02 — The Agent Loop

🕒 state of the art 2026-07 · revised 2026-08-01 · 📖 ~9 min read

Learning objectives

1. **Explain** the prompt → decision → tool → result cycle and the structural stopping criterion;
2. **Compare** the industry's two termination contracts (absence of tool calls × satisfied `output_type`);
3. **Implement** a loop with brakes (turns, budget) and an observable trace (step 1 of harness-zero);
4. **Design** two-layer retry (inside the step × loop replay) and recognize what requires idempotency;
5. **Evaluate** the durability of a real loop (what survives a crash?).

The problem

The loop is the heart of the harness: it sends context to the model, receives a decision (text and/or **tool calls** — structured requests for action: "run this tool with these arguments"), executes, feeds back and repeats — until someone decides to stop.

One full turn, in slow motion. You type: "the `test_login` test failed, fix it". What the loop does:

1. Assembles the context (project rules + your message) and **calls the model**;
2. The model does not answer with text — it answers with a tool call: `run_shell("pytest test_login")`;
3. The harness **actually executes it** and returns the output (the error traceback) to the model, as if it were a new message;
4. The model has now *seen* the error and emits another tool call: `edit_file("auth.py", …)`;
5. The harness executes (perhaps asking for your approval — ch. 07) and returns the result;
6. A new call to the model, which asks for the test again; this time it passes;
7. The model answers **with text only** ("fixed: it was the expired cookie") — and *that* is what ends the turn: **no tool call, the loop stops**.

Seven steps, three model calls, two real executions. Everything else in this chapter is the hard questions hiding in that cycle: who decides to stop (and what if the model never stops?), how errors come back, what happens when the process dies at step 5, how much this can cost. The design questions: who decides to stop? how do results and errors come back? what happens when things go wrong? does the loop survive a restart?

Scientific foundations

- **ReAct** (arXiv 2210.03629) is the seminal paper: interleaving reasoning and action with environment feedback beats pure reasoning — it is the scientific justification for the loop's existence.

- The survey of **agentic reasoning frameworks** (arXiv 2508.17692) systematizes the cycle's variants (ReAct, plan-and-act, reflection), useful as a map of the territory.
- The trained frontier: surveys of **agentic search with RL** (arXiv 2510.16724) show the loop ceasing to be mere orchestration and becoming a training target — when the model is trained *in* the loop, part of the harness migrates into the weights.

(Full bibliography: `livro/bibliografia.md`.)

Industry sources

- **How the agent loop works** (Claude Agent SDK (Software Development Kit), docs): the canonical 5-stage loop; a "turn" ends **when the model responds without tool calls**; and the most modern detail — termination as a **typed state** (`success`, `error_max_turns`, `error_max_budget_usd`...): success and limit exhaustion are distinct, mandatory code paths. Includes `max_budget_usd` **propagated to subagents** and compaction as an observable loop event (`compact_boundary`).
- **Loop engineering** (Claude blog): the vendor names the discipline and gives the taxonomy by axes (how it fires, how it stops, which primitive it uses) — with the quotable design rule: *if you cannot write the verification, the loop is not ready to exist.*
- **Building Effective AI Agents** (Anthropic): the founding workflow × agent distinction and the **evaluator-optimizer** pattern — semantic stopping (quality reached) with a separate judge.
- **Running agents** (OpenAI Agents SDK): the alternative contract — stop when the agent produces the declared **`output_type`** (validatable), with a typed `MaxTurnsExceeded`.
- **LoopAgent** (Google ADK): only two ways to stop — `max_iterations` or a judge sub-agent emitting `escalate=True` — the dumb loop separated from the addressable judge.
- **Durable AI Loops** (Restate) and Inngest: the loop as a **long-running workflow** — each step journaled, failure = replay from the last completed step; retry becomes two categories (backoff inside the step × loop replay), with idempotency mandatory for mutating tools.
- **See also**: the living collection Awesome Harness Engineering — Agent Loop gathers more resources for this dimension (patterns, articles and implementations), curated by problem.

The state of the art

1. Stopping became a multi-axis contract

The structural criterion (no tool call) remains universal, but on its own it is naive. The modern contract combines: a turn limit; a **budget ceiling in money** (the real novelty of 2025–26, already propagating to subagents); a typed termination *subtype*; and, in the Agents SDK's alternative contract, **stopping by output type** — which turns "are we done?" into verifiable validation. On top of this, two refinements measured in the benchmark: gemini-cli's **next-speaker check** (a cheap inference decides whether the model continues on its own) and the termination veto — `Stop` hooks that can **refuse the end of the turn** and reinject feedback (software-agent-sdk; Hermes's verify-on-stop is the same principle as a nudge).

2. Anti-runaway: from counter to detector

Every mature loop has `MAX_TURNS`; the best have repetition detection — `LoopDetectionService` (gemini-cli), `RepetitionInspector` (Goose), a stuck detector with `stalled/stuck` states (software-agent-sdk, OpenClaw). The field technique (hashing `tool+args` over a sliding window) circulates among practitioners but has no vendor doc — citable as practice, not as norm.

3. Durability became a property of the loop, not of the infra

The 2026 consensus: per-step journaling + replay. In the benchmark: recoverable jsonl rollouts (Codex), a durable prompt inbox with cursor-replayable events (opencode V2), an append-only event log with directory-based resumption (software-agent-sdk) and — the most radical design — the executor that **returns only durable references** and never mutates state, with an applier validating evidence before applying (IronClaw). The corollary for tools: idempotency stops being a virtue and becomes a requirement.

4. The loop is not the perimeter

The most important architectural lesson of round 2 (IronClaw): *"the loop is intentionally not the security perimeter"* — the loop requests effects through ports; the kernel decides. Even outside the security context, the software-agent-sdk's separation of policy (when to stop/confirm/give up — `Conversation.run()`) × mechanics (view → LLM → dispatch — `Agent.step()`) is the clean cut that lets you swap the engine while keeping the loop.

> **EXECUTIVE SUMMARY**
>
> What is most modern: typed termination with a dollar budget; a separate, addressable judge (evaluator-optimizer/escalate) instead of a heuristic in the prompt; durability via journaling/replay; and the policy×mechanics separation. **What to steal:** the typed `ResultMessage.subtype`; budget propagated to subagents; Stop hooks with veto power; the LoopExit via durable references.

Hands-on — harness-zero, step 1

Step 1 (`harness-zero/etapas/01-loop/`) implements the core in ~30 lines: structural stopping, `MAX_TURNS` as a brake, tool errors returning **as text** for the model to decide, and a trace of actions visible in the chat. Extension exercises: (a) add a termination subtype (`success` × `max_turns`); (b) add an estimated-cost budget and the third subtype.

Check your understanding

1. Why is "the model responded without tool calls" a good stopping default — and why is it insufficient on its own? (Multi-axis contract.)

2. Your agent called the same tool with the same arguments 5 times in a row. List two defenses of different natures. (Repetition detector × budget ceiling.)
3. The process died in the middle of turn 7. What must your loop have persisted in order to resume without repeating side effects? (Journaling + idempotency.)

Appendix A — How each repository handles the loop

> Per-harness evidence, with paths — online supplement, expanded with each round.

opencode (round 1)

`packages/opencode/src/session/processor.ts`: response consumed as an Effect `Stream` (`Stream.tap(handleEvent)` → `takeUntil(needsCompaction)` → `runDrain`); explicit `continue | stop | compact` verdict; per-provider retry (`SessionRetry.policy`); V2 (`CONTEXT.md`): durable inbox and cursor-replayable events.

gemini-cli (round 1)

`packages/core/src/core/client.ts` (`MAX_TURNS=100`) + `turn.ts`; **next-speaker check** (`utils/nextSpeakerChecker.ts`: mini-prompt `{reasoning, next_speaker}` re-invokes the stream if `model`); `LoopDetectionService`; clean core/cli separation.

OpenHarness (round 1)

`src/openharness/engine/query.py` (`run_query`): async `while` until `max_turns` or no tool-uses; **parallelism when all tools in the turn are read-only** (`asyncio.gather`); PreToolUse → permission → execution → PostToolUse per call; retry with backoff and cost tracking.

Codex CLI (round 2)

`core/src/session/turn.rs` (`run_turn`, 2,581 lines) on top of the `SessionTask` trait (Regular/Review/Compact/UserShell); SSE (Server-Sent Events) streaming **and WebSocket with WS→HTTPS fallback**; hierarchical `CancellationToken`; each turn persisted in jsonl rollouts; no explicit repetition detector (mitigated by budgets).

Goose (round 2)

`crates/goose/src/agents/agent.rs` (`reply` → `BoxStream<AgentEvent>`): two levels of retry (transient per provider + a recipe-level `RetryManager` with a `SuccessCheck` that resets the conversation); `DEFAULT_MAX_TURNS=1000`; `RepetitionInspector`; `MAX_EMPTY_TURN_RETRIES=3`.

OpenClaw (round 2)

`src/system-agent/agent-turn.ts` + `gateway/agent-*.ts`: runs serialized per *session lane* with an inter-process file-based write-lock; three event streams (lifecycle/assistant/tool); `stalled/stuck` watchdogs; dual hooks (Gateway + plugins).

Hermes (round 2)

`agent/conversation_loop.py` (~6.5k lines) with separate phases (turn_context/tool_executor/turn_finalizer); `iteration_budget`; **interrupt-and-redirect** (`/steer` drained pre-API and post-tool); nudges for empty responses; role-alternation repair; **verify-on-stop nudge**.

IronClaw (round 2) ⭐

`crates/ironclaw_agent_loop`: a pipeline of sealed stages (input → prompt → model → capability → gate/checkpoint → stop), each stage a private strategy; the executor returns a `LoopExit` containing **only durable references** — it never mutates state — and the `LoopExitApplier` validates host-owned evidence before applying (the architecture's explicit thesis: *"the loop is intentionally not the security perimeter"*). Resumable state via checkpoints; a Queued→Running→Blocked→Completed state machine with leases/heartbeats; "one active run per canonical thread".

Aider (round 2)

`aider/coders/base_coder.py`: not a tool-calling loop — it is a chat REPL + direct editing. The only iterative mechanism is **reflection** (`reflected_message`, max 3): files requested outside the chat, linter errors or failing tests trigger a new round, always with human confirmation. Reactive self-correction by design, not autonomy.

OpenHands/Canvas (round 2)

`app_server/event/`: the event-stream persists each `Event` as JSON per conversation (pagination, filters, trajectory export) — but the action/observation loop runs in the `openhands-agent-server` (SDK); the app consumes events, it does not generate them. The core is in the software-agent-sdk (below).

ohmo (round 2.5)

Loop inherited from OpenHarness's `QueryEngine`; what is its own: a **multi-session pool** (`ohmo/gateway/runtime.py`: one `RuntimeBundle` per `session_key`, recreated when the cwd changes) and **real interruption by new message** (`bridge.py`: each message is an asyncio.Task; a new message from the same session cancels the previous one) — few competitors cancel correctly.

n8n (round 2)

V2 uses LangChain's classic `AgentExecutor` (`maxIterations` default 10); **V3** keeps `createToolCallingAgent` only to *decide* — tool calls become `EngineRequest`s handed back to the **n8n workflow engine**, which schedules the tool nodes and re-enters with `EngineResponse`

(`ToolsAgent/V3/helpers/runAgent.ts`). n8n re-internalized the execution loop: decision by the framework, execution by the engine.

Frameworks (frameworks round) — four answers to the same question

LangGraph: the real primitive is **Pregel/BSP** (supersteps + channels + reducers), with per-node retry/cache/timeout — and the ready-made agent (`create_react_agent`) formally deprecated (migrated to `langchain.agents`). **OpenAI Agents SDK (Software Development Kit)**: explicit loop in `run.py` (output_type terminates · handoff switches agent · `max_turns` with handlers), on top of a swappable `AgentRunner`. **CrewAI**: a **100% in-house executor, zero LangChain** (`crew_agent_executor.py`), with dual dispatch — native tool-calling or a ReAct fallback with `json_repair`. **software-agent-sdk**: `LocalConversation.run()` (policy: stop, confirm, give up) separated from `Agent.step()` (stateless mechanics view → LLM → dispatch), an append-only event log with a derived `View`, and `Stop` hooks with **veto** power over termination.