10 — Subagents & Orchestration

🕒 state of the art 2026-07 · revised 2026-07-26 · 📖 ~12 min read

Learning objectives

By the end of this chapter, you should be able to:

1. **Explain** why a subagent's primary gain is context isolation (reads a lot, returns a little), not parallelism;
2. **Compare** the three philosophies — subagent-as-tool, as-service, and as-teammate;
3. **Evaluate** the cost/benefit gate of decompose-and-parallelize (the Anthropic × Cognition tension) and the failure modes that justify guardrails;
4. **Distinguish** local delegation from cross-system delegation (A2A (Agent-to-Agent)/ACP (Agent Client Protocol)) and when each applies;
5. **Implement** the `task` tool with a child session and derived permissions in harness-zero (step 9).

The problem

A single context cannot hold large tasks: codebase exploration pollutes the window with file dumps; parallelizable work runs serially; and a generalist agent does everything mediocrely. Subagents solve this through **context division** (the subagent reads 50 files and returns only the conclusion), **specialization** (per-role prompts and permissions), and **parallelism**.

The design decisions:

- **Isolation**: a child session? A separate process? Its own git worktree (for parallel edits without conflict)?
- **Permissions**: inherit the parent's? Derived and restricted? Degraded by depth?
- **Communication**: fire-and-forget (returns one result) or a continuous channel (mailbox, messages)?
- **Reach**: local only, or delegation to remote agents from other vendors?

Scientific foundations

The multi-agent systems (MAS) literature has two messages for harness builders: the patterns that work, and the warning that most failures are design failures.

- **The failure is in the design, not the model** — MAST, "Why Do Multi-Agent LLM (Large Language Model) Systems Fail?", arXiv 2503.13657 empirically derives 14 failure modes in three categories (specification/roles · inter-agent misalignment · task verification), and concludes that most come from the *system*, not the weights. Decision: invest in explicit role specs, alignment checks, and a dedicated verification stage — not in a bigger model.
- **Roles and SOPs against cascading hallucination** — MetaGPT, arXiv 2308.00352 codifies *Standardized Operating Procedures* and assembly-line roles (PM, architect, engineer, QA) with

structured intermediate artifacts, because naively chaining LLMs propagates hallucination; role-scoped outputs let the next agent verify the previous one. And CAMEL, arXiv 2303.17760 shows that role-play **drifts** (role swapping, repetition, early termination) — role stability must be *enforced*, not assumed.

- **Programmable topology and dynamic recruitment** — AutoGen, arXiv 2308.08155 separates agents from conversation topology (swap the orchestration pattern without rewriting agents); AgentVerse, arXiv 2308.10848 assembles the group per task and monitors negative emergent behavior. ChatDev, arXiv 2307.07924 decomposes the pipeline into two-party dialogues per phase. Taxonomy pointer: the MAS survey, arXiv 2402.01680.
- **The healthy skepticism** — multi-agent debate is a verification primitive (Du et al., arXiv 2305.14325), but Should We Be Going MAD?, arXiv 2311.17371 and Stop Overvaluing Multi-Agent Debate, arXiv 2502.08788 show it does not always beat self-consistency/CoT at equal compute. Decision: **always compare the multi-agent harness against a compute-matched single-agent baseline** before accepting the complexity.

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

Industry sources

- **Subagent = isolated instance with a restricted toolset** — Create custom subagents (Claude Code): each subagent is a *fresh, isolated* instance launched by the `Task` tool, with its own context window and a per-agent-type toolset. The Agent SDK subagents are declared as config (name, tools, model, prompt) — you can pin cheap models (Haiku for read-only Explore) per role and enforce least-privilege per type. Decision: a search subagent burns tokens exploring without polluting the orchestrator's context, returning only a compact summary.
- **Orchestrator-worker — and the price** — Anthropic's multi-agent research system: a *lead* plans, writes the plan to memory, and spawns parallel subagents, each with isolated context and an **explicit contract** (objective, output format, tools, boundaries). The breadth gain comes at **~15× the tokens** of a single chat (and, per the post, tokens explain ~80% of performance variance) — it only pays on high-value, high-breadth tasks. The guide to when to use multi-agent gives the three cases: context pollution, genuinely parallel subtasks, and specialization that sharpens tool selection. *(anthropic.com 403 through the proxy; numbers via independent mirrors.)*
- **The counter-argument** — Don't Build Multi-Agents (Cognition): prefer a **single-threaded agent with context compression**. When the work fans out in parallel, each subagent acts on a partial view and makes conflicting implicit decisions (the Flappy Bird example: one builds a Mario-style background, another an incompatible bird) — a game of "telephone" that creates the reconciliation step the architecture itself produced. Two principles: *share the full trace with every agent* and *actions carry implicit decisions; avoid conflicting ones*. For long tasks, add a compression model instead of splitting the thread. *(cognition.com 403; confirmed via HN/GitHub.)*
- **Frameworks materialize the patterns** — Agents SDK (OpenAI) distinguishes **handoffs** (transfers control to a specialist) from **agents-as-tools** (a manager calls sub-agents as functions, keeping the thread); Swarm was the educational origin of the handoff. CrewAI chooses between **sequential** and **hierarchical** (`manager_llm` delegates and validates); LangGraph models a **supervisor** routing among

workers with persistent state; Magentic-One (AutoGen) keeps a **progress ledger** and replans on failure; Google's ADK mixes coordinator/dispatcher with `Sequential/Parallel/Loop` primitives. Decision: choose the coordination form (handoff × tool × supervisor × ledger) by what you need to retain — thread, control, or recovery.

- **Cross-system delegation: A2A (and ACP converging into it)** — when subagents live in different vendors, delegation becomes protocol: A2A uses **Agent Cards** (JSON announcing identity, skills, endpoint, auth) for discovery and **Tasks** with a lifecycle as the unit of delegated work, over HTTP+JSON-RPC (Remote Procedure Call)+SSE (Server-Sent Events); it is the cross-org generalization of the `Task` tool's handoff. ACP (IBM/BeeAI) was the REST-native alternative — but merged into A2A under the Linux Foundation in Aug 2025. Decision: for new work, standardize on A2A (connects to ch. 17).
- **See also**: the living collection Awesome Harness Engineering — Task Runners & Orchestration gathers more consultable resources for this dimension (patterns, articles, and implementations), curated by problem.

The state of the art

1. Three philosophies — tool, service, teammate

Round 1's framing persists and got reinforced by round 2. **Subagent-as-tool**: one-shot, contained, with guardrails (opencode `task` → child session, depth 1; Aider's architect→editor split, depth 1). **Subagent-as-service**: registry, termination contracts, remote reach (gemini-cli `invoke_agent` + A2A; Codex `multi_agents_v2` with a **persisted agent graph** and ~100 profiles; Goose `orchestrator` lead/worker). **Subagent-as-teammate**: persistent teams with continuous communication (OpenHarness Swarm with a mailbox + a git worktree per member; Hermes with a **Kanban dispatcher** and structured handoffs).

2. The primary gain is context isolation — not parallelism

What the three round-1 harnesses already showed, the industry consolidated: the subagent is valuable because it **reads a lot and returns a little**. That is why Claude Code models it as a *fresh*, isolated instance, and why the git worktree (OpenHarness) matters — it isolates parallel *edits*, not just reads. This is the same principle as ch. 09's "context scoped per subtask" (Beyond Entangled Planning): the subagent is the vehicle of context scoping.

3. The central tension: parallelizing costs, and most failures are design failures

The dimension's decision axis is the Anthropic × Cognition tension. Orchestrator-worker buys breadth (+~90% in research) at **~15× tokens**; single-thread avoids the "telephone" game but serializes. MAST closes the argument with data: most MAS failures are failures of *specification and coordination*, not of the model — which explains why every serious harness surrounds subagents with **guardrails**: bounded depth (opencode/Aider depth 1; OpenClaw 1–5), termination contracts (gemini-cli GOAL/MAX_TURNS/TIMEOUT), permissions **degraded by depth** (OpenClaw: a subagent never gets `message`/`gateway`/`cron`), and the extreme expression — **IronClaw deny-filters `spawn_subagent` in all**

production profiles (the design supports it; policy forbids it until there is trust). The design rule: decompose-and-parallelize is a cost/benefit gate, with a single-agent baseline as the control.

4. The turn: orchestrating other vendors' harnesses

The frontier round 2 made concrete: the subagent can be *another harness*. OpenClaw orchestrates Claude Code, Gemini CLI, opencode, and Codex as subagents via an **ACP** runtime; OpenHands (Canvas) orchestrates Claude Code, Codex, and Gemini via **ACP** profiles; gemini-cli is an A2A client **and server**. With ACP-IBM (Agent Communication Protocol) converging into A2A under the Linux Foundation, the *agent card* becomes the universal contract for cross-system delegation. Orchestration has stopped being internal to the harness and become interoperability (ch. 17).

> **Round ext-1 addendum (2026-07-31): *workspace* isolation became infrastructure.** The corpus isolated the subagent's **context**; Grok Build (in Portuguese; xAI, opened on 2026-07-15) closes the other half — the **filesystem**. Each `spawn_subagent` with isolation active gets its **own git worktree** created by a dedicated crate (`xai-fast-worktree`: parallel CoW, O(1) BTRFS snapshots, overlayfs, metadata with auto-GC), with merge-back as a protocol operation (`x.ai/git/worktree/apply`) and graceful fallback to the shared workspace. The lesson is not "use worktrees" (several harnesses have them); it is the investment in making them **cheap enough for the agent to use without thinking** — parallel subagents that edit stop fighting over the working tree. Confirmed in the code (`agent/subagent/handle_request.rs`), not just the announcement.

EXECUTIVE SUMMARY

What's most modern: the subagent as context isolation with an explicit contract; the coordination choice (handoff × tool × supervisor × ledger); guardrails motivated by real failure modes (MAST); cross-vendor delegation via A2A; and — since round ext-1 — workspace isolation via cheap worktrees (Grok Build). **What to steal:** give every subagent a contract (objective/format/tools/boundaries) and isolated context; bound depth and degrade permissions by depth; always compare against a compute-matched single agent; if subagents edit in parallel, isolate the filesystem (worktree), not just the context; and, if you orchestrate across systems, speak A2A.

Hands-on — harness-zero, step 9

Step 9 (`harness-zero/etapas/09-subagentes/`) adds a `task` tool that launches a **subagent in a child session**: its own context, **permissions derived and restricted** from the parent session, and **maximum depth 1** (a subagent does not spawn a subagent) — the guardrails MAST justifies, in their minimal form. The subagent receives a contract (objective + output format), runs its own loop, and returns only the summary to the parent. Completeness exercise: you add permission degradation by depth and a configurable termination contract (objective + per-subagent timeout).

Check your understanding

1. Your orchestrator needs to understand 40 files to decide on a refactor, but you don't want 40 dumps in the main context. How does a subagent solve this, and what is the real gain? (Context isolation — the subagent reads the 40 and returns only the conclusion; the primary gain is not parallelism.)
2. A colleague proposes running 5 subagents in parallel to speed things up. Name the main risk (with a name from the literature/industry) and the gate you apply before accepting. (Telephone game / conflicting implicit decisions — Cognition; coordination failures — MAST. Gate: ~15× token cost/benefit + compute-matched single-agent baseline.)
3. You want your harness to delegate a subtask to another vendor's agent. What mechanism do you use, and what is the "contract"? (A2A; the Agent Card announces identity/skills/endpoint/auth, and the Task is the unit of delegated work.)

Appendix A — How each repository handles subagents and orchestration

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

opencode (round 1) — contained delegation

`task` tool (`tool/task.ts`) → subagent in a **child session** (`parentID`), **derived, restricted permissions** (`agent/subagent-permissions.ts`), depth 1. Agents in markdown with mode `primary|subagent|all`; built-in `build`/`plan`/`general`/`compaction`. Experimental background mode (`BackgroundJob`) with a `task_id` to **resume the subagent session**.

gemini-cli (round 1) — from local subagent to remote

`invoke_agent` over an `AgentRegistry` (`packages/core/src/agents/registry.ts`); built-in codebase-investigator, generalist, cli-help, browser, skill-extraction, each with a `ModelConfig`. Explicit termination (`AgentTerminateMode`: GOAL/MAX_TURNS/TIMEOUT). Its exclusive: **A2A** client+server (`@a2a-js/sdk`, agent cards). Its own delegation evals.

OpenHarness (round 1) — teams, not subagents

Swarm (`src/openharness/swarm/`, 11 modules): `AgentTool` with three backends (subprocess, remote, in-process teammate); `TeamRegistry`; a **mailbox** (continuous communication); **git worktrees** (`worktree.py`) for parallel edits; `permission_sync.py`. Tools `team_create/delete`, `send_message`.

Codex CLI (round 2) — persisted agent graph

Two API generations (`multi_agents_v2`: spawn, send_message, followup, interrupt, wait); ~100 subagent profiles in TOML; **`agent-graph-store`** (persisted graph), agent identity, inter-agent communication,

SubagentStart/Stop hooks; a `ThreadManager` coordinating parallel threads.

OpenClaw (round 2) — push-based spawn and external ACP

`sessions_spawn` creates isolated subagents with **push-based completion** (`sessions_yield` as polling-free waiting); nesting 1–5; tool policy **degraded by depth** (subagents never get `message`/`gateway`/`cron`). An **ACP** runtime orchestrates Claude Code, Gemini CLI, opencode, and Codex as subagents; Swarm via Code Mode.

Hermes (round 2) — Kanban dispatcher

`delegate_task` spawns child `AIAgent`s with isolated context and safe non-interactive approval; a **Kanban dispatcher** in the gateway spawns workers with structured handoffs, blocking for human input, and heartbeats on long operations.

Goose (round 2) — SubRecipes and orchestrator

`summon` delegates to subagents (a child Agent with its own recipe, streamed events); **SubRecipes** with hierarchical composition and parallel/sequential execution; the `orchestrator` extension (lead/worker: list/start/send/interrupt/stop).

Aider (round 2) — architect → editor

The `architect_coder.py` split: a reasoning model produces the plan; after confirmation, a second coder (with its own `editor_model`/`editor_edit_format`) executes. Two-role orchestration with distinct models, fixed depth 1.

IronClaw (round 2) — elegant design, restrictive policy

Subagents as child-runs in the same pipeline, with unified gates/checkpoints and an E2E test — **but `spawn_subagent` is deny-filtered in all production profiles** (`TEMP(disable-spawn-subagents)`). The score reflects the available capability, not the design (which would be a 3). The extreme case of "guardrail beats capability".

OpenHands / ohmo (round 2)

OpenHands: SDK primitives (`openhands.sdk.subagent`) + per-organization **AgentProfiles**, including **ACP** profiles — the Canvas orchestrates Claude Code, Codex, and Gemini. ohmo: inherited Agent/Task/Team/SendMessage; an observed asymmetry (`/tasks run` blocked remotely, equivalent tools available to the model).

Grok Build (round ext-1) — worktrees as infrastructure ⭐

`agent/subagent/handle_request.rs`: `spawn_subagent` with `capability_mode` **intersected** with the type's toolset (`intersect_capability_modes`), max depth 1, `resume_from`, I/O contracts between personas; isolation via `WorktreeBuilder…worktree_kind(WorktreeKind::Subagent)` over `xai-`

`fast-worktree` (CoW + O(1) BTRFS + auto-GC), merge via `x.ai/git/worktree/apply`; plugin agents forbidden from declaring `mcpServers`/hooks/`bypassPermissions`.

Pi (round ext-1) — the documented refusal

No subagents in the core, by manifesto ("There's many ways to do this. Spawn pi instances via tmux, or build your own"); the first-class example `examples/extensions/subagent/` spawns **full `pi` processes** (real context isolation) with 4 personas and 3 workflows — the feature exists as proof that the extension surface suffices.

n8n (round 2) — agent as another agent's tool

AI Agent Tool (`AgentTool.node.ts` v3): a full agent as another agent's tool — V3 runs the sub-agent's loop inline (`resolveSubAgentRequest`), with nested HITL forbidden; **ToolWorkflow** (sub-workflows as tools). Visual hierarchical orchestration.

Frameworks (frameworks round)

Agents SDK: handoffs × agents-as-tools; CrewAI: sequential × hierarchical (`manager_llm`); LangGraph: supervisor + workers as stateful nodes; AutoGen/Magentic-One: orchestrator with a ledger and replanning; Google ADK: coordinator/dispatcher + `Sequential/Parallel/Loop`. Frameworks expose as first-class API what coding harnesses implement by hand.