09 — Planejamento

🕒 estado da arte 2026-07 · revisão 2026-07-26 · 📖 ~12 min de leitura

Objetivos de aprendizagem

Ao final deste capítulo, você deve ser capaz de:

1. **Distinguir** os três instrumentos de planejamento — plan mode, todo list e decomposição — e o requisito de cada um;
2. **Explicar** por que plan mode se implementa como um caso do sistema de permissões (imposto, não pedido);
3. **Comparar** ReAct (intercalar razão e ação) com plan-then-execute e decidir quando cada um serve;
4. **Avaliar** a estratificação tático × durável (plano da tarefa × objetivo da sessão) e a decomposição com dependências;
5. **Implementar** plan mode imposto por permissões no harness-zero (etapa 8).

O problema

Modelos tendem a agir precipitadamente: editam antes de entender, "resolvem" antes de mapear o problema. Os artefatos de planejamento forçam uma fase de leitura e desenho antes da fase de escrita — e dão ao humano um ponto de aprovação barato (revisar um plano custa menos que revisar um diff).

Três instrumentos distintos, frequentemente confundidos:

1. **Plan mode** — um *estado* do harness em que escrever é proibido; o agente só pesquisa e propõe.
2. **Todo list** — memória de trabalho da tarefa em andamento: o que falta, o que está feito.
3. **Decomposição** — quebrar trabalho grande em subtarefas rastreáveis, possivelmente com dependências.

Fundamentos científicos

A literatura de planejamento explica *por que* esses instrumentos existem — e adverte contra confiar no plano do modelo.

- **Intercalar vence planejar-tudo-antes (quando o ambiente é imprevisível)** — ReAct, arXiv 2210.03629 (ICLR '23) intercala traço de raciocínio e ações de tool no mesmo loop: cada observação revisa o próximo pensamento, então o agente se recupera de surpresas em vez de executar um plano velho. Decisão: carregue raciocínio e observações num único transcript alternado.
- **Planejar-antes ajuda (quando o escopo é conhecido)** — Plan-and-Solve, arXiv 2305.04091 faz o modelo emitir um plano explícito antes de resolver, suprimindo passos faltantes. Os dois não se contradizem: são regimes distintos — o plano explícito para tarefas de escopo conhecido, a intercalação para ambientes incertos.

- **Decompor só quando preciso** — ADaPT, arXiv 2311.05772 decompõe **recursivamente e apenas quando o executor falha** uma subtarefa, adaptando a profundidade à dificuldade e à capacidade do modelo. Decisão: tente executar primeiro, decomponha na falha — evita o over-planning que a maioria dos harnesses (sabiamente) não impõe.
- **Isolar o contexto por subtarefa** — Beyond Entangled Planning, arXiv 2601.07577 (2026) decompõe num **DAG de sub-objetivos** e dá contexto *escopado* a cada um, para que erros locais e replanejamento não poluam um histórico monolítico — reporta até −82% de tokens. Ponte direta com subagentes (cap. 10).
- **Não confie no plano do modelo — externalize** — PlanBench, arXiv 2206.10498 e TravelPlanner, arXiv 2402.01622 mostram que modelos crus falham em geração de plano e perdem o fio de múltiplas restrições (GPT-4 ~0,6% no TravelPlanner). Decisão: externalize o rastreio de restrições num artefato (plano/todo), em vez de confiar que o modelo segura tudo no contexto. É *a* justificativa da todo list.
- **A taxonomia como checklist** — survey de planejamento, arXiv 2402.02716 organiza os componentes em cinco vias (decomposição de tarefa · seleção de plano · módulo externo · reflexão · memória); PlanGenLLMs, arXiv 2502.11221 dá seis critérios (completude, executabilidade, otimalidade, representação, generalização, eficiência) e PLANET, arXiv 2504.14773 organiza benchmarks por categoria.

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

Fontes da indústria

- **Plan mode é uma camada de permissão** — Choose a permission mode (Claude Code): plan mode remove escrita/execução pela *sessão inteira*; o agente lê e explora, mas toda mutação fica retida até você sair (Shift+Tab cicla Normal → Plan → Auto-accept; `/plan`; `--permission-mode plan` para CI). Decisão: o planejamento é garantido **revogando as tools de mutação**, não pedindo ao modelo que "planeje primeiro". É a confirmação oficial da descoberta da rodada 1.
- **Explorar → Planejar → Codar → Commitar** — Best practices (Claude Code): as fases de exploração e planejamento são "as mais baratas em tokens e as mais valiosas em resultado". Decisão: separar exploração de execução impede estruturalmente resolver o problema errado antes de entender o código.
- **Todo como artefato rastreado por máquina** — Todo tracking (Agent SDK): o `TodoWrite` cria checklists com três estados (pending/in_progress/completed) atualizados em tempo real. Decisão: externalizar o plano num artefato estruturado dá ao agente uma âncora de memória de trabalho e ao usuário visibilidade de progresso — e a evolução para um sistema de *tasks* com dependências e persistência torna o plano infraestrutura durável, não scrollback.
- **Pensar entre as ações** — The "think" tool adiciona um passo de raciocínio *no meio* do uso de tools (depois que o resultado chega); o extended thinking expõe blocos de raciocínio com `budget_tokens` e, nos modelos 4, **interleaved thinking** (pensar → chamar tool → pensar sobre o resultado → chamar de novo). Decisão: aloque orçamento explícito para passos de planejamento e deixe o raciocínio intercalar com as tools — planejar não é um prefixo único, é contínuo. *(anthropic.com retorna 403 pelo proxy; confirmado por espelhos independentes.)*

- **Spec-driven: o spec é o plano durável** — GitHub Spec Kit formaliza `specify` (o quê/porquê) → `plan` (arquitetura) → `tasks` (lista acionável) → `implement`, com gates de aprovação entre estágios; a Kiro gera `requirements.md` (EARS `WHEN…THE SYSTEM SHALL…`), `design.md` e `tasks.md`, e **deriva um grafo de dependências** que executa tarefas independentes em ondas concorrentes. Decisão: o plano vira fonte de verdade persistida e re-consumida a cada fase — é exatamente o método com que **este livro é escrito** (ver a constituição do projeto).
- **Planejar é uma função de orquestração — e a tensão sobre paralelizar** — o sistema multi-agente da Anthropic faz o *lead* analisar a query, **gravar o plano em memória** e só então spawnar workers com specs isolados (planejamento como papel dedicado). A Cognition ("Don't Build Multi-Agents") contrapõe: o Devin centraliza o planejamento num contexto contínuo, porque planejar *é* gestão de contexto — paralelizar workers vira "telefone sem fio" de decisões implícitas conflitantes. Decisão: decompor-e-paralelizar é um gate de custo/benefício, não um default (liga ao cap. 10). *(cognition.com 403 pelo proxy; confirmado por espelhos.)*
- **Consulte também**: a coleção viva Awesome Harness Engineering — Planning & Task Decomposition reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

O estado da arte

1. Plan mode = modo de permissão (agora padrão oficial)

A descoberta da primeira rodada — os harnesses implementam plan mode **como um caso do sistema de permissões** (cap. 07), não como subsistema próprio — deixou de ser observação e virou padrão documentado: a doc oficial do Claude Code descreve plan mode exatamente assim (remove mutação pela sessão). Entrar em plan mode = trocar para um ruleset que nega escritas; sair = restaurar, com aprovação explícita. O padrão maduro combina três garantias: read-only **imposto** (não pedido), plano como **artefato persistido** (não só texto na conversa) e **aprovação explícita** antes de executar.

2. ReAct virou o default; o plano explícito recuou para o trabalho longo

O sinal mais claro do livro vivo veio do n8n: seu **Plan-and-Execute Agent foi depreciado** (só existe na V1 legada, ao lado do ReAct), e a V2/V3 convergiram para o Tools Agent puro — planejamento implícito no modelo. Isso instancia a tese científica: conforme os modelos planejam melhor inline, a intercalação (ReAct) vence o plan-then-execute como default, e o **plano explícito se concentra onde ainda paga**: trabalho longo, humano no loop, e decomposição de tarefas grandes. Não é que planejar morreu — é que o planejamento barato migrou para dentro do loop.

3. A todo list é rastreio de restrições externalizado

O que PlanBench e TravelPlanner provam (modelos perdem o fio de múltiplas restrições) é o que a todo list resolve: um checklist com estados (Codex `update_plan`, `TodoWrite`, `todo` do Hermes/Goose, `TODO.md` do OpenHarness) tira as restrições da cabeça do modelo e as põe num artefato. A evolução moderna é dar

dependências e persistência a esse artefato — o tracker em grafo do gemini-cli, o grafo de dependências da Kiro, o DAG do "Beyond Entangled Planning".

4. Tático × durável — a contribuição dos agentes pessoais

Os harnesses de código têm um plano *da tarefa*; falta-lhes o *durável*. O **OpenClaw** preenche isso com quatro camadas: `update_plan` (tático, um passo `in_progress` por vez), **Goals** (um objetivo durável por sessão, com token budget e estados, injetado por turno e visível na UI), **Task Flow** (orquestração durável com steps e estado JSON) e standing orders (políticas persistentes). Essa estratificação tática × durável é a fronteira que a categoria de agentes pessoais trouxe à disciplina.

5. Planejamento é a dimensão mais fraca — e isso é um dado, não um acaso

Em todas as rodadas, planejamento foi a nota mais baixa da indústria (Codex 2, Goose 2, Aider 2, Hermes 2, OpenHands 1, n8n 1, IronClaw 2; só o gemini-cli e o OpenClaw chegam a 3). A leitura do livro vivo (registro de expiração, "plan mode imposto", 🔵 aberta): a prótese existe porque os modelos agem precipitadamente, e ela expira quando os modelos planejarem sob risco espontaneamente — o que ainda não aconteceu. A fraqueza persistente da dimensão *é* a evidência de que a prótese ainda é necessária.

> **LEITURA EXECUTIVA**
>
> O que está mais moderno: plan mode como camada de permissão (padrão oficial); ReAct/interleaved thinking como default, com plano explícito reservado a trabalho longo; todo/checklist como rastreio de restrições externalizado, evoluindo para grafos de dependência; e a estratificação tático × durável. **O que roubar:** imponha o read-only pela permissão, não pelo prompt; externalize o plano num artefato persistido com estados; dê orçamento de thinking aos passos de planejamento; e decomponha-e-paralelize só quando a largura da tarefa paga o custo.

Mão na massa — harness-zero, etapa 8

A etapa 8 (`harness-zero/etapas/08-plan/`) adiciona plan mode ao harness-zero **reusando** a `PermissionPolicy` da etapa 6: entrar em plan mode seta um modo que a política traduz em "toda tool de escrita é negada"; o agente só lê e propõe; sair pede aprovação e restaura o modo. É a demonstração concreta da tese do capítulo — plan mode não é um subsistema, é uma configuração do domínio de permissões que já existe. Exercício de completude: o `propor_plano` já persiste o artefato (`PLAN.md`); você adiciona a exigência de que a saída do plan mode só aconteça com um `PLAN.md` aprovado — o gate entre planejar e executar.

Verificação

1. Por que faz sentido implementar plan mode como um modo do sistema de permissões, em vez de um subsistema dedicado? (Reusa um mecanismo existente e ganha de graça a garantia de que o read-only é

imposto, não sugerido ao modelo.)

2. Seu agente opera num ambiente imprevisível (respostas de API mudam o próximo passo). Você planeja tudo antes ou intercala razão e ação? Por quê? (Intercala — ReAct: cada observação revisa o próximo pensamento; um plano fixo fica velho.)
3. Um benchmark mostra seu agente perdendo o fio de 8 restrições numa tarefa. Que instrumento de planejamento ataca isso, e por quê? (Todo list / checklist — externaliza o rastreio de restrições para fora do contexto do modelo.)

Apêndice A — Como cada repositório trata o planejamento

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

opencode (rodada 1) — plan como agente

Plan mode é um **agente built-in `plan`** com ruleset read-only (nega edições, pede confirmação para bash) — planejar é trocar de agente, não só de modo. A tool `plan_exit` (`tool/plan.ts`) fecha o ciclo: pergunta se aprova, **escreve o plano em arquivo** e transiciona para o agente `build`. Prompts dedicados (`prompt/plan-mode.txt`, `plan-reminder-anthropic.txt` — lembretes por família de modelo). Todos por sessão via `todowrite` (`session/todo.ts`).

gemini-cli (rodada 1) — plan com gatekeeping e decomposição

`ApprovalMode.PLAN` (`policy/types.ts`) com `enter-plan-mode`/`exit-plan-mode`: estado read-only cujo prompt lista as tools disponíveis, e `getApprovedPlanPath()` **gatekeepa a execução**. Todos via `WriteTodosTool`. O instrumento que os outros não têm: o **tracker** opcional (`trackerTools.ts`) — tarefas com dependências (`tracker_add_dependency`) e grafo (`tracker_visualize`). Plan mode tem eval comportamental própria (`evals/plan_mode.eval.ts`).

OpenHarness (rodada 1) — a versão mínima e correta

`EnterPlanModeTool` seta `settings.permission.mode = PLAN` (bloqueia todas as escritas); `ExitPlanModeTool` restaura — a implementação mais direta da equivalência plan-mode-é-permissão. Todos em `TODO.md` via `TodoWriteTool` (persistente, legível). Skill bundled `plan`; decomposição pesada no subsistema autopilot (fila de `RepoTaskCard`).

OpenClaw (rodada 2) ⭐ — tático × durável em quatro camadas

`update_plan` (plano multi-step, um `in_progress` por vez), **Goals** (objetivo durável por sessão com token budget e estados, injetado por turno e visível na UI), **Task Flow** (orquestração durável com steps e estado JSON) e **standing orders** (políticas persistentes). A estratificação tática × durável que os harnesses de código não têm.

Codex CLI (rodada 2) — checklist estruturado

Tool `update_plan` (checklist visível na TUI (Terminal User Interface)) + `ReviewTask`. Sem plan mode de duas fases com aprovação de plano antes da execução — a economia de "planejar antes" fica no checklist, não num gate de permissão.

Aider (rodada 2) — plan-then-edit por modos de coder

`/ask` (discute sem editar), `/architect` (raciocina o "como" antes de delegar) e `/context` (usa o repo-map para convergir nos arquivos). Plan-then-edit leve, sem artefato de plano persistido nem todo list; o split `architect→editor` executa o plano com um segundo modelo.

Goose (rodada 2) — recipes declarativos

Recipes (YAML/JSON com instructions, parâmetros tipados, `response.json_schema`, retry) + extensão `todo` + `final_output_tool`. Planejamento declarativo/reusável, sem plan mode de duas fases.

Hermes (rodada 2) — todo + Kanban

Tool `todo` + orçamento de iterações + **sistema Kanban** para coordenação multi-agente com specs. Planejamento acoplado ao loop, sem planner formal separado.

n8n (rodada 2) — o planejamento que recuou

O **Plan-and-Execute Agent** existe mas é **legado** (só na V1, junto com ReAct/Conversational); V2/V3 convergiram para o Tools Agent puro. Planejamento ficou implícito no modelo (+ `ToolThink` opcional). O caso mais claro de plano explícito perdendo para intercalação.

IronClaw (rodada 2) — planejamento temporal, não decomposição

Sem decomposição de tarefas de primeira classe; o "planner" do loop é composição de strategies. A força está no planejamento *temporal* (agendamento, leases/heartbeats — dim. suplementar 14).

OpenHands / ohmo (rodada 2)

OpenHands: aba planner na UI e ganchos, sem subsistema de decomposição de 1ª classe neste repo (nota 1; o núcleo migrou para o SDK). ohmo: plan mode/todos herdados que assumem TUI — sem superfície de aprovação de plano num canal de chat.

Frameworks (rodada frameworks)

LangGraph: planejamento como grafo explícito de nós (o plano *é* a topologia); Agents SDK e CrewAI: papéis planner/executor e processos sequencial/hierárquico; a spec-driven (Spec Kit/Kiro) trata o plano como artefato versionado com gates. Onde os harnesses de código improvisam o plano no loop, os frameworks o materializam como estrutura de primeira classe.