Compaction internals

Session Management Deep Dive

This document explains how OpenClaw manages sessions end-to-end:

- **Session routing** (how inbound messages map to a `sessionKey`)
- **Session store** (`sessions.json`) and what it tracks
- **Transcript persistence** (`*.jsonl`) and its structure
- **Transcript hygiene** (provider-specific fixups before runs)
- **Context limits** (context window vs tracked tokens)
- **Compaction** (manual + auto-compaction) and where to hook pre-compaction work
- **Silent housekeeping** (e.g. memory writes that shouldn't produce user-visible output)

If you want a higher-level overview first, start with:

- /concepts/session
- /concepts/compaction
- /concepts/session-pruning
- /reference/transcript-hygiene

Source of truth: the Gateway

OpenClaw is designed around a single **Gateway process** that owns session state.

- UIs (macOS app, web Control UI, TUI) should query the Gateway for session lists and token counts.
- In remote mode, session files are on the remote host; “checking your local Mac files” won’t reflect what the Gateway is using.

Two persistence layers

OpenClaw persists sessions in two layers:

1. **Session store (`sessions.json`)**
   - Key/value map: `sessionKey -> SessionEntry`
   - Small, mutable, safe to edit (or delete entries)
   - Tracks session metadata (current session id, last activity, toggles, token counters, etc.)
2. **Transcript (`<sessionId>.jsonl`)**
   - Append-only transcript with tree structure (entries have `id` + `parentId`)
   - Stores the actual conversation + tool calls + compaction summaries
   - Used to rebuild the model context for future turns

On-disk locations

Per agent, on the Gateway host:

- Store: `~/.openclaw/agents/<agentId>/sessions/sessions.json`
- Transcripts: `~/.openclaw/agents/<agentId>/sessions/<sessionId>.jsonl`
  - Telegram topic sessions: `.../<sessionId>-topic-<threadId>.jsonl`

OpenClaw resolves these via `src/config/sessions.ts` .

Session keys (`sessionKey`)

A `sessionKey` identifies *which conversation bucket* you're in (routing + isolation).

Common patterns:

- Main/direct chat (per agent): `agent:<agentId>:<mainKey>` (default `main`)
- Group: `agent:<agentId>:<channel>:group:<id>`
- Room/channel (Discord/Slack): `agent:<agentId>:<channel>:channel:<id>` or `...:room:<id>`
- Cron: `cron:<job.id>`
- Webhook: `hook:<uuid>` (unless overridden)

The canonical rules are documented at **/concepts/session**.

Session ids (`sessionId`)

Each `sessionKey` points at a current `sessionId` (the transcript file that continues the conversation).

Rules of thumb:

- **Reset** (`/new` , `/reset`) creates a new `sessionId` for that `sessionKey` .
- **Daily reset** (default 4:00 AM local time on the gateway host) creates a new `sessionId` on the next message after the reset boundary.
- **Idle expiry** (`session.reset.idleMinutes` or legacy `session.idleMinutes`) creates a new `sessionId` when a message arrives after the idle window. When daily + idle are both configured, whichever expires first wins.

Implementation detail: the decision happens in `initSessionState()` in `src/auto-reply/reply/session.ts` .

Session store schema (`sessions.json`)

The store's value type is `SessionEntry` in `src/config/sessions.ts` .

Key fields (not exhaustive):

- `sessionId` : current transcript id (filename is derived from this unless `sessionFile` is set)
- `updatedAt` : last activity timestamp
- `sessionFile` : optional explicit transcript path override
- `chatType` : `direct | group | room` (helps UIs and send policy)
- `provider` , `subject` , `room` , `space` , `displayName` : metadata for group/channel labeling
- Toggles:
  - `thinkingLevel` , `verboseLevel` , `reasoningLevel` , `elevatedLevel`
  - `sendPolicy` (per-session override)
- Model selection:
  - `providerOverride` , `modelOverride` , `authProfileOverride`

- Token counters (best-effort / provider-dependent):
  - `inputTokens`, `outputTokens`, `totalTokens`, `contextTokens`
- `compactionCount`: how often auto-compaction completed for this session key
- `memoryFlushAt`: timestamp for the last pre-compaction memory flush
- `memoryFlushCompactionCount`: compaction count when the last flush ran

The store is safe to edit, but the Gateway is the authority: it may rewrite or rehydrate entries as sessions run.

Transcript structure (`*.jsonl`)

Transcripts are managed by `@mariozechner/pi-coding-agent`'s `SessionManager`.

The file is JSONL:

- First line: session header (`type: "session"`, includes `id`, `cwd`, `timestamp`, optional `parentSession`)
- Then: session entries with `id` + `parentId` (tree)

Notable entry types:

- `message`: user/assistant/toolResult messages
- `custom_message`: extension-injected messages that *do* enter model context (can be hidden from UI)
- `custom`: extension state that does *not* enter model context
- `compaction`: persisted compaction summary with `firstKeptEntryId` and `tokensBefore`
- `branch_summary`: persisted summary when navigating a tree branch

OpenClaw intentionally does **not** “fix up” transcripts; the Gateway uses `SessionManager` to read/write them.

›

Context windows vs tracked tokens

Two different concepts matter:

1. **Model context window**: hard cap per model (tokens visible to the model)
2. **Session store counters**: rolling stats written into `sessions.json` (used for /status and dashboards)

If you’re tuning limits:

- The context window comes from the model catalog (and can be overridden via config).
- `contextTokens` in the store is a runtime estimate/reporting value; don’t treat it as a strict guarantee.

For more, see `/token-use`.

Compaction: what it is

Compaction summarizes older conversation into a persisted `compaction` entry in the transcript and keeps recent messages intact.

After compaction, future turns see:

- The compaction summary
- Messages after `firstKeptEntryId`

Compaction is **persistent** (unlike session pruning). See /concepts/session-pruning.

›

When auto-compaction happens (Pi runtime)

In the embedded Pi agent, auto-compaction triggers in two cases:

1. **Overflow recovery**: the model returns a context overflow error → compact → retry.
2. **Threshold maintenance**: after a successful turn, when:

```
contextTokens > contextWindow - reserveTokens
```

Where:

- `contextWindow` is the model's context window
- `reserveTokens` is headroom reserved for prompts + the next model output

These are Pi runtime semantics (OpenClaw consumes the events, but Pi decides when to compact).

Compaction settings (`reserveTokens` , `keepRecentTokens`)

Pi's compaction settings live in Pi settings:

```
{
  compaction: {
    enabled: true,
    reserveTokens: 16384,
    keepRecentTokens: 20000,
  },
}
```

OpenClaw also enforces a safety floor for embedded runs:

- If `compaction.reserveTokens < reserveTokensFloor`, OpenClaw bumps it.
- Default floor is `20000` tokens.
- Set `agents.defaults.compaction.reserveTokensFloor: 0` to disable the floor.
- If it's already higher, OpenClaw leaves it alone.

Why: leave enough headroom for multi-turn "housekeeping" (like memory writes) before compaction becomes unavoidable.

Implementation: `ensurePiCompactionReserveTokens()` in `src/agents/pi-settings.ts` (called from `src/agents/pi-embedded-runner.ts`).

User-visible surfaces

You can observe compaction and session state via:

- `/status` (in any chat session)
- `openclaw status` (CLI)
- `openclaw sessions` / `sessions --json`
- Verbose mode: `🧹 Auto-compaction complete` + compaction count

Silent housekeeping (`NO_REPLY`)

OpenClaw supports “silent” turns for background tasks where the user should not see intermediate output.

Convention:

- The assistant starts its output with `NO_REPLY` to indicate “do not deliver a reply to the user”.
- OpenClaw strips/suppresses this in the delivery layer.

As of `2026.1.10` , OpenClaw also suppresses **draft/typing streaming** when a partial chunk begins with `NO_REPLY` , so silent operations don’t leak partial output mid-turn.

Pre-compaction “memory flush” (implemented)

Goal: before auto-compaction happens, run a silent agentic turn that writes durable state to disk (e.g. `memory/YYYY-MM-DD.md` in the agent workspace) so compaction can’t erase critical context.

OpenClaw uses the **pre-threshold flush** approach:

1. Monitor session context usage.
2. When it crosses a “soft threshold” (below Pi’s compaction threshold), run a silent “write memory now” directive to the agent.
3. Use `NO_REPLY` so the user sees nothing.

Config (`agents.defaults.compaction.memoryFlush`):

- `enabled` (default: `true`)
- `softThresholdTokens` (default: `4000`)

- `prompt` (user message for the flush turn)
- `systemPrompt` (extra system prompt appended for the flush turn)

Notes:

- The default prompt/system prompt include a `NO_REPLY` hint to suppress delivery.
- The flush runs once per compaction cycle (tracked in `sessions.json`).
- The flush runs only for embedded Pi sessions (CLI backends skip it).
- The flush is skipped when the session workspace is read-only (`workspaceAccess: "ro"` or `"none"`).
- See **Memory** for the workspace file layout and write patterns.

Pi also exposes a `session_before_compact` hook in the extension API, but OpenClaw's flush logic lives on the Gateway side today.

Troubleshooting checklist

- Session key wrong? Start with **/concepts/session** and confirm the `sessionKey` in `/status`.
- Store vs transcript mismatch? Confirm the Gateway host and the store path from `openclaw status`.
- Compaction spam? Check:
  - model context window (too small)
  - compaction settings (`reserveTokens` too high for the model window can cause earlier compaction)
  - tool-result bloat: enable/tune session pruning

Silent turns leaking? Confirm the reply starts with `NO_REPLY` (exact token) and you’re on a build that includes the streaming suppression fix.

‹ Node.js Setup ›