04 — Compactação

🕒 estado da arte 2026-07 · revisão 2026-07-25 · 📖 ~13 min de leitura

Objetivos de aprendizagem

Ao final deste capítulo, você deve ser capaz de:

1. **Explicar** por que a compactação existe e quais restrições ela equilibra (fidelidade × custo × cache);
2. **Comparar** as quatro camadas da escada de agressividade e **justificar** a ordem entre elas;
3. **Analisar** a implementação de compactação de um harness real e localizar suas escolhas na escada (Apêndice A como gabarito);
4. **Implementar** truncamento com preservação de bordas e sumarização com tail preservado (etapa 5 do harness-zero);
5. **Avaliar** quando uma compactação falhou (perda de decisão, de estado de arquivo ou de objetivo) — e **antecipar** o que muda quando o provedor compacta por você.

O problema

Toda conversa de agente cresce até não caber na janela de contexto do modelo. A compactação é o conjunto de estratégias para continuar trabalhando quando isso acontece — sem perder o que importa. É a dimensão onde os harnesses avaliados mais convergem: todos chegaram, independentemente, à mesma arquitetura em camadas.

As restrições em tensão:

- **Fidelidade**: o resumo não pode perder decisões, estado de arquivos ou o objetivo da tarefa.
- **Custo**: sumarizar via LLM (Large Language Model) é caro; truncar é barato mas destrutivo.
- **Cache**: compactar invalida o prefixo cacheado — deve acontecer o mínimo possível e em momentos controlados.

Fundamentos científicos

- **A janela não é uniforme** — *Lost in the Middle* (arXiv 2307.03172) mostrou que modelos usam melhor o início e o fim do contexto e degradam no meio. É a base empírica de duas práticas da escada: preservar o *tail* recente intacto e truncar outputs mantendo início+fim.
- **Contexto como memória virtual** — *MemGPT* (arXiv 2310.08560) formulou a analogia com sistemas operacionais: a janela é a "RAM", o armazenamento externo é o "disco", e o harness pagina entre eles. Trabalhos recentes levam a analogia ao limite literal (*demand paging*, arXiv 2603.09023).
- **Compactar é decisão de orçamento** — *ContextBudget* (arXiv 2604.01664) trata a gestão de contexto como alocação explícita por tipo de conteúdo — o que os produtos implementam como limiares e budgets.

(Bibliografia completa e status de validação: `livro/bibliografia.md`.)

Fontes da indústria

- **Compaction — Claude Platform Docs** (Anthropic, oficial): a compactação chegou **ao nível da API** (beta `compact-2026-01-12`) — o provedor sumariza automaticamente ao atingir o limiar configurado e devolve um "compaction block". É a confirmação de vendor da tendência central deste capítulo (ver Estado da arte).
- **Práticas de operação do Claude Code** (CometAPI, okhlopkov, hyperdev): a recomendação convergente dos praticantes é a mesma que os harnesses codificam — **o que precisa sobreviver à compactação não deve morar na conversa**: convenções vão para o arquivo de contexto (CLAUDE.md/AGENTS.md, reinjetado a cada sessão) e estado de progresso vai para arquivos que o agente relê depois do compact. A compactação define, por exclusão, o que merece persistência.
- **Consulte também**: a coleção viva Awesome Harness Engineering — Context Delivery & Compaction reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

O estado da arte

O padrão consolidado: a escada de agressividade

Os harnesses aplicam as estratégias em escada, da mais barata à mais cara — este é o consenso da indústria, verificado em todas as rodadas do benchmark:

1. **Truncar saídas de tools na origem** — limitar linhas/bytes antes de entrar no histórico, preservando início e fim (*Lost in the Middle* justifica as bordas). O refinamento moderno: **não descartar** — mover o conteúdo integral para arquivos referenciáveis (opencode) ou manter o bruto fora da view do modelo mas visível na UI (Goose).
2. **Prune / microcompact** — apagar o *conteúdo* de resultados de tools antigas (o modelo raramente relê um `cat` de 30 turnos atrás), mantendo o registro da chamada. Camadas intermediárias mais novas: *tool distillation* e *output masking* (gemini-cli).
3. **Sumarização via LLM (full compact)** — resumir a porção antiga preservando um tail intacto (tipicamente 20–30% ou um orçamento de 2k–20k tokens). O estado da arte tem três refinamentos: **resumo estruturado** com campos obrigatórios (intenção do usuário, tarefas pendentes, estado de código — Goose e software-agent-sdk), **modelo auxiliar barato** para o resumo (Hermes), e **flush de memória antes de compactar** — salvar notas duráveis antes de perder o contexto (OpenClaw).
4. **Disparo automático + caminho reativo** — gatilho por percentual da janela (50–90% conforme o projeto) e, cobrindo a falha, compactação **reativa** ao erro "prompt too long" da API (OpenHarness, OpenClaw).

As duas fronteiras modernas

1. Compactação auditável (tombstones). A implementação mais avançada medida no benchmark (condenser do software-agent-sdk) não muta o histórico: o log é append-only e o esquecimento é um *evento* (`Condensation`) — um tombstone, como em Cassandra/Kafka. A view do modelo é derivada aplicando os tombstones; nada se perde para auditoria, e invariantes formais (pareamento tool_call/result, atomicidade de batch) são **código testável**, com a distinção *hard/soft trigger*: se compactar agora violaria uma invariante, o gatilho suave espera o próximo turno; o duro força um reset explícito. Refinamento correlato: o **circuit-breaker de efetividade** (IronClaw) — comparar a estimativa pós-compactação contra baseline e detectar compactações que não estão funcionando.

2. A compactação está migrando para o provedor. (E o cache também vira contrato de protocolo: a spec MCP 2026-07-28 adicionou `ttlMs`/`cacheScope` às respostas de `tools/list` — o protocolo assumindo o que antes era heurística do harness.) Dois sinais independentes no mesmo ano: o Codex CLI implementa **compactação remota v2** (o backend compacta) e a Anthropic lançou **compaction na própria API** (docs, beta `compact-2026-01-12`). É a cláusula de expiração em movimento — mas com uma inversão interessante: em vez de o componente desaparecer quando o modelo melhora, ele **muda de dono** (do harness para a plataforma). O que resta ao harness quando o provedor compacta: decidir *o que proteger* (skills, estado de tarefa, arquivos de memória), *quando confiar* (auditoria de qualidade do resumo — o modo `safeguard` do OpenClaw antecipou isso) e o caminho reativo para provedores que não oferecem o serviço.

> **Adendo (2026-07-31, texto integral verificado): a terceira via — compactação aprendida no treino.** O preprint CompactionRL (Tsinghua/Z.AI, 06-jul-2026) propõe o passo seguinte da migração: treinar o modelo por RL **com a compactação dentro do loop** — "CompactionRL incorporates compaction into rollout collection, and reconstructs the agent context from a summary once context budget is exhausted" (§1); sumarização vira "a learned part of the model rather than an inference-time heuristic", com recompensa de nível de **tarefa**. Os números (Tabela 2, sempre contra o mesmo modelo *já com compactação de inferência*): GLM-4.5-Air **59,8 → 66,8** no SWE-bench Verified (+7,0) e +3,1 no Terminal-Bench 2.0; GLM-4.7-Flash **+5,5 e +6,8**. E o protocolo do experimento é exatamente a escada deste capítulo — limiar por orçamento restante, sumário estruturado por prompt fixo, **cauda preservada de k=2 passos** — ou seja, o paper valida a tríade e muda o *treino*, não a arquitetura. Três consequências: (1) o harness continua dono do *quando*, mas o *como sumarizar* começa a migrar para os pesos — descasamento harness ↔ modelo vira risco novo; (2) a limitação declarada é reveladora: "its gains do not consistently transfer to single-window evaluation when compaction is disabled. This indicates a train–test mismatch" — compactação treinada cria *acoplamento* (com compactação desligada, o GLM-4.7-Flash treinado chega a **piorar**, 47,5 → 43,7), o argumento mais forte até agora para o *contrato de compactação* explícito entre harness e modelo; (3) na contramão, a Tabela 1 devolve poder ao harness: fixado o executor, **trocar só o sumarizador** move o SWE-Verified de 49,0 a 55,5 (+6,5) — "compaction is a performance-critical decision process rather than a passive preprocessing step", e um sumarizador dedicado melhor **supera o auto-sumário**: escolher quem resume é decisão de harness, e das grandes.

A terceira fronteira: a compactação deixa de ser involuntária (rodada ext-4, 2026-08)

O lançamento do **Prime Agent** (Prime Intellect, ago/2026) veio com uma acusação direta a este capítulo: *"fixed tool-calling schemas and context compaction force the model to work around its own scaffolding instead of leveraging it"*. A leitura do código (avaliação completa) mostra que **a acusação é retórica e o código diz outra coisa** — e a diferença entre as duas é o achado.

A compactação **não foi eliminada nem enfraquecida**. O Prime Agent é construído sobre o Pi, e as 1.398 linhas de `core/compaction/` estão lá intactas — corte seguro, split turns, arquivos cumulativos, recuperação reativa de overflow —, ainda melhoradas com instruções customizadas e recálculo de `tokensBefore`. O que mudou é **quem manda**: `compact.run()` e `compact.status()` viraram chamáveis **pelo próprio agente** (`skills/compact/`), com um handler que **agenda em vez de executar** — executar na hora abortaria a célula do REPL que pediu a compactação — e que roda mesmo com a compactação automática desligada, sob doze casos de teste. Some-se a isso que o caminho do JSONL da sessão é injetado no system prompt: o histórico completo, **inclusive as compactações anteriores**, continua acessível por programa.

A ressalva a registrar neste capítulo é, portanto, precisa: **a compactação deixa de ser um evento involuntário do harness e passa a ser um mecanismo entre outros, disponível ao agente**. Ela ganha ainda um papel novo — virou gatilho de destilação, com `autoRefine.compact: true` por padrão: toda compactação é uma oportunidade de o agente extrair aprendizado do que está prestes a ser resumido.

O que a escada de agressividade não previa não é a sua obsolescência, mas a **inversão do controle**: até aqui, o harness compacta *no* agente; aqui, o agente compacta *a si mesmo*. A lacuna que a leitura encontrou é reveladora — o anúncio menciona um subagente atuando como coletor de lixo do REPL, e **não existe nada disso no código** (busca por `garbage`/`prune`/`evict` em `src/core`, `skills` e `prime-agent-runtime` não retorna nada). O contexto-como-variável resolve o acesso ao passado; **não** resolve o crescimento do namespace que ele mesmo cria.

LEITURA EXECUTIVA

A convergência na escada é quase total — o padrão está consolidado e um harness novo que não a implemente precisa justificar. As diferenças que restam são refinamentos de fidelidade (estruturar o resumo, auditar sua qualidade, nunca descartar) e a grande questão em aberto é de *arquitetura de mercado*: quanto da escada sobrevive no harness quando a plataforma oferece compaction como serviço — questão que o adendo acima agudiza: depois de migrar para o provedor, a compactação começa a migrar **para os pesos**. **O que roubar** hoje: tombstones sobre log append-only; memory-flush pré-compactação; resumo estruturado com IDs de tarefa preservados; circuit-breaker de efetividade; e — novo na ext-4 — **compactação chamável pelo agente que agenda em vez de executar**, mais o caminho do log da sessão no system prompt, que devolve o passado ao alcance do modelo sem gastar janela.

> **Ressalva de edição (2026-08-06).** Esta Leitura executiva foi confrontada na rodada ext-4 e **mantida**, com a qualificação da seção anterior: a escada continua sendo o padrão, mas a *autoridade*

> sobre quando aplicá-la começou a migrar para o agente. Se o padrão se repetir em outros harnesses, a síntese muda — e este parágrafo será reescrito, não emendado.

Mão na massa — harness-zero, etapa 5

Na etapa 5 do projeto (`harness-zero/`), você implementa a escada no seu próprio harness, nesta ordem: (1) truncamento de output de tool com preservação de início+fim; (2) prune de resultados de tools antigas além de um orçamento; (3) sumarização via LLM da cabeça do histórico, preservando o tail; (4) disparo automático por limiar de tokens estimados — com um **indicador visível no chat** quando a compactação acontece (a janela de observação do leitor). Exercício de completude: o esqueleto da função de prune vem pronto; você escreve a seleção do que proteger.

Verificação

1. Por que truncar outputs de tools **antes** de sumarizar via LLM, e não o contrário? (Custo e destrutividade — se precisar, releia a escada.)
2. Um harness sumarizou o histórico e o agente, no turno seguinte, reescreveu um arquivo que já estava correto. Qual informação a compactação provavelmente perdeu, e qual mecanismo do estado da arte previne isso? (Dica: resumo estruturado com `CODE_STATE`/`CHANGES`.)
3. Seu provedor passou a oferecer compaction na API. Quais responsabilidades da escada você **transfere** e quais **mantém** no harness? (Conecte com "as duas fronteiras modernas".)

Apêndice A — Como cada repositório trata a compactação

> Evidência por harness, com paths — material de complementação (versão online), expandido a cada rodada do benchmark. Fonte-base do capítulo: o código destes repositórios.

opencode (rodada 1) — três mecanismos + arquivos gerenciados

`packages/opencode/src/session/compaction.ts` (+ `overflow.ts`, `summary.ts`): (a) sumarização automática em overflow com **agente dedicado `compaction`**, tail sob orçamento (`preserveRecentBudget`, 2k–8k tokens), novo Context Epoch e auto-continue opcional; (b) **prune** de trás para frente marcando `compacted` saídas de tools além de 40k tokens (`PRUNE_PROTECT`), protegendo skills; (c) truncamento na origem (`tool/truncate.ts`) preservando início+fim e movendo o texto completo para "Managed Tool Output Files".

gemini-cli (rodada 1) — compressão + destilação + mascaramento

`packages/core/src/context/chatCompressionService.ts`: dispara a 50% do limite (`DEFAULT_COMPRESSION_TOKEN_THRESHOLD = 0.5`), preserva os últimos 30% (`COMPRESSION_PRESERVE_THRESHOLD`), orçamento próprio para function responses (50k) e salvamento de outputs truncados. Camadas extras: `toolDistillationService.ts` e `toolOutputMaskingService.ts`. `/compress` manual, evento `ChatCompressed`, hooks `PreCompressTrigger`.

OpenHarness (rodada 1) — a tradução fiel do Claude Code

`src/openharness/services/compact/__init__.py` (1.725 linhas; docstring: "Faithfully translated from Claude Code's compaction system"): **microcompact** (limpa `COMPACTABLE_TOOLS`), **full compact** (resumo LLM), **auto-compact** (limiar) e compactação **reativa** a "prompt too long" (`_is_prompt_too_long_error`). Hooks `PRE_COMPACT`/`POST_COMPACT`; preserva task state e logs de canal.

Codex CLI (rodada 2) — local + remota v1/v2

`core/src/compact.rs`, `compact_remote_v2.rs`, `compact_token_budget.rs`: auto-compact a ~90% da janela; três estratégias — local (`SUMMARIZATION_PROMPT`) e **remota v1/v2** (o backend compacta, via `ResponsesStreamRequest::RemoteCompactionV2`, com retry próprio); janelas versionadas com prefill tracking; injeção controlada pré/mid-turn; `TruncationPolicy` para outputs.

Goose (rodada 2) — resumo estruturado + middle-out

`crates/goose/src/context_mgmt/mod.rs`: limiar 0.8 da janela; `StructuredSummary` (user_intent, files, pending_tasks, current_work); se a sumarização estoura, **remoção progressiva "middle-out"** de tool-responses (0→100%); **sumarização incremental de pares tool-call/response** em batches de 10 protegendo os N últimos; metadados de visibilidade preservam o bruto na UI; respeita `provider.manages_own_context()`.

OpenClaw (rodada 2) — safeguard + memory flush

`src/context-engine/` + `docs/concepts/compaction.md`: auto por limiar e reativa (reconhece dezenas de strings de erro de overflow de múltiplos provedores), split preservando pares tool-call/result; modo `safeguard` com **auditoria de qualidade do resumo**; **memory flush silencioso antes de compactar**; `keepRecentTokens` 20k; providers de compactação plugáveis; distinção compaction (semântica) × pruning (trim in-memory).

Hermes (rodada 2) — engine plugável + modelo auxiliar

`agent/context_engine.py` (interface `should_compact`/`compress`/`prune`) + `trajectory_compressor.py` (~1.6k linhas): sumarização de tool-responses antigas via **modelo auxiliar barato** (default Gemini Flash, até 50 requisições concorrentes); `/compress` manual; `/usage` e `/insights` expõem a janela.

IronClaw (rodada 2) — política pura + circuit-breaker

`crates/ironclaw_agent_loop/src/strategies/compaction.rs` (+ `active_task_compaction.rs`): a estratégia é **política pura** (retorna Skip ou o limite `drop_through_seq`; mutação só no host); `PromptContextTokenBudget` com `preserve_tail_tokens`; **circuit-breaker de efetividade** (compara estimativa pós-compactação contra `CompactionEffectivenessBaseline`); variante que preserva a tarefa ativa; o host rejeita compactar através de mensagens não-usuário.

software-agent-sdk (rodada frameworks) — tombstones + invariantes testáveis ⭐

`openhands-sdk/openhands/sdk/context/condenser/`: esquecimento por **tombstones** (`Condensation` event) sobre log append-only; disparo por três razões (REQUEST/TOKENS/EVENTS) com **hard/soft** (`condensation_requirement`) e `hard_context_reset()` para o caso patológico; `keep_first` + re-sumarização recursiva de sumários; prompt estruturado (`summarizing_prompt.j2`: USER_CONTEXT, TASK_TRACKING com IDs exatos, CODE_STATE, TESTS, CHANGES); invariantes em `context/view/properties/` (tool_call_matching, batch_atomicity...) **testadas contra LLMs reais** (`tests/integration/tests/c01..c05`); `pipeline_condenser` para compor.

Aider (rodada 2) — sumarização clássica bem-feita

`aider/history.py` (`ChatSummary`): mantém a cauda (~metade do orçamento), sumariza a cabeça via LLM com split após mensagem `assistant`, **recursivo** até profundidade 3, com lista de modelos com fallback.

n8n (rodada 2) — a ausência que confirma a categoria

Sem compactação no loop (`contextWindowLength` dos memory sub-nodes + `maxTokensFromMemory` apenas) — coerente com execuções curtas acionadas por evento; é o teto da categoria "harness embutido" para tarefas longas.

LangGraph / OpenAI Agents SDK / CrewAI (rodada frameworks) — a linha divisória

LangGraph: **zero suporte nativo** (uma docstring sugerindo `pre_model_hook`); Agents SDK (Software Development Kit): apenas `OpenAIResponsesCompactionSession` como session opcional; CrewAI: nada. A compactação é a dimensão que mais separa "framework" de "harness pronto".