-2663: Tasks extension*. <https://modelcontextprotocol.io/seps/2663-tasks-extension>
34. Model Context Protocol. *SEP-2596: Specification feature lifecycle and deprecation*. <https://modelcontextprotocol.io/seps/2596-spec-feature-lifecycle-and-deprecation>
35. Agent Client Protocol. *RFD (v2): Prompt lifecycle*. <https://agentclientprotocol.com/rfds/v2/prompt>
36. Agent Client Protocol. *RFD (v2): Diff file states*. <https://agentclientprotocol.com/rfds/v2/diff-file-states>
37. Model Context Protocol. *SEP-2484: Conformance tests required for Final SEPs*. <https://modelcontextprotocol.io/seps/2484-conformance-tests-required-for-final-seps>
38. Agent Client Protocol. *RFD: Subagents*. <https://agentclientprotocol.com/rfds/subagents>
39. GitHub Changelog. *Trace any Copilot coding agent commit to its session logs*, March 20, 2026. <https://github.blog/changelog/2026-03-20-trace-any-copilot-coding-agent-commit-to-its-session-logs/>
40. GitHub. *copilot-cli issue #4130*. <https://github.com/github/copilot-cli/issues/4130>
41. The Linux kernel documentation. *AI coding assistants*. <https://www.kernel.org/doc/html/latest/process/coding-assistants.html>
42. Cursor. *Git integration* (help center). <https://cursor.com/help/integrations/git>
43. *Agent Session Record*. <https://agentsessionrecord.org>
44. Agent Client Protocol. *About RFDs*. <https://agentclientprotocol.com/rfds/about>
45. Agent Client Protocol. *Governance*. <https://agentclientprotocol.com/community/governance>
46. Linux Foundation. *Linux Foundation announces the formation of the Agentic AI Foundation*, December 2025. <https://www.linuxfoundation.org/press/linux-foundation-announces-the-formation-of-the-agentic-ai-foundation>