Sessions and memory

Memory

OpenClaw memory is **plain Markdown in the agent workspace**. The files are the source of truth; the model only “remembers” what gets written to disk.

Memory search tools are provided by the active memory plugin (default: `memory-core`). Disable memory plugins with `plugins.slots.memory = "none"`.

Memory files (Markdown)

The default workspace layout uses two memory layers:

- `memory/YYYY-MM-DD.md`
  - Daily log (append-only).
  - Read today + yesterday at session start.
- `MEMORY.md` (optional)
  - Curated long-term memory.
  - **Only load in the main, private session** (never in group contexts).

These files live under the workspace (`agents.defaults.workspace`, default `~/.openclaw/workspace`). See **Agent workspace** for the full layout.

When to write memory

Decisions, preferences, and durable facts go to `MEMORY.md`.

Day-to-day notes and running context go to `memory/YYYY-MM-DD.md`.

If someone says "remember this," write it down (do not keep it in RAM).

This area is still evolving. It helps to remind the model to store memories; it will know what to do.

If you want something to stick, **ask the bot to write it** into memory.

Automatic memory flush (pre-compaction ping)

When a session is **close to auto-compaction**, OpenClaw triggers a **silent, agentic turn** that reminds the model to write durable memory **before** the context is compacted. The default prompts explicitly say the model *may reply*, but usually `NO_REPLY` is the correct response so the user never sees this turn.

This is controlled by `agents.defaults.compaction.memoryFlush`:

```
{
  agents: {
    defaults: {
      compaction: {
        reserveTokensFloor: 20000,
        memoryFlush: {
          enabled: true,
          softThresholdTokens: 4000,
          systemPrompt: "Session nearing compaction. Store durable memories now."
          prompt: "Write any lasting notes to memory/YYYY-MM-DD.md; reply with NO
        },
      },
    },
  },
}
```

Details:

- **Soft threshold**: flush triggers when the session token estimate crosses `contextWindow - reserveTokensFloor - softThresholdTokens`.
- **Silent** by default: prompts include `NO_REPLY` so nothing is delivered.
- **Two prompts**: a user prompt plus a system prompt append the reminder.
- **One flush per compaction cycle** (tracked in `sessions.json`).
- **Workspace must be writable**: if the session runs sandboxed with `workspaceAccess: "ro"` or `"none"`, the flush is skipped.

For the full compaction lifecycle, see **Session management + compaction**.

Vector memory search

OpenClaw can build a small vector index over `MEMORY.md` and `memory/*.md` so semantic queries can find related notes even when wording differs.

Defaults:

- Enabled by default.
- Watches memory files for changes (debounced).
- Configure memory search under `agents.defaults.memorySearch` (not top-level `memorySearch`).
- Uses remote embeddings by default. If `memorySearch.provider` is not set, OpenClaw auto-selects:
  1. `local` if a `memorySearch.local.modelPath` is configured and the file exists.
  2. `openai` if an OpenAI key can be resolved.
  3. `gemini` if a Gemini key can be resolved.

4. `voyage` if a Voyage key can be resolved.
5. Otherwise memory search stays disabled until configured.

- Local mode uses node-llama-cpp and may require `pnpm approve-builds`.
- Uses sqlite-vec (when available) to accelerate vector search inside SQLite.

Remote embeddings **require** an API key for the embedding provider. OpenClaw resolves keys from auth profiles, `models.providers.*.apiKey`, or environment variables. Codex OAuth only covers chat/completions and does **not** satisfy embeddings for memory search. For Gemini, use `GEMINI_API_KEY` or `models.providers.google.apiKey`. For Voyage, use `VOYAGE_API_KEY` or `models.providers.voyage.apiKey`. When using a custom OpenAI-compatible endpoint, set `memorySearch.remote.apiKey` (and optional `memorySearch.remote.headers`).

QMD backend (experimental)

Set `memory.backend = "qmd"` to swap the built-in SQLite indexer for **QMD**: a local-first search sidecar that combines BM25 + vectors + reranking. Markdown stays the source of truth; OpenClaw shells out to QMD for retrieval. Key points:

Prereqs

- Disabled by default. Opt in per-config (`memory.backend = "qmd"`).
- Install the QMD CLI separately (`bun install -g https://github.com/tobi/qmd` or grab a release) and make sure the `qmd` binary is on the gateway's `PATH`.
- QMD needs an SQLite build that allows extensions (`brew install sqlite` on macOS).
- QMD runs fully locally via Bun + `node-llama-cpp` and auto-downloads GGUF models from HuggingFace on first use (no separate Ollama daemon required).

The gateway runs QMD in a self-contained XDG home under `~/.openclaw/agents/<agentId>/qmd/` by setting `XDG_CONFIG_HOME` and `XDG_CACHE_HOME`.

OS support: macOS and Linux work out of the box once Bun + SQLite are installed. Windows is best supported via WSL2.

How the sidecar runs

The gateway writes a self-contained QMD home under `~/.openclaw/agents/<agentId>/qmd/` (config + cache + sqlite DB).

Collections are created via `qmd collection add` from `memory.qmd.paths` (plus default workspace memory files), then `qmd update` + `qmd embed` run on boot and on a configurable interval (`memory.qmd.update.interval`, default 5 m).

The gateway now initializes the QMD manager on startup, so periodic update timers are armed even before the first `memory_search` call.

Boot refresh now runs in the background by default so chat startup is not blocked; set `memory.qmd.update.waitForBootSync = true` to keep the previous blocking behavior.

Searches run via `memory.qmd.searchMode` (default `qmd search --json`; also supports `vsearch` and `query`). If the selected mode rejects flags on your QMD build, OpenClaw retries with `qmd query`. If QMD fails or the binary is missing, OpenClaw automatically falls back to the builtin SQLite manager so memory tools keep working.

OpenClaw does not expose QMD embed batch-size tuning today; batch behavior is controlled by QMD itself.

First search may be slow: QMD may download local GGUF models (reranker/query expansion) on the first `qmd query` run.

OpenClaw sets `XDG_CONFIG_HOME` / `XDG_CACHE_HOME` automatically when it runs QMD.

If you want to pre-download models manually (and warm the same index OpenClaw uses), run a one-off query with the agent's XDG

dirs.

OpenClaw's QMD state lives under your **state dir** (defaults to `~/.openclaw`). You can point `qmd` at the exact same index by exporting the same XDG vars OpenClaw uses:

```
# Pick the same state dir OpenClaw uses
STATE_DIR="${OPENCLAW_STATE_DIR:-$HOME/.openclaw}"

export XDG_CONFIG_HOME="$STATE_DIR/agents/main/qmd/xdg-config"
export XDG_CACHE_HOME="$STATE_DIR/agents/main/qmd/xdg-cache"

# (Optional) force an index refresh + embeddings
qmd update
qmd embed

# Warm up / trigger first-time model downloads
qmd query "test" -c memory-root --json >/dev/null 2>&1
```

Config surface (`memory.qmd.*`)

- `command` (default `qmd`): override the executable path.
- `searchMode` (default `search`): pick which QMD command backs `memory_search` (`search`, `vsearch`, `query`).
- `includeDefaultMemory` (default `true`): auto-index `MEMORY.md` + `memory/**/*.md`.
- `paths[]`: add extra directories/files (`path`, optional `pattern`, optional stable `name`).
- `sessions`: opt into session JSONL indexing (`enabled`, `retentionDays`, `exportDir`).
- `update`: controls refresh cadence and maintenance execution: (`interval`, `debounceMs`, `onBoot`, `waitForBootSync`, `embedInterval`, `commandTimeoutMs`, `updateTimeoutMs`, `embedTimeoutMs`).
- `limits`: clamp recall payload (`maxResults`, `maxSnippetChars`, `maxInjectedChars`, `timeoutMs`).

`scope` : same schema as `session.sendPolicy` . Default is DM-only (`deny` all, `allow` direct chats); loosen it to surface QMD hits in groups/channels.

`match.keyPrefix` matches the **normalized** session key (lowercased, with any leading `agent:<id>:` stripped). Example: `discord:channel:` .

`match.rawKeyPrefix` matches the **raw** session key (lowercased), including `agent:<id>:` . Example: `agent:main:discord:` .

Legacy: `match.keyPrefix: "agent:..."` is still treated as a raw-key prefix, but prefer `rawKeyPrefix` for clarity.

When `scope` denies a search, OpenClaw logs a warning with the derived `channel` / `chatType` so empty results are easier to debug.

Snippets sourced outside the workspace show up as `qmd/<collection>/<relative-path>` in `memory_search` results; `memory_get` understands that prefix and reads from the configured QMD collection root.

When `memory.qmd.sessions.enabled = true` , OpenClaw exports sanitized session transcripts (User/Assistant turns) into a dedicated QMD collection under `~/.openclaw/agents/<id>/qmd/sessions/` , so `memory_search` can recall recent conversations without touching the builtin SQLite index.

`memory_search` snippets now include a `Source: <path#line>` footer when `memory.citations` is `auto` / `on` ; set `memory.citations = "off"` to keep the path metadata internal (the agent still receives the path for `memory_get` , but the snippet text omits the footer and the system prompt warns the agent not to cite it).

Example

```
memory: {
  backend: "qmd",
  citations: "auto",
  qmd: {
    includeDefaultMemory: true,
    update: { interval: "5m", debounceMs: 15000 },
    limits: { maxResults: 6, timeoutMs: 4000 },
    scope: {
      default: "deny",
      rules: [
        { action: "allow", match: { chatType: "direct" } },
        // Normalized session-key prefix (strips `agent:<id>:`).
        { action: "deny", match: { keyPrefix: "discord:channel:" } },
        // Raw session-key prefix (includes `agent:<id>:`).
        { action: "deny", match: { rawKeyPrefix: "agent:main:discord:" } },
      ]
    },
    paths: [
      { name: "docs", path: "~/notes", pattern: "**/*.md" }
    ]
  }
}
```

Citations & fallback

`memory.citations` applies regardless of backend (`auto` / `on` / `off`).

When `qmd` runs, we tag `status().backend = "qmd"` so diagnostics show which engine served the results. If the QMD subprocess exits or JSON output can't be parsed, the search manager logs a warning and returns the builtin provider (existing Markdown embeddings) until QMD recovers.

Additional memory paths

If you want to index Markdown files outside the default workspace layout, add explicit paths:

```
agents: {
  defaults: {
    memorySearch: {
      extraPaths: ["../team-docs", "/srv/shared-notes/overview.md"]
    }
  }
}
```

Notes:

- Paths can be absolute or workspace-relative.
- Directories are scanned recursively for `.md` files.
- Only Markdown files are indexed.
- Symlinks are ignored (files or directories).

Gemini embeddings (native)

Set the provider to `gemini` to use the Gemini embeddings API directly:

```
agents: {
  defaults: {
    memorySearch: {
      provider: "gemini",
      model: "gemini-embedding-001",
      remote: {
        apiKey: "YOUR_GEMINI_API_KEY"
      }
    }
  }
}
```

Notes:

- `remote.baseUrl` is optional (defaults to the Gemini API base URL).

`remote.headers` lets you add extra headers if needed.

Default model: `gemini-embedding-001`.

If you want to use a **custom OpenAI-compatible endpoint** (OpenRouter, vLLM, or a proxy), you can use the `remote` configuration with the OpenAI provider:

```
agents: {
  defaults: {
    memorySearch: {
      provider: "openai",
      model: "text-embedding-3-small",
      remote: {
        baseUrl: "https://api.example.com/v1/",
        apiKey: "YOUR_OPENAI_COMPAT_API_KEY",
        headers: { "X-Custom-Header": "value" }
      }
    }
  }
}
```

If you don't want to set an API key, use `memorySearch.provider = "local"` or set `memorySearch.fallback = "none"`.

Fallbacks:

`memorySearch.fallback` can be `openai`, `gemini`, `local`, or `none`.

The fallback provider is only used when the primary embedding provider fails.

Batch indexing (OpenAI + Gemini + Voyage):

Disabled by default. Set `agents.defaults.memorySearch.remote.batch.enabled = true` to enable for large-corpus indexing (OpenAI, Gemini, and Voyage).

Default behavior waits for batch completion; tune `remote.batch.wait`, `remote.batch.pollIntervalMs`, and `remote.batch.timeoutMinutes` if needed.

Set `remote.batch.concurrency` to control how many batch jobs we submit in parallel (default: 2).

Batch mode applies when `memorySearch.provider = "openai"` or `"gemini"` and uses the corresponding API key.

Gemini batch jobs use the async embeddings batch endpoint and require Gemini Batch API availability.

Why OpenAI batch is fast + cheap:

For large backfills, OpenAI is typically the fastest option we support because we can submit many embedding requests in a single batch job and let OpenAI process them asynchronously.

OpenAI offers discounted pricing for Batch API workloads, so large indexing runs are usually cheaper than sending the same requests synchronously.

See the OpenAI Batch API docs and pricing for details:

https://platform.openai.com/docs/api-reference/batch

https://platform.openai.com/pricing

Config example:

```
agents: {
  defaults: {
    memorySearch: {
      provider: "openai",
      model: "text-embedding-3-small",
      fallback: "openai",
      remote: {
        batch: { enabled: true, concurrency: 2 }
      },
      sync: { watch: true }
    }
  }
}
```

Tools:

- `memory_search` — returns snippets with file + line ranges.
- `memory_get` — read memory file content by path.

Local mode:

- Set `agents.defaults.memorySearch.provider = "local"`.
- Provide `agents.defaults.memorySearch.local.modelPath` (GGUF or `hf:` URI).
- Optional: set `agents.defaults.memorySearch.fallback = "none"` to avoid remote fallback.

How the memory tools work

`memory_search` semantically searches Markdown chunks (~400 token target, 80-token overlap) from `MEMORY.md` + `memory/**/*.md`. It returns snippet text (capped ~700 chars), file path, line range, score, provider/model, and whether we fell back from local → remote embeddings. No full file payload is returned.

`memory_get` reads a specific memory Markdown file (workspace-relative), optionally from a starting line and for N lines. Paths

outside `MEMORY.md` / `memory/` are rejected.

Both tools are enabled only when `memorySearch.enabled` resolves true for the agent.

›

What gets indexed (and when)

File type: Markdown only (`MEMORY.md` , `memory/**/*.md`).

Index storage: per-agent SQLite at `~/.openclaw/memory/<agentId>.sqlite` (configurable via `agents.defaults.memorySearch.store.path` , supports `{agentId}` token).

Freshness: watcher on `MEMORY.md` + `memory/` marks the index dirty (debounce 1.5s). Sync is scheduled on session start, on search, or on an interval and runs asynchronously. Session transcripts use delta thresholds to trigger background sync.

Reindex triggers: the index stores the embedding **provider/model + endpoint fingerprint + chunking params**. If any of those change, OpenClaw automatically resets and reindexes the entire store.

Hybrid search (BM25 + vector)

When enabled, OpenClaw combines:

Vector similarity (semantic match, wording can differ)

BM25 keyword relevance (exact tokens like IDs, env vars, code symbols)

If full-text search is unavailable on your platform, OpenClaw falls back to vector-only search.

Why hybrid?

Vector search is great at “this means the same thing”:

“Mac Studio gateway host” vs “the machine running the gateway”

“debounce file updates” vs “avoid indexing on every write”

But it can be weak at exact, high-signal tokens:

- IDs (`a828e60` , `b3b9895a…`)
- code symbols (`memorySearch.query.hybrid`)
- error strings (“sqlite-vec unavailable”)

BM25 (full-text) is the opposite: strong at exact tokens, weaker at paraphrases. Hybrid search is the pragmatic middle ground: **use both retrieval signals** so you get good results for both “natural language” queries and “needle in a haystack” queries.

How we merge results (the current design)

Implementation sketch:

1. Retrieve a candidate pool from both sides:
   - **Vector**: top `maxResults * candidateMultiplier` by cosine similarity.
   - **BM25**: top `maxResults * candidateMultiplier` by FTS5 BM25 rank (lower is better).
2. Convert BM25 rank into a 0..1-ish score:

```
textScore = 1 / (1 + max(0, bm25Rank))
```

3. Union candidates by chunk id and compute a weighted score:

```
finalScore = vectorWeight * vectorScore + textWeight * textScore
```

Notes:

- `vectorWeight` + `textWeight` is normalized to 1.0 in config resolution, so weights behave as percentages.
- If embeddings are unavailable (or the provider returns a zero-vector), we still run BM25 and return keyword matches.

If FTS5 can't be created, we keep vector-only search (no hard failure).

This isn't "IR-theory perfect", but it's simple, fast, and tends to improve recall/precision on real notes. If we want to get fancier later, common next steps are Reciprocal Rank Fusion (RRF) or score normalization (min/max or z-score) before mixing.

Post-processing pipeline

After merging vector and keyword scores, two optional post-processing stages refine the result list before it reaches the agent:

```
Vector + Keyword → Weighted Merge → Temporal Decay → Sort → MMR -
```

Both stages are **off by default** and can be enabled independently.

MMR re-ranking (diversity)

When hybrid search returns results, multiple chunks may contain similar or overlapping content. For example, searching for "home network setup" might return five nearly identical snippets from different daily notes that all mention the same router configuration.

MMR (Maximal Marginal Relevance) re-ranks the results to balance relevance with diversity, ensuring the top results cover different aspects of the query instead of repeating the same information.

How it works:

1. Results are scored by their original relevance (vector + BM25 weighted score).
2. MMR iteratively selects results that maximize: `λ × relevance - (1-λ) × max_similarity_to_selected` .

3. Similarity between results is measured using Jaccard text similarity on tokenized content.

The `lambda` parameter› controls the trade-off:

`lambda = 1.0` → pure relevance (no diversity penalty)

`lambda = 0.0` → maximum diversity (ignores relevance)

Default: `0.7` (balanced, slight relevance bias)

Example — query: "home network setup"

Given these memory files:

```
memory/2026-02-10.md  ⟶ "Configured Omada router, set VLAN 10 for Io
memory/2026-02-08.md  ⟶ "Configured Omada router, moved IoT to VLAN 10"
memory/2026-02-05.md  ⟶ "Set up AdGuard DNS on 192.168.10.2"
memory/network.md     ⟶ "Router: Omada ER605, AdGuard: 192.168.10.2, VLAN 10: Io
```

Without MMR — top 3 results:

```
1. memory/2026-02-10.md  (score: 0.92)  ⟵ router + VLAN
2. memory/2026-02-08.md  (score: 0.89)  ⟵ router + VLAN (near-duplicate!)
3. memory/network.md     (score: 0.85)  ⟵ reference doc
```

With MMR (λ=0.7) — top 3 results:

```
1. memory/2026-02-10.md  (score: 0.92)  ⟵ router + VLAN
2. memory/network.md     (score: 0.85)  ⟵ reference doc (diverse!)
3. memory/2026-02-05.md  (score: 0.78)  ⟵ AdGuard DNS (diverse!)
```

The near-duplicate from Feb 8 drops out, and the agent gets three distinct pieces of information.

When to enable: If you notice `memory_search` returning redundant or near-duplicate snippets, especially with daily notes that often repeat similar information across days.

Temporal decay (recency boost)

Agents with daily notes accumulate hundreds of dated files over time. Without decay, a well-worded note from six months ago can outrank yesterday's update on the same topic.

Temporal decay applies an exponential multiplier to scores based on the age of each result, so recent memories naturally rank higher while old ones fade:

```
decayedScore = score × e^(-λ × ageInDays)
```

where `λ = ln(2) / halfLifeDays`.

With the default half-life of 30 days:

- Today's notes: **100%** of original score
- 7 days ago: **~84%**
- 30 days ago: **50%**
- 90 days ago: **12.5%**
- 180 days ago: **~1.6%**

Evergreen files are never decayed:

- `MEMORY.md` (root memory file)
- Non-dated files in `memory/` (e.g., `memory/projects.md`, `memory/network.md`)
- These contain durable reference information that should always rank normally.

Dated daily files (`memory/YYYY-MM-DD.md`) use the date extracted from the filename. Other sources (e.g., session transcripts) fall back to file modification time (`mtime`).

Example — query: "what's Rod's work schedule?"

Given these memory files (today is Feb 10):

```
memory/2025-09-15.md  → "Rod works Mon-Fri, standup at 10am, pairing
memory/2026-02-10.md  → "Rod has standup at 14:15, 1:1 with Zeb at 14:45"    (to
memory/2026-02-03.md  → "Rod started new team, standup moved to 14:15"        (7
```

Without decay:

```
1. memory/2025-09-15.md  (score: 0.91)  ← best semantic match, but s
2. memory/2026-02-10.md  (score: 0.82)
3. memory/2026-02-03.md  (score: 0.80)
```

With decay (halfLife=30):

```
1. memory/2026-02-10.md  (score: 0.82 × 1.00 = 0.82)  ← today, no de
2. memory/2026-02-03.md  (score: 0.80 × 0.85 = 0.68)  ← 7 days, mild decay
3. memory/2025-09-15.md  (score: 0.91 × 0.03 = 0.03)  ← 148 days, nearly gone
```

The stale September note drops to the bottom despite having the best raw semantic match.

When to enable: If your agent has months of daily notes and you find that old, stale information outranks recent context. A half-life of 30 days works well for daily-note-heavy workflows; increase it (e.g., 90 days) if you reference older notes frequently.

Configuration

Both features are configured under `memorySearch.query.hybrid`:

```
agents: {
  defaults: {
    memorySearch: {
      query: {
        hybrid: {
          enabled: true,
          vectorWeight: 0.7,
          textWeight: 0.3,
          candidateMultiplier: 4,
          // Diversity: reduce redundant results
          mmr: {
            enabled: true,    // default: false
            lambda: 0.7       // 0 = max diversity, 1 = max relevance
          },
          // Recency: boost newer memories
          temporalDecay: {
            enabled: true,    // default: false
            halfLifeDays: 30  // score halves every 30 days
          }
        }
      }
    }
  }
}
```

You can enable either feature independently:

- **MMR only** — useful when you have many similar notes but age doesn't matter.
- **Temporal decay only** — useful when recency matters but your results are already diverse.
- **Both** — recommended for agents with large, long-running daily note histories.

Embedding cache

OpenClaw can cache **chunk embeddings** in SQLite so reindexing and frequent updates (especially session transcripts) don't re-embed unchanged text.

Config:

```
agents: {
  defaults: {
    memorySearch: {
      cache: {
        enabled: true,
        maxEntries: 50000
      }
    }
  }
}
```

Session memory search (experimental)

You can optionally index **session transcripts** and surface them via `memory_search` . This is gated behind an experimental flag.

```
agents: {
  defaults: {
    memorySearch: {
      experimental: { sessionMemory: true },
      sources: ["memory", "sessions"]
    }
  }
}
```

Notes:

- Session indexing is **opt-in** (off by default).

Session updates are debounced and **indexed asynchronously** once they cross delta thresholds (best-effort).

`memory_search` never blocks on indexing; results can be slightly stale until background sync finishes.

Results still include snippets only; `memory_get` remains limited to memory files.

Session indexing is isolated per agent (only that agent's session logs are indexed).

Session logs live on disk (`~/.openclaw/agents/<agentId>/sessions/*.jsonl`). Any process/user with filesystem access can read them, so treat disk access as the trust boundary. For stricter isolation, run agents under separate OS users or hosts.

Delta thresholds (defaults shown):

```
agents: {
  defaults: {
    memorySearch: {
      sync: {
        sessions: {
          deltaBytes: 100000,   // ~100 KB
          deltaMessages: 50     // JSONL lines
        }
      }
    }
  }
}
```

SQLite vector acceleration (sqlite-vec)

When the sqlite-vec extension is available, OpenClaw stores embeddings in a SQLite virtual table (`vec0`) and performs vector distance queries

in the database. This keeps search fast without loading every embedding into JS.

Configuration (optional):

```
agents: {
  defaults: {
    memorySearch: {
      store: {
        vector: {
          enabled: true,
          extensionPath: "/path/to/sqlite-vec"
        }
      }
    }
  }
}
```

Notes:

- `enabled` defaults to true; when disabled, search falls back to in-process cosine similarity over stored embeddings.
- If the sqlite-vec extension is missing or fails to load, OpenClaw logs the error and continues with the JS fallback (no vector table).
- `extensionPath` overrides the bundled sqlite-vec path (useful for custom builds or non-standard install locations).

Local embedding auto-download

- Default local embedding model: `hf:ggml-org/embeddinggemma-300m-qat-q8_0-GGUF/embeddinggemma-300m-qat-Q8_0.gguf` (~0.6 GB).
- When `memorySearch.provider = "local"`, `node-llama-cpp` resolves `modelPath`; if the GGUF is missing it **auto-downloads** to the cache

(or `local.modelCacheDir` if set), then loads it. Downloads resume on retry.

Native build requirement: run `pnpm approve-builds`, pick `node-llama-cpp`, then `pnpm rebuild node-llama-cpp`.

Fallback: if local setup fails and `memorySearch.fallback = "openai"`, we automatically switch to remote embeddings (`openai/text-embedding-3-small` unless overridden) and record the reason.

Custom OpenAI-compatible endpoint example

```
agents: {
  defaults: {
    memorySearch: {
      provider: "openai",
      model: "text-embedding-3-small",
      remote: {
        baseUrl: "https://api.example.com/v1/",
        apiKey: "YOUR_REMOTE_API_KEY",
        headers: {
          "X-Organization": "org-id",
          "X-Project": "project-id"
        }
      }
    }
  }
}
```

Notes:

- `remote.*` takes precedence over `models.providers.openai.*`.
- `remote.headers` merge with OpenAI headers; remote wins on key conflicts. Omit `remote.headers` to use the OpenAI defaults.

‹ Session Tools Compaction ›