04 — Compaction

🕒 state of the art 2026-07 · revised 2026-07-25 · 📖 ~13 min read

Learning objectives

By the end of this chapter, you should be able to:

1. **Explain** why compaction exists and which constraints it balances (fidelity × cost × cache);
2. **Compare** the four layers of the aggressiveness ladder and **justify** their ordering;
3. **Analyze** a real harness's compaction implementation and locate its choices on the ladder (Appendix A as the answer key);
4. **Implement** truncation with edge preservation and summarization with a preserved tail (step 5 of harness-zero);
5. **Evaluate** when a compaction has failed (loss of a decision, of file state or of the goal) — and **anticipate** what changes when the provider compacts for you.

The problem

Every agent conversation grows until it no longer fits in the model's context window. Compaction is the set of strategies for continuing to work when that happens — without losing what matters. It is the dimension where the evaluated harnesses converge the most: all of them arrived, independently, at the same layered architecture.

The constraints in tension:

- **Fidelity**: the summary cannot lose decisions, file state or the task's goal.
- **Cost**: summarizing via LLM (Large Language Model) is expensive; truncating is cheap but destructive.
- **Cache**: compacting invalidates the cached prefix — it should happen as little as possible and at controlled moments.

Scientific foundations

- **The window is not uniform** — *Lost in the Middle* (arXiv 2307.03172) showed that models use the beginning and end of the context best and degrade in the middle. It is the empirical basis for two of the ladder's practices: preserving the recent *tail* intact and truncating outputs while keeping start+end.
- **Context as virtual memory** — *MemGPT* (arXiv 2310.08560) framed the operating-systems analogy: the window is "RAM", external storage is "disk", and the harness pages between them. Recent work takes the analogy to its literal limit (*demand paging*, arXiv 2603.09023).
- **Compacting is a budget decision** — *ContextBudget* (arXiv 2604.01664) treats context management as explicit allocation per content type — what products implement as thresholds and budgets.

(Full bibliography and validation status: `livro/bibliografia.md`.)

Industry sources

- **Compaction — Claude Platform Docs** (Anthropic, official): compaction has reached **the API level** (beta `compact-2026-01-12`) — the provider summarizes automatically upon hitting the configured threshold and returns a "compaction block". It is vendor confirmation of this chapter's central trend (see The state of the art).
- **Claude Code operating practices** (CometAPI, okhlopkov, hyperdev): the practitioners' convergent recommendation is the same one the harnesses encode — **what needs to survive compaction should not live in the conversation**: conventions go to the context file (CLAUDE.md/AGENTS.md, reinjected every session) and progress state goes to files the agent rereads after the compact. Compaction defines, by exclusion, what deserves persistence.
- **See also**: the living collection Awesome Harness Engineering — Context Delivery & Compaction gathers more resources for this dimension (patterns, articles and implementations), curated by problem.

The state of the art

The consolidated pattern: the aggressiveness ladder

Harnesses apply the strategies as a ladder, from cheapest to most expensive — this is the industry consensus, verified in every benchmark round:

1. **Truncate tool outputs at the source** — limit lines/bytes before they enter the history, preserving start and end (*Lost in the Middle* justifies the edges). The modern refinement: **do not discard** — move the full content to referenceable files (opencode) or keep the raw output outside the model's view but visible in the UI (Goose).
2. **Prune / microcompact** — erase the *content* of old tool results (the model rarely rereads a `cat` from 30 turns ago), keeping the record of the call. Newer intermediate layers: *tool distillation* and *output masking* (gemini-cli).
3. **LLM summarization (full compact)** — summarize the old portion while preserving an intact tail (typically 20–30% or a 2k–20k token budget). The state of the art has three refinements: a **structured summary** with mandatory fields (user intent, pending tasks, code state — Goose and software-agent-sdk), a **cheap auxiliary model** for the summary (Hermes), and a **memory flush before compacting** — saving durable notes before losing the context (OpenClaw).
4. **Automatic trigger + reactive path** — a trigger by window percentage (50–90% depending on the project) and, covering the failure case, compaction **reactive** to the API's "prompt too long" error (OpenHarness, OpenClaw).

The two modern frontiers

1. Auditable compaction (tombstones). The most advanced implementation measured in the benchmark (the software-agent-sdk's condenser) does not mutate the history: the log is append-only and forgetting is an *event* (`Condensation`) — a tombstone, as in Cassandra/Kafka. The model's view is derived by applying the tombstones; nothing is lost to auditing, and formal invariants (tool_call/result pairing, batch atomicity) are

testable code, with the *hard/soft trigger* distinction: if compacting now would violate an invariant, the soft trigger waits for the next turn; the hard one forces an explicit reset. A related refinement: the **effectiveness circuit-breaker** (IronClaw) — comparing the post-compaction estimate against a baseline and detecting compactions that are not working.

2. Compaction is migrating to the provider. (And caching is becoming a protocol contract too: the MCP 2026-07-28 spec added `ttlMs`/`cacheScope` to `tools/list` responses — the protocol taking over what used to be harness heuristics.) Two independent signals in the same year: the Codex CLI implements **remote compaction v2** (the backend compacts) and Anthropic launched **compaction in the API itself** (docs, beta `compact-2026-01-12`). It is the expiration clause in motion — but with an interesting inversion: instead of the component disappearing when the model improves, it **changes owner** (from the harness to the platform). What remains for the harness when the provider compacts: deciding *what to protect* (skills, task state, memory files), *when to trust* (auditing the summary's quality — OpenClaw's `safeguard` mode anticipated this) and the reactive path for providers that do not offer the service.

> **Addendum (2026-07-31, full text verified): the third way — compaction learned in training.** The preprint CompactionRL (Tsinghua/Z.AI, 06 Jul 2026) proposes the migration's next step: training the model via RL **with compaction inside the loop** — "CompactionRL incorporates compaction into rollout collection, and reconstructs the agent context from a summary once context budget is exhausted" (§1); summarization becomes "a learned part of the model rather than an inference-time heuristic", with a **task-level reward**. The numbers (Table 2, always against the same model *already using inference-time compaction*): GLM-4.5-Air **59.8 → 66.8** on SWE-bench Verified (+7.0) and +3.1 on Terminal-Bench 2.0; GLM-4.7-Flash **+5.5 and +6.8**. And the experiment's protocol is exactly this chapter's ladder — a threshold by remaining budget, a structured summary from a fixed prompt, a **preserved tail of k=2 steps** — that is, the paper validates the triad and changes the *training*, not the architecture. Three consequences: (1) the harness remains the owner of the *when*, but the *how to summarize* is starting to migrate into the weights — harness ↔ model mismatch becomes a new risk; (2) the declared limitation is revealing: "its gains do not consistently transfer to single-window evaluation when compaction is disabled. This indicates a train–test mismatch" — trained compaction creates *coupling* (with compaction turned off, the trained GLM-4.7-Flash actually gets worse, 47.5 → 43.7), the strongest argument so far for an explicit *compaction contract* between harness and model; (3) in the other direction, Table 1 hands power back to the harness: with the executor fixed, **swapping only the summarizer** moves SWE-Verified from 49.0 to 55.5 (+6.5) — "compaction is a performance-critical decision process rather than a passive preprocessing step", and a better dedicated summarizer **beats self-summarization**: choosing who summarizes is a harness decision, and a big one.

The third frontier: compaction stops being involuntary (round ext-4, 2026-08)

The launch of **Prime Agent** (Prime Intellect, Aug/2026) came with a charge aimed straight at this chapter: *"fixed tool-calling schemas and context compaction force the model to work around its own scaffolding instead*

of leveraging it." Reading the code (full evaluation) shows that **the charge is rhetoric and the code says something else** — and the gap between the two is the finding.

Compaction was **neither removed nor weakened**. Prime Agent is built on Pi, and the 1,398 lines of `core/compaction/` are there intact — safe cutting, split turns, cumulative files, reactive overflow recovery — and even improved, with custom instructions and `tokensBefore` recomputation. What changed is **who is in charge**: `compact.run()` and `compact.status()` became callable **by the agent itself** (`skills/compact/`), with a handler that **schedules instead of executing** — executing on the spot would abort the very REPL cell that requested the compaction — and that runs even with automatic compaction turned off, under twelve test cases. Add to that the session's JSONL path injected into the system prompt: the full history, **including previous compactions**, remains programmatically reachable.

The caveat to record in this chapter is therefore precise: **compaction stops being an involuntary event of the harness and becomes one mechanism among others, available to the agent**. It also gains a new role — it became a distillation trigger, with `autoRefine.compact: true` by default: every compaction is an opportunity for the agent to extract learning from what is about to be summarized.

What the aggressiveness ladder did not anticipate is not its own obsolescence but the **inversion of control**: until now, the harness compacts *the* agent; here, the agent compacts *itself*. The gap the reading found is telling — the announcement mentions a subagent acting as a garbage collector for the REPL, and **there is nothing of the sort in the code** (searching for `garbage`/`prune`/`evict` in `src/core`, `skills` and `prime-agent-runtime` returns nothing). Context-as-a-variable solves access to the past; it does **not** solve the growth of the namespace it creates.

EXECUTIVE SUMMARY

Convergence on the ladder is nearly total — the pattern is consolidated, and a new harness that does not implement it needs to justify itself. The remaining differences are fidelity refinements (structuring the summary, auditing its quality, never discarding) and the big open question is one of *market architecture*: how much of the ladder survives in the harness when the platform offers compaction as a service — a question the addendum above sharpens: after migrating to the provider, compaction is starting to migrate **into the weights**. **What to steal** today: tombstones over an append-only log; pre-compaction memory-flush; a structured summary with task IDs preserved; the effectiveness circuit-breaker; and — new in ext-4 — **agent-callable compaction that schedules instead of executing**, plus the session log path in the system prompt, which puts the past back within the model's reach without spending window.

> **Editorial caveat (2026-08-06).** This Executive summary was confronted in round ext-4 and **upheld**, with the qualification in the previous section: the ladder is still the pattern, but *authority* over when to apply it has begun migrating to the agent. If the pattern repeats in other harnesses, the synthesis changes — and this paragraph will be rewritten, not patched.

Hands-on — harness-zero, step 5

In step 5 of the project (`harness-zero/`), you implement the ladder in your own harness, in this order: (1) tool output truncation with start+end preservation; (2) pruning of old tool results beyond a budget; (3) LLM summarization of the history's head, preserving the tail; (4) automatic triggering by an estimated-token threshold — with a **visible indicator in the chat** when compaction happens (the reader's observation window). Completion exercise: the prune function's skeleton comes ready; you write the selection of what to protect.

Check your understanding

1. Why truncate tool outputs **before** summarizing via LLM, and not the other way around? (Cost and destructiveness — if needed, reread the ladder.)
2. A harness summarized the history and the agent, on the next turn, rewrote a file that was already correct. What information did the compaction probably lose, and which state-of-the-art mechanism prevents it? (Hint: structured summary with `CODE_STATE`/`CHANGES`.)
3. Your provider now offers compaction in the API. Which of the ladder's responsibilities do you **transfer** and which do you **keep** in the harness? (Connect with "the two modern frontiers".)

Appendix A — How each repository handles compaction

> Per-harness evidence, with paths — supplementary material (online version), expanded with each benchmark round. The chapter's base source: the code of these repositories.

opencode (round 1) — three mechanisms + managed files

`packages/opencode/src/session/compaction.ts` (+ `overflow.ts`, `summary.ts`): (a) automatic summarization on overflow with a **dedicated `compaction` agent**, tail under budget (`preserveRecentBudget`, 2k–8k tokens), a new Context Epoch and optional auto-continue; (b) back-to-front **prune** marking tool outputs beyond 40k tokens as `compacted` (`PRUNE_PROTECT`), protecting skills; (c) truncation at the source (`tool/truncate.ts`) preserving start+end and moving the full text to "Managed Tool Output Files".

gemini-cli (round 1) — compression + distillation + masking

`packages/core/src/context/chatCompressionService.ts`: fires at 50% of the limit (`DEFAULT_COMPRESSION_TOKEN_THRESHOLD = 0.5`), preserves the last 30% (`COMPRESSION_PRESERVE_THRESHOLD`), its own budget for function responses (50k) and saving of truncated outputs. Extra layers: `toolDistillationService.ts` and `toolOutputMaskingService.ts`. Manual `/compress`, `ChatCompressed` event, `PreCompressTrigger` hooks.

OpenHarness (round 1) — the faithful translation of Claude Code

`src/openharness/services/compact/__init__.py` (1,725 lines; docstring: "Faithfully translated from Claude Code's compaction system"): **microcompact** (clears `COMPACTABLE_TOOLS`), **full compact** (LLM summary), **auto-compact** (threshold) and compaction **reactive** to "prompt too long" (`_is_prompt_too_long_error`). `PRE_COMPACT`/`POST_COMPACT` hooks; preserves task state and channel logs.

Codex CLI (round 2) — local + remote v1/v2

`core/src/compact.rs`, `compact_remote_v2.rs`, `compact_token_budget.rs`: auto-compact at ~90% of the window; three strategies — local (`SUMMARIZATION_PROMPT`) and **remote v1/v2** (the backend compacts, via `ResponsesStreamRequest::RemoteCompactionV2`, with its own retry); versioned windows with prefill tracking; controlled pre/mid-turn injection; `TruncationPolicy` for outputs.

Goose (round 2) — structured summary + middle-out

`crates/goose/src/context_mgmt/mod.rs`: threshold at 0.8 of the window; `StructuredSummary` (user_intent, files, pending_tasks, current_work); if summarization overflows, **progressive "middle-out" removal** of tool-responses (0→100%); **incremental summarization of tool-call/response pairs** in batches of 10 protecting the last N; visibility metadata preserves the raw output in the UI; respects `provider.manages_own_context()`.

OpenClaw (round 2) — safeguard + memory flush

`src/context-engine/` + `docs/concepts/compaction.md`: automatic by threshold and reactive (recognizes dozens of overflow error strings from multiple providers), split preserving tool-call/result pairs; `safeguard` mode with **summary quality auditing**; **silent memory flush before compacting**; `keepRecentTokens` 20k; pluggable compaction providers; the compaction (semantic) × pruning (in-memory trim) distinction.

Hermes (round 2) — pluggable engine + auxiliary model

`agent/context_engine.py` (interface `should_compact`/`compress`/`prune`) + `trajectory_compressor.py` (~1.6k lines): summarization of old tool-responses via a **cheap auxiliary model** (default Gemini Flash, up to 50 concurrent requests); manual `/compress`; `/usage` and `/insights` expose the window.

IronClaw (round 2) — pure policy + circuit-breaker

`crates/ironclaw_agent_loop/src/strategies/compaction.rs` (+ `active_task_compaction.rs`): the strategy is **pure policy** (returns Skip or the `drop_through_seq` limit; mutation only in the host); `PromptContextTokenBudget` with `preserve_tail_tokens`; an **effectiveness circuit-breaker** (compares the post-compaction estimate against

`CompactionEffectivenessBaseline`); a variant that preserves the active task; the host refuses to compact through non-user messages.

software-agent-sdk (frameworks round) — tombstones + testable invariants ⭐

`openhands-sdk/openhands/sdk/context/condenser/`: forgetting via **tombstones** (`Condensation` event) over an append-only log; triggering for three reasons (REQUEST/TOKENS/EVENTS) with **hard/soft** (`condensation_requirement`) and `hard_context_reset()` for the pathological case; `keep_first` + recursive re-summarization of summaries; a structured prompt (`summarizing_prompt.j2`: USER_CONTEXT, TASK_TRACKING with exact IDs, CODE_STATE, TESTS, CHANGES); invariants in `context/view/properties/` (tool_call_matching, batch_atomicity...) **tested against real LLMs** (`tests/integration/tests/c01..c05`); `pipeline_condenser` for composition.

Aider (round 2) — classic summarization done well

`aider/history.py` (`ChatSummary`): keeps the tail (~half the budget), summarizes the head via LLM with a split after an `assistant` message, **recursive** up to depth 3, with a fallback model list.

n8n (round 2) — the absence that confirms the category

No compaction in the loop (the memory sub-nodes' `contextWindowLength` + `maxTokensFromMemory` only) — consistent with short, event-triggered executions; it is the "embedded harness" category's ceiling for long tasks.

LangGraph / OpenAI Agents SDK / CrewAI (frameworks round) — the dividing line

LangGraph: **zero native support** (a docstring suggesting `pre_model_hook`); Agents SDK (Software Development Kit): only `OpenAIResponsesCompactionSession` as an optional session; CrewAI: nothing. Compaction is the dimension that most separates "framework" from "ready-made harness".