ChatGPT

Claude Code Conversation Logs (`.jsonl`) – Format and Usage

Storage and Format of Project Chat Logs

Claude Code records each coding session's conversation as newline-delimited JSON (`.jsonl`) files on your local machine [1] [2]. These are stored under a hidden `.claude/projects` directory, organized by project folder (often an encoded form of your working directory) and session ID. For example, a transcript might reside at:

```
~/.claude/projects/<project-id>/<session-id>.jsonl
```

Each line in a transcript file is one JSON object representing a single turn or event in the dialogue [3]. This JSONL format makes it easy for Claude to append new events sequentially as the conversation progresses [4]. (It is *append-only*; the file grows with each new message, rather than being re-written.) Because the raw `.jsonl` logs are machine-oriented (one minified JSON per line with no pretty formatting), several community tools have emerged to view or convert them for humans [5].

JSONL Entry Schema and Fields

Each JSON object (line) contains structured fields describing either a user message, an assistant reply, or a tool interaction. Key fields in a typical entry include:

- `role` – The role of the speaker or actor for that entry (e.g. `"user"`, `"assistant"`, or `"tool"`). This is nested under a `message` field in newer Claude versions (see below).
- `content` – The text content of the message. For user entries this is the prompt text; for assistant entries this is the assistant's reply (if the assistant's reply is text). If an assistant entry instead initiates a tool use, `content` may be `null` (with tool details provided in a different field).
- `timestamp` – A timestamp of when the message was created, in ISO8601 format (e.g. `"2024-08-02T11:05:20Z"`). This helps order events chronologically [6].
- `tool_calls` / `tool_call_id` – Present only for tool-related events. When the assistant invokes a tool, the assistant's JSON line will include a `tool_calls` array describing the call (with fields like an `id` for the tool invocation and the tool type/command) [7]. A subsequent line with `role: "tool"` contains the actual output/result of that tool call, linked by the matching `tool_call_id` [7]. (See **Roles and Ordering** below.)
- `uuid` – A unique identifier for the message/event. Each entry has a `uuid` (usually a UUID string) to uniquely tag it [8].
- `parentUuid` – References the `uuid` of a preceding message if this entry is a direct reply or follow-up. This helps maintain the thread structure. For the first user turn in a session,

`parentUuid` is often `null` (no parent) [8]. Assistant responses typically reference the user message's UUID as their parent.

- `type` – A high-level classification of the entry's source, often mirroring the role. For example, `type: "user"` for user messages and `type: "assistant"` for assistant replies [8]. (Internally, Claude may also use types like `"tool_result"` for tool outputs, etc., though the `role` field is the clearer indicator of that in the logs.)
- `message` – In recent versions, the actual message content is nested in a sub-object. For example, an entry might have `"message": { "role": "assistant", "content": [...], "id": "...", "model": "claude-opus-4-20250514", ... }`. This sub-object can include:
- `role` (again, indicating user/assistant/tool),
- `content` (which could be a string or a structured array of segments for assistant replies [9]),
- `id` (another identifier for the message, like `"msg_..."`),
- `model` (the model name used for the assistant's reply, e.g. which Claude version) [9],
- `stop_reason`, `stop_sequence` – info on why the assistant stopped,
- `usage` – token usage counts for that reply (input tokens, output tokens, etc.) [10].

In older logs (or simplified views), the role and content might appear top-level for brevity [6], but fundamentally the schema captures similar information.

- **Other metadata** – Claude logs include additional context fields:
- `userType` – e.g. `"external"` for a human user message (versus an internal system message, if any) [11]. In nearly all normal cases, user prompts and assistant replies have `userType: "external"` (since they originate from the user or Claude externally).
- `isSidechain` – a boolean indicating if the message is part of a sidechain (a parallel thread or sub-conversation). This is usually `false` for the main conversation flow [11]. If Claude spawns sub-agents or background threads, those might be marked as sidechains.
- `cwd` – the current working directory (project path) at the time of the message [8]. This ties the conversation to a project context (helpful if you move between directories).
- `sessionId` – the unique ID of the session (usually reflected in the filename). All entries in a given `.jsonl` file share the same sessionId [8].
- `version` – the Claude Code version or build number in use (e.g. `"1.0.58"`) [8].
- `gitBranch` – the current Git branch name, if the project is in a git repo (empty if not applicable) [12].

Example: The first line of a sample transcript (a user prompt asking Claude to add something to a README) might look like:

```
{
  "parentUuid": null,
  "isSidechain": false,
  "userType": "external",
  "cwd": "/Users/schacon/projects/testing",
  "sessionId": "eb5b0174-0555-4601-804e-672d68069c89",
  "version": "1.0.58",
  "gitBranch": "",
  "type": "user",
  "message": { "role": "user", "content": "add rick to the readme" },
  "uuid": "b4575262-5ecc-4188-abe3-82c4eef91120",
```

```
  "timestamp": "2025-07-23T15:00:55.015Z"
}
```

This illustrates many of the fields above [13] [14]. The `message.role` and `type` are both `"user"`, `content` holds the prompt text, and metadata like `cwd`, `sessionId`, etc., provide context. An assistant's reply entry would be similar but with `role: "assistant"` and additional fields inside `message` (like the model and token usage) [9] [10].

Roles, Tool Calls, and Message Order

Claude's chat history is structured as an alternating sequence of user and assistant turns, with special entries inserted for tool usage. **User and assistant messages strictly alternate** in a valid conversation log – you won't see two user messages or two assistant replies back-to-back without something in between. The system enforces this ordering; in fact, if an assistant action (like a tool use) is not properly followed by the expected result entry, Claude will error about a missing block [15].

When the assistant needs to use a tool (e.g. run a shell command, search, edit a file, etc.), the transcript will reflect this in multiple entries:

1. **Assistant tool-call entry:** An assistant message indicating the tool invocation. In older/simplified form, this might appear as an assistant entry with `"content": null` and a `tool_calls` field describing the call [7]. (For example: an entry where `role: "assistant"` and `tool_calls` includes `{"id": "tc_1", "type": "bash", "bash": "cargo test"}` to signify Claude decided to run `cargo test` in a Bash tool [7].) In newer structured logs, the assistant's `message.content` might contain a special marker or be an empty content with a separate `type` indicating a tool use – but conceptually it's the same: the assistant turn is initiating a tool.
2. **Tool result entry:** Immediately after, there will be a separate JSON line with `role`: `"tool"` (or an equivalent designation) that captures the output from the tool call [16]. This entry is linked to the above by a `tool_call_id` (e.g. `"tool_call_id": "tc_1"`, matching the id from the assistant's tool call) [7]. The `content` of this entry contains the tool's stdout/stderr or result text. In our example, the tool entry might have `role: "tool"` and content like `"test result: FAILED. 1 passed; 1 failed"` [16].
3. **Assistant continuation:** After logging the tool's output, the assistant may then respond based on that result. The next entry would be another `role: "assistant"` message with its own content (e.g. an explanation or next step), continuing the conversation flow.

Crucially, Claude's formatting rules require that any assistant `tool_use` is followed by a matching `tool_result` **before** the next normal message [15]. In other words, the assistant cannot skip logging a tool's output – the JSONL must include the tool result entry(s) in sequence. This alternating pattern (user → assistant → *tool output* → assistant → ...) preserves the exact order of events.

For example, a raw snippet from a Claude Code transcript might look like this [7]:

```
{"role":"user","content":"Fix the failing
test","timestamp":"2024-08-02T11:05:20Z"}
```

```
{"role":"assistant","content":null,"tool_calls":
[{"id":"tc_1","type":"bash","bash":"cargo test"}]}
{"role":"tool","tool_call_id":"tc_1","content":"test result: FAILED. 1 passed; 1
failed"}
{"role":"assistant","content":"The tests are failing for one case. I will review
the failure output..."}
```

Here we see a user request, then an assistant entry calling a Bash tool (`cargo test`), then the tool's output, and finally the assistant's follow-up textual answer. In the actual `.jsonl` file, each of those is one line as shown. This illustrates how Claude interleaves tool interactions into the chat history. (Note that in structured logs, the assistant's tool call entry would also include all the metadata fields discussed above, not just `role` and `content` – we've shown a simplified view for clarity.)

Notably, there is **no separate "system" role line** in these project transcripts for Claude's internal system prompts. Any project-specific context (like instructions in `CLAUDE.md` memory files) isn't represented as a distinct message in the JSONL log; instead, such context is injected into Claude's prompt behind the scenes. The JSONL focuses on the user/assistant dialogue and any tool actions.

Context Metadata and Project Association

The JSONL transcript format includes fields that help Claude manage context across sessions and projects:

- **Project association:** The fact that transcripts live under a project-specific directory is itself a way of scoping conversations to a project. The `cwd` field in each entry records the working directory [12], so Claude knows which project it's operating in. Each project folder under `~/.claude/projects` will contain all session logs for that project [2]. For example, if you have a project called "MyApp", all Claude Code sessions run in that project's directory will be logged in `~/.claude/projects/<MyApp-encoded>/...jsonl`. This separation means Claude can keep different project conversations (and their file contexts) isolated.
- **Session identifiers:** Every conversation session has a `sessionId` (often used as the filename). This ID ties all the messages in that file together [12]. It's also passed to hooks and used by the CLI to resume sessions. For instance, Claude's hooks API provides the `session_id` and full `transcript_path` to any Stop-hook, so you know which session just ended [17].
- `parentUuid` **and threading:** As noted, each message can point to a `parentUuid`. In normal linear chats this just links an assistant answer to the user prompt before it. However, Claude Code can have branching in some advanced cases (like sub-agents or when using the `/plan` feature), where `isSidechain: true` may indicate a side conversation. The `parentUuid` / `isSidechain` fields let Claude keep track of which messages belong to the main thread vs. a spawned sub-task thread (ensuring, for example, that responses go to the correct branch of the conversation).
- **User vs. internal messages:** The `userType` field can distinguish human-initiated messages (`"external"`) from system-injected ones. Typically, all your prompts and Claude's replies are marked external [11]. If Claude were to internally add a message (not common in normal operation, but possibly for system notices), it might use a different `userType`. In general, this field isn't something users manipulate, but it exists in the log for completeness.
- **Claude Code version and config:** The `version` field logs the Claude Code client version, which can be useful for debugging or analyzing behavior differences across versions [12]. The `gitBranch`

(if present) provides context about what code branch was active, which can be relevant to the conversation (Claude might behave slightly differently if it knows it's on a certain branch, e.g. for committing changes). These fields are metadata and don't directly affect the content of messages, but they help contextualize the session.

Overall, these special fields ensure that when Claude loads or processes the log, it has all necessary context (where you were, what was happening in your repo, which message is replying to which, etc.). They're mostly for Claude's internal use or for tools analyzing the history; as a user, you typically see just the conversation text, but behind the scenes Claude uses this metadata to maintain continuity.

Chronology and Editing of Logs

The `.jsonl` project files are **chronologically ordered** transcripts of the session. Each new user query or Claude response (including tool outputs) simply gets appended as a new JSON line at the end of the file [4]. The integrity of time-order is preserved by the timestamps on each entry and the append-only nature of JSONL. There's no built-in mechanism to go back and "insert" or modify lines in the middle of a log – if Claude revises something or the user undoes an action, those events will themselves be logged as new entries rather than retroactively editing prior lines. In other words, the log is an immutable record of the conversation as it happened.

This means you generally **should not manually edit** a `.jsonl` transcript to try to change Claude's memory – Claude won't read a tampered log to alter its state. Instead, to undo or rewind, you use Claude Code's commands (which themselves get logged). For example, Claude Code's checkpoint/rewind feature can revert the code or conversation to an earlier point, but under the hood this will either start a new session or add new entries indicating the rewind; it doesn't rewrite the old log lines. The JSONL files are meant as **historical logs**, not live data stores to modify.

By default, Claude Code will automatically clean up (delete) old conversation logs after a certain period (30 days) to save space [18]. This is controlled by the `cleanupPeriodDays` setting. You can extend it (for example, setting it to a very large number to effectively keep logs indefinitely [19]) if you want to preserve history. Otherwise, be aware that without configuration, transcripts older than a month get purged from `.claude/projects`. (The rationale is that these can accumulate many hundreds of MB of data over time [20].)

Using JSONL Logs to Resume Conversations

One of the powerful features of Claude Code is the ability to **resume a previous session** – and these JSONL files are exactly how it does that. Whenever a session ends, its full history is saved in the `.jsonl` file. Later, you can re-launch Claude in the same project and use the CLI flags or commands to pick up where you left off. For example, running `claude --continue` will automatically continue the most recent session, and `claude --resume` will let you pick from a list of past sessions in that project [21] [22]. This works by loading the corresponding JSONL file from disk.

When Claude resumes a conversation, it **deserializes the entire message history** from the JSONL log back into memory [23]. In practice, Claude reconstructs every turn (user messages, assistant replies, even tool call results) in order, so that the context is exactly as it was. All the metadata and content we discussed are read

in to re-establish state. The official docs confirm that on resuming, "the entire conversation history is restored to maintain context," including tool usage and results, and the conversation resumes with all previous context intact [23]. This means Claude will remember what you were doing: it knows what code it saw, what changes it made, what the discussion was – as if you never stopped. Any *in-progress* state, like open TODOs or the last question asked, is preserved since the assistant can see the prior turns.

From the user's perspective, after using `--resume` or `--continue`, Claude will typically display the past messages (so you can scroll through the chat history) and then you can continue chatting. The model behind Claude is stateless on its own – it doesn't "remember" past sessions – but by feeding it the loaded transcript as context, it effectively regains the memory from that session. In summary, the JSONL logs are the source of truth that Claude uses to **reload conversations** on demand. This local-first design also means your data stays on your machine; conversation history is *"stored locally on your machine"* and only sent to Claude when you actively resume or reference it [24].

Summary of Format Rules and Tips

- **JSON Lines structure:** Each conversation log file is a series of independent JSON objects, one per line [3]. There's no enclosing array or comma – it's line-delimited for easy appending. Ensure any parser reads it line-by-line.
- **Schema:** Every entry has a role (user/assistant/tool), content (which may be text or structured or null if replaced by a tool call), a timestamp, and various metadata fields (IDs, context info). User messages and assistant messages have slightly different structures (assistant messages often containing more fields like model and usage) but both follow the general schema [13] [9]. Tool outputs are logged as their own entries with `role: "tool"` or equivalent.
- **Ordering:** The log should reflect the true sequence of interactions. User and assistant turns alternate. If the assistant uses tools, you will see an assistant entry for the tool invocation followed by a tool result entry, then the assistant's next turn. The formatting *must* respect the tool-use→tool-result pairing [15]. Out-of-order or missing entries will cause errors or confusion.
- **No inline editing:** Treat the JSONL as an immutable history. Don't remove or edit lines expecting Claude to "forget" something – use the provided `/rewind` or start a fresh session if needed. The system doesn't support partial edits of a session log; it loads it wholesale.
- **Metadata usage:** Fields like `sessionId`, `cwd`, etc., are mostly behind-the-scenes. They ensure that when you resume or when Claude is performing actions, it knows the proper context (project and session). As a user, you typically don't modify these – Claude Code manages them for you.
- **Resuming sessions:** You can safely close Claude and later resume. The `.jsonl` file will be read in full on resume so Claude sees the entire conversation context again [23]. If you want to archive or share a conversation, you can copy the JSONL (or convert it to a nicer format via community tools). Just remember that by default old sessions might be cleared after 30 days unless you adjust settings [18].

References: The information above is drawn from official Claude documentation and developer explorations of Claude Code's logs. For instance, Anthropic's docs on *Common workflows* detail how session history is saved and restored [24] [23], and various blog posts have reverse-engineered the JSONL schema [13] [7]. Tools like `claude-notes` and others confirm the structure by parsing these files [7]. In short, the `.jsonl` project files are a chronological record of the chat (plus actions) that Claude uses to maintain continuity across your coding sessions. Keeping them intact (and extending the retention if needed) allows you to fully leverage the *"resume where you left off"* capability of Claude Code.

[1] DevLog: Making A Claude Code History Viewer - John Damask
https://johndamask.substack.com/p/devlog-making-a-claude-code-history

[2] [4] [8] [12] [13] [14] [17] Automate Your AI Workflows with Claude Code Hooks | Butler's Log
https://blog.gitbutler.com/automate-your-ai-workflows-with-claude-code-hooks

[3] [5] [6] [7] [16] Simple Notes for Claude Projects – Bright Coding – Blog pour les developpeurs
https://www.blog.brightcoding.dev/2025/08/04/simple-notes-for-claude-projects/

[9] [10] [11] Analyzing Claude Code Interaction Logs with DuckDB - Liam ERD
https://liambx.com/blog/claude-code-log-analysis-with-duckdb

[15] Missing Tool Result Block After Tool Use This error suggests that after a tool use block in the conversation, you did not include the corresponding tool result block, which is required by the API. To resolve this: 1. Ensure that any `tool_use` block is i · Issue #473 · anthropics/claude-code · GitHub
https://github.com/anthropics/claude-code/issues/473?timeline_page=1

[18] [19] [20] Don't let Claude Code delete your session logs
https://simonwillison.net/2025/Oct/22/claude-code-logs/

[21] [22] [23] [24] Common workflows - Claude Docs
https://docs.claude.com/en/docs/claude-code/common-workflows