Shape your harness.

The Monday-morning checklist: five moves, and four files to start from.

1

Write your `AGENTS.md` like a system contract.

It is not a README. It is the contract the agent reads before every task, so treat it like load-bearing code: precise, current, reviewed.

2

Ship one MCP server.

Take the one thing your team reaches for every day that the agent cannot see, be that deployment status, internal docs, or ticketing, and make it a tool, not a paragraph of instructions.

3

Install one hook for lineage.

The smallest hook that captures the *why* alongside the *what* in your history, so that six months from now the diff still explains itself.

4

Run a permissions audit.

Open your harness's permission rules and actually read them, then tighten one and loosen one, both deliberately. Stop treating permissions as someone else's default.

5

Run the single-agent default.

One focused agent with good tools, until you hit a real trigger for decomposition, not before. Resist the swarm.

None of these needs a budget, a meeting, or a promotion.

STARTING POINT ONE

1. AGENTS.md skeleton

A system-contract starter. Keep it short and current: the agent reads it before every task, so stale lines cost you on every run. Whatever your harness calls it (`AGENTS.md`, `CLAUDE.md`, `.cursorrules`), the discipline is the same.

```
# [Project] agent contract

## What this is
One paragraph: what the project does, who it serves, the stack.

## Conventions
- Language, formatting, naming. How tests are run.
- What "done" means here (tests pass, lint clean, reviewed).

## The agent should
- Prefer editing existing files; follow the patterns already here.
- Run the test suite before claiming a task is done.
- Keep changes surgical: touch only what the task needs.

## The agent should not
- Edit [generated dirs], [secrets], or [migrations] without asking.
- Add a dependency without flagging it first.

## Review
This file is load-bearing. Review it like code: precise, current, owned.
```

READ MORE

The **AGENTS.md** open format, used across Claude Code, Cursor, Copilot and more:
agents.md
Writing **CLAUDE.md** and project memory in Claude Code:
code.claude.com/docs/en/memory

STARTING POINT TWO

2. An MCP-server template

A minimal stdio server exposing one tool. Give the agent something your team already has, like a deployment-status lookup. Extend by adding more `.tool(...)` calls, then register it in your harness.

```
// minimal stdio MCP server that exposes one tool.
// npm i @modelcontextprotocol/sdk zod
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";
import { z } from "zod";

const server = new McpServer({ name: "team-tools", version: "0.1.0" });

server.tool(
  "deploy_status",
  { service: z.string().describe("service name") },
  async ({ service }) => {
    const status = await fetchStatus(service); // your internal API
    return { content: [{ type: "text", text: status }] };
  }
);

await server.connect(new StdioServerTransport());
```

READ MORE

Model Context Protocol, the standard itself: modelcontextprotocol.io
Build an MCP server, the official quickstart: modelcontextprotocol.io/docs/develop/build-server

STARTING POINT THREE

3. A lineage hook

A minimal `PostToolUse` hook that appends the *why* to a log on every edit, so the reasoning travels with the change. Save as `.claude/hooks/post-tool-use.sh` and wire it into `.claude/settings.json`.

```
#!/usr/bin/env bash
# Fires after every Edit/Write. Appends one JSONL line of lineage.
# Wire in .claude/settings.json:
#   "hooks": { "PostToolUse": [{ "matcher": "Edit|Write",
#     "hooks": [{ "type": "command",
#                 "command": ".claude/hooks/post-tool-use.sh" }] }] }
set -euo pipefail
PAYLOAD="$(cat -)"
mkdir -p .harness

jq -nc \
  --arg ts      "$(date -u +%Y-%m-%dT%H:%M:%SZ)" \
  --arg session "${CLAUDE_SESSION_ID:-unknown}" \
  --arg tool    "$(printf '%s' "$PAYLOAD" | jq -r '.tool_name // "unknown"')" \
  --arg file    "$(printf '%s' "$PAYLOAD" | jq -r '.tool_input.file_path // ""')" \
  --arg why     "$(printf '%s' "$PAYLOAD" | jq -r '.tool_input.description // ""')" \
  '{ts:$ts, session:$session, tool:$tool, file:$file, why:$why}' \
  >> .harness/checkpoint.log
```

READ MORE

Claude Code **hooks reference** (PostToolUse and the full event list): code.claude.com/docs/en/hooks
Liu et al., *Dive into Claude Code* (the 27-event hook pipeline): arxiv.org/abs/2604.14228

STARTING POINT FOUR

4. A permissions audit

Open `.claude/settings.json` and read the rules out loud. Deny beats allow; the `ask` list is what interrupts you, so keep it short and meaningful. Then change exactly two rules, on purpose.

```
# Permissions audit of .claude/settings.json

[ ] Read the "allow" list. Anything allowed that should not be?
[ ] Read the "deny"  list. Deny-first: a deny always beats an allow.
[ ] Read the "ask"   list. These interrupt you, so keep them few.
[ ] Tighten ONE rule  (allow → ask, or add a deny).
[ ] Loosen  ONE rule  (a safe, frequent action you keep approving → allow).
[ ] Confirm the mode (default / acceptEdits / plan / ...) fits the task.
```

A starting shape:

```
{
  "permissions": {
    "allow": ["Read(*)", "Bash(npm test)"],
    "ask":   ["Edit(*)", "Write(*)", "Bash(*)"],
    "deny":  ["Bash(rm -rf *)"]
  }
}
```

READ MORE

Claude Code **settings & permissions** (allow / deny / ask, and the modes):
code.claude.com/docs/en/settings
Liu et al., *Dive into Claude Code* (the permission-mode design space):
arxiv.org/abs/2604.14228

THE FIFTH MOVE

5. Run the single-agent default

The one move with no file to copy, just a default to adopt. One focused agent with good tools, until you hit a real reason to split the work.

Researchers held one variable equal that the breathless multi-agent benchmarks usually fudge: the *thinking budget*, the tokens an agent spends reasoning. With that controlled, a single agent matched or beat every multi-agent system tested, across three model families and five architectures. The reported "swarm wins" mostly came from compute nobody counted: one headline configuration burned up to **4.7×** more reasoning tokens than its single-agent comparator. More agents do not add information. A single agent sees the whole context at once; a swarm can only pass lossy message-summaries between its members, a game of telephone.

So make one focused agent your default, and treat a swarm as the exception that needs a real trigger. There are three: context that is long *and* genuinely corrupted, noisy enough that one reasoning pass gets lost; sub-tasks that truly do not need to talk to each other; or sub-tasks reaching for genuinely different tool surfaces. If none of those holds, you have a single-agent problem wearing an org chart. And when a single agent struggles, restructure the prompt before you reach for orchestration.

READ MORE

Tran & Kiela, *Single-Agent LLMs Outperform Multi-Agent Systems on Multi-Hop Reasoning Under Equal Thinking Token Budgets* (Stanford, 2026): arxiv.org/abs/2604.02460