10 — Subagentes e Orquestração

🕒 estado da arte 2026-07 · revisão 2026-07-26 · 📖 ~13 min de leitura

Objetivos de aprendizagem

Ao final deste capítulo, você deve ser capaz de:

1. **Explicar** por que o ganho primário de um subagente é isolamento de contexto (lê muito, devolve pouco), não paralelismo;
2. **Comparar** as três filosofias — subagente-como-ferramenta, como-serviço e como-colega;
3. **Avaliar** o gate de custo/benefício de decompor-e-paralelizar (a tensão Anthropic × Cognition) e os modos de falha que justificam guardrails;
4. **Distinguir** delegação local de delegação entre sistemas (A2A (Agent-to-Agent)/ACP (Agent Client Protocol)) e quando cada uma se aplica;
5. **Implementar** a tool `task` com sessão-filha e permissões derivadas no harness-zero (etapa 9).

O problema

Um único contexto não segura tarefas grandes: exploração de codebase polui a janela com dumps de arquivos; trabalhos paralelizáveis rodam em série; e um agente generalista faz tudo mediocremente. Subagentes resolvem por **divisão de contexto** (o subagente lê 50 arquivos e devolve só a conclusão), **especialização** (prompts e permissões por papel) e **paralelismo**.

As decisões de projeto:

- **Isolamento**: sessão-filha? Processo separado? Worktree git próprio (para edições paralelas sem conflito)?
- **Permissões**: herda as do pai? Derivadas e restritas? Degradadas por profundidade?
- **Comunicação**: fire-and-forget (retorna um resultado) ou canal contínuo (mailbox, mensagens)?
- **Alcance**: só local, ou delegação a agentes remotos de outros vendors?

Fundamentos científicos

A literatura de sistemas multi-agente (MAS) tem duas mensagens para quem constrói harness: os padrões que funcionam, e a advertência de que a maioria das falhas é de projeto.

- **A falha é de design, não do modelo** — MAST, "Why Do Multi-Agent LLM (Large Language Model) Systems Fail?", arXiv 2503.13657 deriva empiricamente 14 modos de falha em três categorias (especificação/papéis · desalinhamento inter-agente · verificação de tarefa), e conclui que a maioria vem do *sistema*, não dos pesos. Decisão: invista em specs de papel explícitos, checagens de alinhamento e um estágio de verificação dedicado — não num modelo maior.

- **Papéis e SOPs contra alucinação em cascata** — MetaGPT, arXiv 2308.00352 codifica *Standardized Operating Procedures* e papéis de linha de montagem (PM, arquiteto, engenheiro, QA) com artefatos intermediários estruturados, porque encadear LLMs ingenuamente propaga alucinação; saídas escopadas por papel deixam o agente seguinte verificar o anterior. E CAMEL, arXiv 2303.17760 mostra que o role-play **deriva** (troca de papel, repetição, término precoce) — a estabilidade de papel precisa ser *imposta*, não assumida.
- **Topologia programável e recrutamento dinâmico** — AutoGen, arXiv 2308.08155 separa os agentes da topologia de conversa (troque o padrão de orquestração sem reescrever agentes); AgentVerse, arXiv 2308.10848 monta o grupo por tarefa e monitora comportamento emergente negativo. ChatDev, arXiv 2307.07924 decompõe o pipeline em diálogos de duas partes por fase. Ponteiro de taxonomia: o survey de MAS, arXiv 2402.01680.
- **O ceticismo saudável** — o debate multi-agente é uma primitiva de verificação (Du et al., arXiv 2305.14325), mas Should We Be Going MAD?, arXiv 2311.17371 e Stop Overvaluing Multi-Agent Debate, arXiv 2502.08788 mostram que ele nem sempre bate self-consistency/CoT a compute igual. Decisão: **sempre compare o harness multi-agente com um baseline single-agent compute-matched** antes de aceitar a complexidade.

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

Fontes da indústria

- **Subagente = instância isolada com toolset restrito** — Create custom subagents (Claude Code): cada subagente é uma instância *fresca e isolada* lançada pela tool `Task`, com janela de contexto própria e toolset por tipo de agente. Os subagents do Agent SDK são declarados como config (nome, tools, modelo, prompt) — dá para fixar modelos baratos (Haiku para Explore read-only) por papel e impor least-privilege por tipo. Decisão: um subagente de busca queima tokens explorando sem poluir o contexto do orquestrador, devolvendo só um resumo compacto.
- **Orchestrator-worker — e o preço** — o multi-agent research system da Anthropic: um *lead* planeja, grava o plano em memória e spawna subagentes paralelos, cada um com contexto isolado e um **contrato explícito** (objetivo, formato de saída, tools, fronteiras). O ganho de largura vem a **~15× os tokens** de um chat único (e, segundo o post, tokens explicam ~80% da variância de desempenho) — só paga em tarefas de alto valor e muita amplitude. O guia de quando usar multi-agente dá os três casos: poluição de contexto, subtarefas genuinamente paralelas, especialização que afia a seleção de tools. *(anthropic.com 403 pelo proxy; números por espelhos independentes.)*
- **O contra-argumento** — Don't Build Multi-Agents (Cognition): prefira um agente **single-thread com compressão de contexto**. Quando o trabalho se abre em paralelo, cada subagente age sobre uma visão parcial e toma decisões implícitas conflitantes (o exemplo do Flappy Bird: um constrói fundo estilo Mario, outro um pássaro incompatível) — um "telefone sem fio" que cria a etapa de reconciliação que a própria arquitetura gerou. Dois princípios: *compartilhe o traço completo com todo agente* e *ações carregam decisões implícitas, evite as conflitantes*. Para tarefas longas, adicione um modelo de compressão em vez de dividir a thread. *(cognition.com 403; confirmado por HN/GitHub.)*

- **Os frameworks materializam os padrões** — Agents SDK (OpenAI) distingue **handoffs** (transfere controle a um especialista) de **agents-as-tools** (um manager chama sub-agentes como funções, mantendo a thread); o Swarm foi a origem educacional do handoff. CrewAI escolhe entre **sequential** e **hierarchical** (`manager_llm` delega e valida); o LangGraph modela um **supervisor** roteando entre workers com estado persistente; o Magentic-One (AutoGen) mantém um **ledger de progresso** e replaneja na falha; o ADK do Google mistura coordinator/dispatcher com primitivas `Sequential/Parallel/Loop`. Decisão: escolha a forma de coordenação (handoff × tool × supervisor × ledger) pelo que precisa reter — thread, controle ou recuperação.
- **Delegação entre sistemas: A2A (e ACP convergindo nele)** — quando os subagentes vivem em vendors diferentes, a delegação vira protocolo: o A2A usa **Agent Cards** (JSON anunciando identidade, skills, endpoint, auth) para descoberta e **Tasks** com ciclo de vida como unidade de trabalho delegado, sobre HTTP+JSON-RPC (Remote Procedure Call)+SSE (Server-Sent Events); é a generalização cross-org do handoff da tool `Task`. O ACP (IBM/BeeAI) era a alternativa REST-nativa — mas fundiu-se no A2A sob a Linux Foundation em ago/2025. Decisão: para trabalho novo, padronize no A2A (liga ao cap. 17).
- **Consulte também**: a coleção viva Awesome Harness Engineering — Task Runners & Orchestration reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

O estado da arte

1. Três filosofias — ferramenta, serviço, colega

A moldura da primeira rodada persiste e ganhou reforço da rodada 2. **Subagente-como-ferramenta**: pontual, contido, com guardrails (opencode `task` → sessão-filha, depth 1; Aider split architect→editor, depth 1). **Subagente-como-serviço**: registry, contratos de terminação, alcance remoto (gemini-cli `invoke_agent` + A2A; Codex `multi_agents_v2` com **grafo de agentes persistido** e ~100 perfis; Goose `orchestrator` lead/worker). **Subagente-como-colega**: equipes persistentes com comunicação contínua (OpenHarness Swarm com mailbox + worktree git por membro; Hermes com **Kanban dispatcher** e handoffs estruturados).

2. O ganho primário é isolamento de contexto — não paralelismo

O que os três harnesses da rodada 1 já mostravam, a indústria consolidou: o subagente vale porque **lê muito e devolve pouco**. É por isso que o Claude Code o modela como instância *fresca* e isolada, e por que o worktree git (OpenHarness) importa — ele isola *edições* paralelas, não só leituras. Isso é o mesmo princípio do "contexto escopado por subtarefa" do cap. 09 (Beyond Entangled Planning): o subagente é o veículo de escopo de contexto.

3. A tensão central: paralelizar custa, e a maioria das falhas é de design

O eixo de decisão da dimensão é a tensão Anthropic × Cognition. Orchestrator-worker compra largura (+~90% em pesquisa) a **~15× tokens**; o single-thread evita o "telefone sem fio" mas serializa. O MAST fecha o argumento com dados: a maioria das falhas de MAS é de *especificação e coordenação*, não do modelo — o que explica por que todo harness sério cerca subagentes de **guardrails**: profundidade limitada (opencode/Aider

depth 1; OpenClaw 1–5), contratos de terminação (gemini-cli GOAL/MAX_TURNS/TIMEOUT), permissões **degradadas por profundidade** (OpenClaw: subagente nunca ganha `message`/`gateway`/`cron`), e a expressão extrema — o **IronClaw deny-filtra `spawn_subagent` em todos os profiles de produção** (o design suporta, a política proíbe até haver confiança). A regra de projeto: decompor-e-paralelizar é um gate de custo/benefício, com baseline single-agent como controle.

4. A virada: orquestrar harnesses de outros vendors

A fronteira que a rodada 2 tornou concreta: o subagente pode ser *outro harness*. O OpenClaw orquestra Claude Code, Gemini CLI, opencode e Codex como subagentes via runtime **ACP**; o OpenHands (Canvas) orquestra Claude Code, Codex e Gemini via perfis **ACP**; o gemini-cli é cliente **e servidor A2A**. Com o ACP-IBM (Agent Communication Protocol) convergindo no A2A sob a Linux Foundation, o *agent card* vira o contrato universal de delegação entre sistemas. A orquestração deixou de ser interna ao harness e virou interoperabilidade (cap. 17).

> **Adendo da rodada ext-1 (2026-07-31): o isolamento de *workspace* virou infraestrutura.** O corpus isolava o **contexto** do subagente; o Grok Build (xAI, aberto em 2026-07-15) fecha a outra metade — o **filesystem**. Cada `spawn_subagent` com isolamento ativo recebe uma **git worktree própria** criada por uma crate dedicada (`xai-fast-worktree`: CoW paralelo, snapshots BTRFS O(1), overlayfs, metadata com auto-GC), com merge de volta como operação de protocolo (`x.ai/git/worktree/apply`) e fallback gracioso para o workspace compartilhado. A lição não é "usar worktrees" (vários harnesses têm); é o investimento em torná-las **baratas o bastante para o agente usar sem pensar** — subagentes paralelos que editam deixam de brigar pelo working tree. Confirmado no código (`agent/subagent/handle_request.rs`), não só no anúncio.

LEITURA EXECUTIVA

O que está mais moderno: subagente como isolamento de contexto com contrato explícito; a escolha de coordenação (handoff × tool × supervisor × ledger); guardrails motivados por modos de falha reais (MAST); a delegação cross-vendor via A2A; e — desde a rodada ext-1 — o isolamento de workspace por worktree barata (Grok Build). **O que roubar:** dê a cada subagente um contrato (objetivo/formato/tools/fronteiras) e contexto isolado; limite profundidade e degrade permissões por profundidade; compare sempre com um single-agent compute-matched; se subagentes editam em paralelo, isole o filesystem (worktree), não só o contexto; e, se orquestrar entre sistemas, fale A2A.

Mão na massa — harness-zero, etapa 9

A etapa 9 (`harness-zero/etapas/09-subagentes/`) adiciona uma tool `task` que lança um **subagente em sessão-filha**: contexto próprio, **permissões derivadas e restritas** da sessão-pai, e **profundidade máxima 1** (subagente não spawna subagente) — os guardrails que o MAST justifica, na sua forma mínima. O subagente

recebe um contrato (objetivo + formato de saída), roda seu próprio loop e devolve só o resumo ao pai. Exercício de completude: você adiciona a degradação de permissões por profundidade e um contrato de terminação configurável (objetivo + timeout por subagente).

Verificação

1. Seu orquestrador precisa entender 40 arquivos para decidir um refactor, mas você não quer 40 dumps no contexto principal. Como um subagente resolve, e qual é o ganho real? (Isolamento de contexto — o subagente lê os 40 e devolve só a conclusão; o ganho primário não é paralelismo.)
2. Um colega propõe rodar 5 subagentes em paralelo para acelerar. Cite o principal risco (com um nome da literatura/indústria) e o gate que você aplica antes de aceitar. (Telefone sem fio / decisões implícitas conflitantes — Cognition; falhas de coordenação — MAST. Gate: custo/benefício ~15× tokens + baseline single-agent compute-matched.)
3. Você quer que seu harness delegue uma subtarefa a um agente de outro vendor. Que mecanismo usa e qual é o "contrato"? (A2A; o Agent Card anuncia identidade/skills/endpoint/auth, e a Task é a unidade de trabalho delegado.)

Apêndice A — Como cada repositório trata subagentes e orquestração

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

opencode (rodada 1) — delegação contida

Tool `task` (`tool/task.ts`) → subagente em **sessão-filha** (`parentID`), **permissões derivadas e restritas** (`agent/subagent-permissions.ts`), depth 1. Agentes em markdown com modo `primary|subagent|all`; built-in `build`/`plan`/`general`/`compaction`. Modo background experimental (`BackgroundJob`) com `task_id` para **retomar a sessão de subagente**.

gemini-cli (rodada 1) — do subagente local ao remoto

`invoke_agent` sobre `AgentRegistry` (`packages/core/src/agents/registry.ts`); built-in codebase-investigator, generalist, cli-help, browser, skill-extraction, cada um com `ModelConfig`. Terminação explícita (`AgentTerminateMode`: GOAL/MAX_TURNS/TIMEOUT). Exclusividade: **A2A** client+server (`@a2a-js/sdk`, agent cards). Evals de delegação próprias.

OpenHarness (rodada 1) — times, não subagentes

Swarm (`src/openharness/swarm/`, 11 módulos): `AgentTool` em três backends (subprocesso, remoto, teammate in-process); `TeamRegistry`; **mailbox** (comunicação contínua); **worktrees git** (`worktree.py`) para edições paralelas; `permission_sync.py`. Tools `team_create/delete`, `send_message`.

Codex CLI (rodada 2) — grafo de agentes persistido

Duas gerações de API (`multi_agents_v2`: spawn, send_message, followup, interrupt, wait); ~100 perfis de subagentes em TOML; **`agent-graph-store`** (grafo persistido), identidade de agente, comunicação inter-agente, hooks SubagentStart/Stop; `ThreadManager` coordenando threads paralelas.

OpenClaw (rodada 2) — spawn push-based e ACP externo

`sessions_spawn` cria subagentes isolados com **conclusão push-based** (`sessions_yield` como espera sem polling); nesting 1–5; política de tools **degradada por profundidade** (subagentes nunca ganham `message`/`gateway`/`cron`). Runtime **ACP** orquestra Claude Code, Gemini CLI, opencode e Codex como subagentes; Swarm via Code Mode.

Hermes (rodada 2) — Kanban dispatcher

`delegate_task` spawna `AIAgent` filhos com contexto isolado e aprovação não-interativa segura; **Kanban dispatcher** no gateway spawna workers com handoffs estruturados, bloqueio para input humano e heartbeat em operações longas.

Goose (rodada 2) — SubRecipes e orchestrator

`summon` delega a subagentes (Agent filho com recipe própria, eventos streamados); **SubRecipes** com composição hierárquica e execução paralela/sequencial; extensão `orchestrator` (lead/worker: list/start/send/interrupt/stop).

Aider (rodada 2) — architect → editor

Split `architect_coder.py`: um modelo raciocinador produz o plano; após confirmação, um segundo coder (com `editor_model`/`editor_edit_format` próprios) executa. Orquestração de dois papéis com modelos distintos, profundidade fixa 1.

IronClaw (rodada 2) — design elegante, política restritiva

Subagentes como child-runs no mesmo pipeline, com gates/checkpoints unificados e teste E2E — **mas `spawn_subagent` está deny-filtrado em todos os profiles de produção** (`TEMP(disable-spawn-subagents)`). A nota reflete a capacidade disponível, não o design (que seria 3). O caso extremo de "guardrail vence capacidade".

OpenHands / ohmo (rodada 2)

OpenHands: primitivas do SDK (`openhands.sdk.subagent`) + **AgentProfiles** por organização, incluindo perfis **ACP** — o Canvas orquestra Claude Code, Codex e Gemini. ohmo: Agent/Task/Team/SendMessage herdados; assimetria observada (`/tasks run` bloqueado remotamente, tools equivalentes disponíveis ao modelo).

Grok Build (rodada ext-1) — worktrees como infraestrutura ⭐

`agent/subagent/handle_request.rs`: `spawn_subagent` com `capability_mode` **intersectado** com o toolset do tipo (`intersect_capability_modes`), profundidade máx. 1, `resume_from`, contratos de I/O entre personas; isolamento por `WorktreeBuilder…worktree_kind(WorktreeKind::Subagent)` sobre `xai-fast-worktree` (CoW + BTRFS O(1) + auto-GC), merge via `x.ai/git/worktree/apply`; agentes de plugin proibidos de declarar `mcpServers`/hooks/`bypassPermissions`.

Pi (rodada ext-1) — a recusa documentada

Sem subagentes no core, por manifesto ("There's many ways to do this. Spawn pi instances via tmux, or build your own"); o exemplo primeiro-classe `examples/extensions/subagent/` spawna **processos `pi` completos** (isolamento real de contexto) com 4 personas e 3 workflows — a feature existe como prova de que a superfície de extensão basta.

n8n (rodada 2) — agente como tool de agente

AI Agent Tool (`AgentTool.node.ts` v3): um agente completo como tool de outro — o V3 roda o loop do sub-agente inline (`resolveSubAgentRequest`), com proibição de HITL aninhado; **ToolWorkflow** (sub-workflows como tools). Orquestração hierárquica visual.

Frameworks (rodada frameworks)

Agents SDK: handoffs × agents-as-tools; CrewAI: sequential × hierarchical (`manager_llm`); LangGraph: supervisor + workers como nós com estado; AutoGen/Magentic-One: orchestrator com ledger e replanejamento; Google ADK: coordinator/dispatcher + `Sequential/Parallel/Loop`. Os frameworks expõem como API de primeira classe o que os harnesses de código implementam à mão.