02 — Loop do Agente

🕒 estado da arte 2026-07 · revisão 2026-08-01 · 📖 ~9 min de leitura

Objetivos de aprendizagem

1. **Explicar** o ciclo prompt → decisão → ferramenta → resultado e o critério de parada estrutural;
2. **Comparar** os dois contratos de terminação da indústria (ausência de tool call × `output_type` satisfeito);
3. **Implementar** um loop com freios (turnos, orçamento) e trace observável (etapa 1 do harness-zero);
4. **Projetar** retry em duas camadas (dentro do passo × replay do loop) e reconhecer o que exige idempotência;
5. **Avaliar** a durabilidade de um loop real (o que sobrevive a um crash?).

O problema

O loop é o coração do harness: envia contexto ao modelo, recebe uma decisão (texto e/ou **tool calls** — pedidos estruturados de ação: "execute tal ferramenta com tais argumentos"), executa, realimenta e repete — até que alguém decida parar.

Um turno completo, em câmera lenta. Você digita: "o teste `test_login` falhou, corrija". O que o loop faz:

1. Monta o contexto (regras do projeto + sua mensagem) e **chama o modelo**;
2. O modelo não responde com texto — responde com uma tool call: `executar_shell("pytest test_login")`;
3. O harness **executa de verdade** e devolve a saída (o traceback do erro) ao modelo, como se fosse uma nova mensagem;
4. O modelo agora *viu* o erro e emite outra tool call: `editar_arquivo("auth.py", …)`;
5. O harness executa (talvez pedindo sua aprovação — cap. 07) e devolve o resultado;
6. Nova chamada ao modelo, que pede o teste de novo; desta vez passa;
7. O modelo responde **só com texto** ("corrigido: era o cookie expirado") — e é *isso* que encerra o turno: **sem tool call, o loop para**.

Sete passos, três chamadas ao modelo, duas execuções reais. Todo o resto deste capítulo são as perguntas difíceis escondidas nesse ciclo: quem decide parar (e se o modelo nunca parar?), como os erros voltam, o que acontece quando o processo morre no passo 5, quanto isso pode custar. As perguntas de projeto: quem decide parar? como os resultados e erros voltam? o que acontece quando dá errado? o loop sobrevive a um reinício?

Fundamentos científicos

- **ReAct** (arXiv 2210.03629) é o paper seminal: intercalar raciocínio e ação com feedback do ambiente supera raciocínio puro — é a justificativa científica de o loop existir.
- O survey de **frameworks de raciocínio agêntico** (arXiv 2508.17692) sistematiza as variantes do ciclo (ReAct, plan-and-act, reflexão), útil como mapa do território.
- A fronteira treinada: surveys de **agentic search com RL** (arXiv 2510.16724) mostram o loop deixando de ser só orquestração e virando objeto de treinamento — quando o modelo é treinado *no* loop, parte do harness migra para os pesos.

(Bibliografia completa: `livro/bibliografia.md`.)

Fontes da indústria

- **How the agent loop works** (Claude Agent SDK (Software Development Kit), docs): o loop canônico em 5 estágios; "turno" termina **quando o modelo responde sem tool calls**; e o detalhe mais moderno — terminação como **estado tipado** (`success`, `error_max_turns`, `error_max_budget_usd`...): sucesso e esgotamento de limite são caminhos de código distintos e obrigatórios. Inclui `max_budget_usd` **propagado a subagentes** e a compactação como evento observável do loop (`compact_boundary`).
- **Loop engineering** (Claude blog): o vendor batiza a disciplina e dá a taxonomia por eixos (como dispara, como para, que primitivo usa) — com a regra de projeto citável: *se você não consegue escrever a verificação, o loop não está pronto para existir*.
- **Building Effective AI Agents** (Anthropic): a distinção fundadora workflow × agente e o padrão **evaluator-optimizer** — parada semântica (qualidade atingida) com um juiz separado.
- **Running agents** (OpenAI Agents SDK): o contrato alternativo — parada quando o agente produz o **`output_type` declarado** (validável), com `MaxTurnsExceeded` tipado.
- **LoopAgent** (Google ADK): só duas formas de parar — `max_iterations` ou um sub-agente juiz emitindo `escalate=True` — o loop burro separado do juiz endereçável.
- **Durable AI Loops** (Restate) e Inngest: o loop como **workflow de longa duração** — cada passo journalado, falha = replay do último passo concluído; retry vira duas categorias (backoff dentro do passo × replay do loop), com idempotência obrigatória em tools mutantes.
- **Consulte também**: a coleção viva Awesome Harness Engineering — Agent Loop reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

O estado da arte

1. Parada virou contrato multi-eixo

O critério estrutural (sem tool call) continua universal, mas sozinho é ingênuo. O contrato moderno combina: limite de turnos; **teto de orçamento em dinheiro** (a novidade real de 2025–26, já propagando a subagentes); *subtype* tipado de terminação; e, no contrato alternativo do Agents SDK, **parada por tipo de saída** — que transforma "acabou?" em validação verificável. Sobre isso, dois refinamentos medidos no benchmark: o **next-speaker check** do gemini-cli (uma inferência barata decide se o modelo continua sozinho) e o veto de término

— hooks `Stop` que podem **recusar o fim do turno** e reinjetar feedback (software-agent-sdk; o verify-on-stop do Hermes é o mesmo princípio como nudge).

2. Anti-runaway: do contador ao detector

Todo loop maduro tem `MAX_TURNS`; os melhores têm detecção de repetição — `LoopDetectionService` (gemini-cli), `RepetitionInspector` (Goose), stuck detector com estados `stalled/stuck` (software-agent-sdk, OpenClaw). A técnica de campo (hash de `tool+args` em janela deslizante) circula entre praticantes mas não tem doc de vendor — citável como prática, não como norma.

3. Durabilidade virou propriedade do loop, não da infra

O consenso 2026: journaling por passo + replay. No benchmark: rollouts jsonl recuperáveis (Codex), inbox durável de prompts com eventos replayáveis por cursor (opencode V2), event log append-only com retomada por diretório (software-agent-sdk) e — o desenho mais radical — o executor que **retorna apenas referências duráveis** e nunca muta estado, com um applier validando evidência antes de aplicar (IronClaw). Corolário para tools: idempotência deixa de ser virtude e vira requisito.

4. O loop não é o perímetro

A lição arquitetural mais importante da rodada 2 (IronClaw): *"the loop is intentionally not the security perimeter"* — o loop pede efeitos por portas; quem decide é o kernel. Mesmo fora do contexto de segurança, a separação política (quando parar/confirmar/desistir — `Conversation.run()`) × mecânica (view → LLM → dispatch — `Agent.step()`) do software-agent-sdk é o corte limpo que permite trocar o motor mantendo o loop.

> **LEITURA EXECUTIVA**
>
> O que está mais moderno: terminação tipada com orçamento em dólares; juiz separado e endereçável (evaluator-optimizer/escalate) em vez de heurística no prompt; durabilidade por journaling/replay; e a separação política×mecânica. **O que roubar:** `ResultMessage.subtype` tipado; budget propagado a subagentes; hooks Stop com poder de veto; o LoopExit por referências duráveis.

Mão na massa — harness-zero, etapa 1

A etapa 1 (`harness-zero/etapas/01-loop/`) implementa o núcleo em ~30 linhas: parada estrutural, `MAX_TURNS` como freio, erros de tool voltando **como texto** para o modelo decidir, e trace das ações visível no chat. Exercícios de extensão: (a) adicione um subtype de terminação (`success` × `max_turns`); (b) adicione um orçamento de custo estimado e o terceiro subtype.

Verificação

1. Por que "o modelo respondeu sem tool calls" é um bom default de parada — e por que é insuficiente sozinho? (Contrato multi-eixo.)
2. Seu agente chamou a mesma tool com os mesmos argumentos 5 vezes seguidas. Liste duas defesas de naturezas diferentes. (Detector de repetição × teto de orçamento.)
3. O processo morreu no meio do turno 7. O que o seu loop precisa ter persistido para retomar sem repetir efeitos colaterais? (Journaling + idempotência.)

Apêndice A — Como cada repositório trata o loop

> Evidência por harness, com paths — complementação online, expandida a cada rodada.

opencode (rodada 1)

`packages/opencode/src/session/processor.ts`: resposta consumida como `Stream` do Effect (`Stream.tap(handleEvent)` → `takeUntil(needsCompaction)` → `runDrain`); veredito explícito `continue | stop | compact`; retry por provedor (`SessionRetry.policy`); V2 (`CONTEXT.md`): inbox durável e eventos replayáveis com cursores.

gemini-cli (rodada 1)

`packages/core/src/core/client.ts` (`MAX_TURNS=100`) + `turn.ts`; **next-speaker check** (`utils/nextSpeakerChecker.ts`: mini-prompt `{reasoning, next_speaker}` re-invoca o stream se `model`); `LoopDetectionService`; separação core/cli limpa.

OpenHarness (rodada 1)

`src/openharness/engine/query.py` (`run_query`): `while` async até `max_turns` ou sem tool-uses; **paralelismo quando todas as tools do turno são read-only** (`asyncio.gather`); PreToolUse → permissão → execução → PostToolUse por chamada; retry com backoff e cost tracking.

Codex CLI (rodada 2)

`core/src/session/turn.rs` (`run_turn`, 2.581 linhas) sobre `SessionTask` trait (Regular/Review/Compact/UserShell); streaming SSE (Server-Sent Events) **e WebSocket com fallback WS→HTTPS**; `CancellationToken` hierárquico; cada turno persistido em rollout jsonl; sem detector de repetição explícito (mitigado por budgets).

Goose (rodada 2)

`crates/goose/src/agents/agent.rs` (`reply` → `BoxStream<AgentEvent>`): dois níveis de retry (transiente por provedor + `RetryManager` de recipe com `SuccessCheck` que reseta a conversa); `DEFAULT_MAX_TURNS=1000`; `RepetitionInspector`; `MAX_EMPTY_TURN_RETRIES=3`.

OpenClaw (rodada 2)

`src/system-agent/agent-turn.ts` + `gateway/agent-*.ts`: runs serializados por *session lane* com write-lock file-based inter-processo; três streams de eventos (lifecycle/assistant/tool); watchdogs `stalled/stuck`; hooks duplos (Gateway + plugins).

Hermes (rodada 2)

`agent/conversation_loop.py` (~6.5k linhas) com fases separadas (turn_context/tool_executor/turn_finalizer); `iteration_budget`; **interrupt-and-redirect** (`/steer` drenado pré-API e pós-tool); nudges para respostas vazias; reparo de alternância de papéis; **verify-on-stop nudge**.

IronClaw (rodada 2) ⭐

`crates/ironclaw_agent_loop`: pipeline de estágios selados (input → prompt → model → capability → gate/checkpoint → stop), cada estágio uma strategy privada; o executor devolve um `LoopExit` contendo **apenas referências duráveis** — nunca muta estado — e o `LoopExitApplier` valida evidência host-owned antes de aplicar (tese explícita da arquitetura: *"the loop is intentionally not the security perimeter"*). Estado resumível por checkpoints; máquina de estados Queued→Running→Blocked→Completed com leases/heartbeats; "one active run per canonical thread".

Aider (rodada 2)

`aider/coders/base_coder.py`: não é um loop de tool-calling — é REPL de chat + edição direta. O único mecanismo iterativo é a **reflexão** (`reflected_message`, máx. 3): arquivos pedidos fora do chat, erros de linter ou testes falhando disparam nova rodada, sempre com confirmação humana. Auto-correção reativa por design, não autonomia.

OpenHands/Canvas (rodada 2)

`app_server/event/`: o event-stream persiste cada `Event` como JSON por conversa (paginação, filtros, export de trajetória) — mas o loop ação/observação roda no `openhands-agent-server` (SDK); o app consome eventos, não os gera. O núcleo está no software-agent-sdk (abaixo).

ohmo (rodada 2.5)

Loop herdado do `QueryEngine` do OpenHarness; o que é próprio: **pool multi-sessão** (`ohmo/gateway/runtime.py`: um `RuntimeBundle` por `session_key`, recriado quando o cwd muda) e **interrupção real por mensagem nova** (`bridge.py`: cada mensagem é uma asyncio.Task; mensagem nova da mesma sessão cancela a anterior) — poucos concorrentes cancelam corretamente.

n8n (rodada 2)

A V2 usa o `AgentExecutor` clássico do LangChain (`maxIterations` default 10); a **V3** mantém o `createToolCallingAgent` só para *decidir* — as tool calls viram `EngineRequest` devolvidos ao **motor de workflow do n8n**, que agenda os nós-tool e reentra com `EngineResponse`

(`ToolsAgent/V3/helpers/runAgent.ts`). O n8n reinternalizou o loop de execução: decisão do framework, execução do engine.

Frameworks (rodada frameworks) — quatro respostas à mesma pergunta

LangGraph: a primitiva real é **Pregel/BSP** (supersteps + channels + reducers), com retry/cache/timeout por nó — e o agente pronto (`create_react_agent`) formalmente deprecado (migrou para `langchain.agents`). **OpenAI Agents SDK (Software Development Kit)**: loop explícito em `run.py` (output_type termina · handoff troca agente · `max_turns` com handlers), sobre um `AgentRunner` substituível. **CrewAI**: executor **100% próprio, zero LangChain** (`crew_agent_executor.py`), com dispatch duplo — tool-calling nativo ou fallback ReAct com `json_repair`. **software-agent-sdk**: `LocalConversation.run()` (política: parar, confirmar, desistir) separado de `Agent.step()` (mecânica stateless view → LLM → dispatch), event log append-only com `View` derivada e hooks `Stop` com poder de **veto** sobre o término.