Playbook

an agent steering methodology for AI coding agents
https://github.com/horiacristescu/claude-playbook-plugin

```
claude plugin marketplace add
horiacristescu/claude-playbook-plugin
claude plugin install playbook@claude-playbook-marketplace
claude playbook:init
```

THREE DIRECTIONS IN THE FIELD

Where Playbook sits among April 2026 coding agent harnesses.

EVOLVING HARNESS

Rewrite the harness itself from accumulated history. Retros read past tasks and chat logs to update how the agent works.

LIGHT ORCHESTRATION

Run many agents in parallel or unattended. Human owns intent; the harness manages execution, not method.

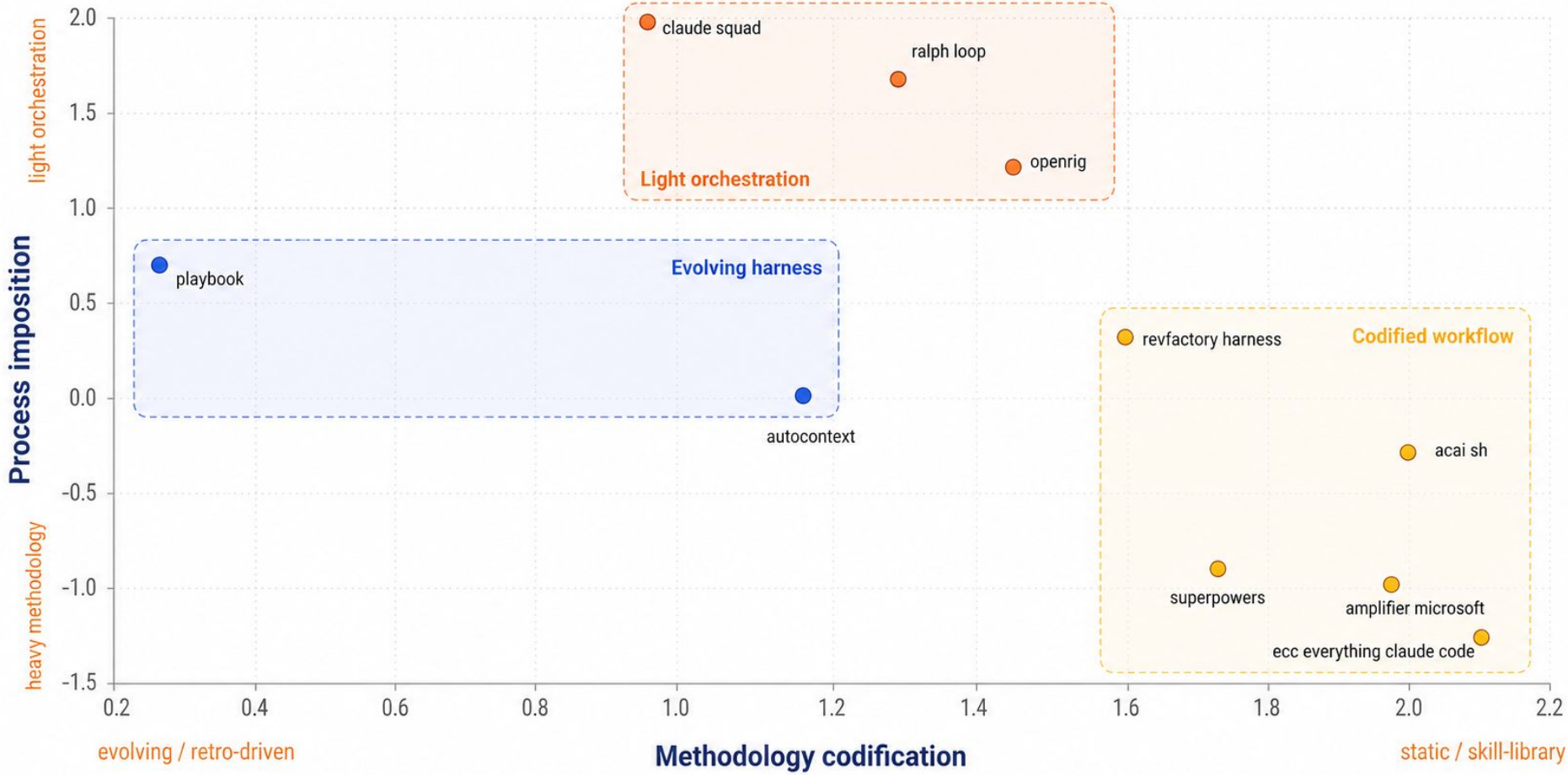
CODIFIED WORKFLOW

Crystallize a process into skills or specs. Brainstorm — plan — subagent — review — TDD, repeated as a fixed pattern.

Playbook is in the EVOLVING HARNESS direction.

Most active systems are codifying their workflow or orchestrating many agents. Few are evolving the harness itself. Playbook and AutoContext are the only two.

Coding agent harnesses for LLM-based development workflows (April 2026)



WHAT MATTERS — AND HOW

1 Intent fidelity

Agent does what you asked

chat_log.md + intent check gate

Every user message is captured. The plan is checked against what you actually said.

2 Verifiable correctness

Wrong work is caught fast

tests after every build gate

Each build step is followed by a test step. Failures are immediately visible to the agent.

3 Auditable trace

Every decision is recorded

task.md as living document

Each gate is annotated as it closes. The task file becomes the record of how the work happened.

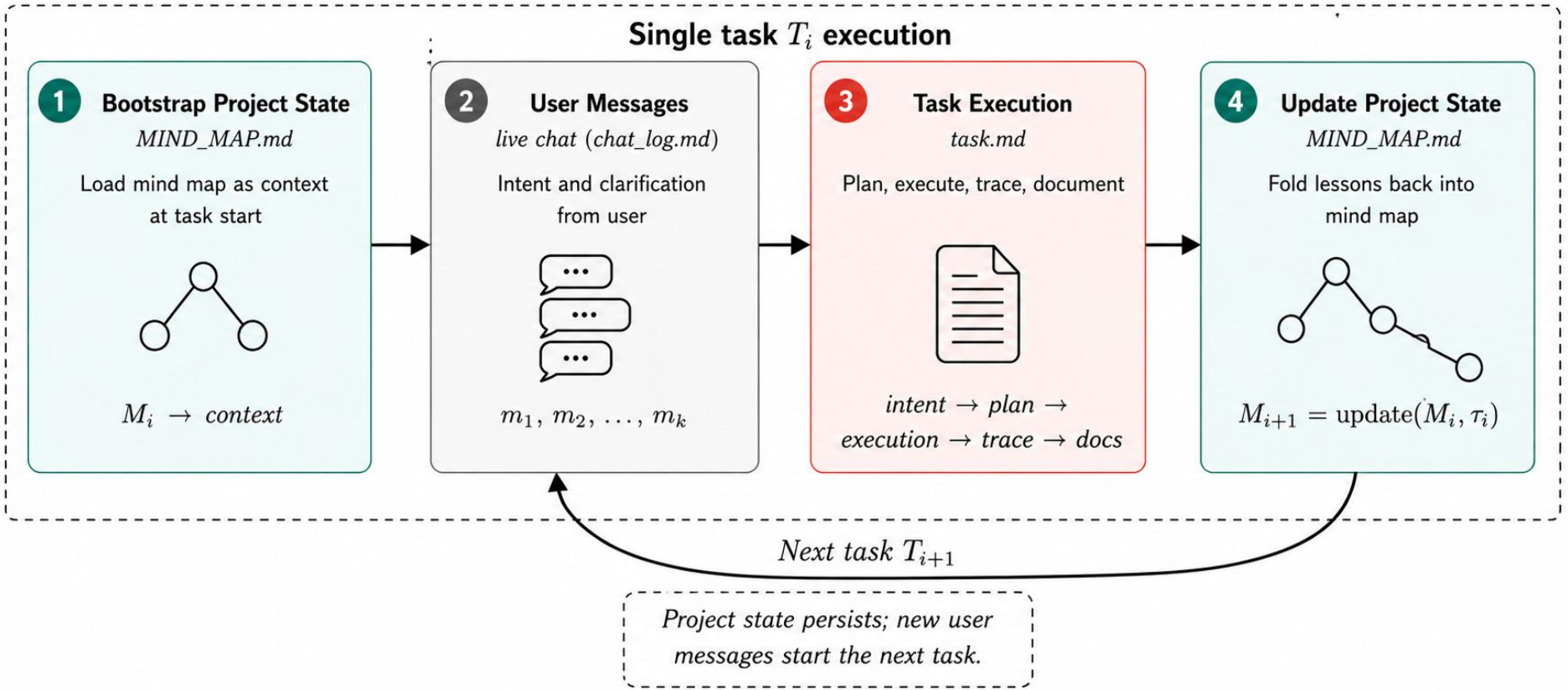
4 Long-term coherence

Survives sessions and compaction

3 memory layers + retros

MIND_MAP, task.md, chat_log persist across sessions. Retros compress many tasks into updated mind map state.

TASK LIFECYCLE: Per-task memory cycle



RETRO: Project-wide compression across multiple tasks

Retrospective update over task sequence $T_1, T_2, \ldots, T_n$

1 Collect Task Traces

task.md × N

Completed tasks across the project

$\tau_1, \tau_2, \ldots, \tau_n$

2 Gather Conversation History

chat_log.md

Accumulated user-agent dialogue

$C = m_1, m_2, \ldots, m_K$

3 Load Current Project State

MIND_MAP.md

Mind map before retro

M_n

4 Compress and Synthesize

retro task

Extract recurring patterns, drop noise

$M_{n+1} = retro(M_n, \{\tau_i\}, C)$

5 Updated Project State

MIND_MAP.md

Project-level patterns folded in

M_{n+1}

ANATOMY OF A TASK

One file, six handoffs.

Zoom: a gate, before and after.

`task.md`

1 **USER**

`## Intent`
What the user wants. Captured from chat.

2 **AGENT (plan)**

`## Design Phase`
Orient, structure, verify scope.

3 **JUDGE**

`## Plan Review`
Blind: judge sees plan + code, never the conversation.

4 **AGENT (work)**

`## Work Plan`
Build gate → test gate → checkpoint.

5 **JUDGE**

`## Implementation Review`
Blind: judge checks code against plan.

6 **AGENT (mind map)**

`## Debrief + Mind Map Update`
Lessons fold back into *MIND_MAP*.md.

top to bottom in time

MIND_MAP.md

BEFORE — gate is drawn

```
- [ ] Step: Implement the rate limiter with
      a token bucket. Watch out for clock
      skew on retries.
```

agent works, checks the box, appends outcome.

AFTER — gate is closed

```
- [x] Step: Implement the rate limiter with
      a token bucket. Watch out for clock
      skew on retries. — Done. Used
      `time.monotonic()` to avoid skew.
      Added unit test for retry-after window.
```

The original gate text is never rewritten.
The outcome is appended in teal after an em-dash.

THINK DEEPLY ABOUT TESTING

Why should you trust what the agent made? What risks are you taking?

1 Tests are project skin

Persistent, deterministic, indifferent to whether the agent is in the room. Code that works without an LLM.

Compensates for agent forgetfulness. The project remembers when the agent doesn't.

2 You only know what you tested

Untested code is unknown code. The test surface is the boundary of what you can claim about the project.

Vibe testing — 'I ran it once, looked fine' — does not transfer or scale.

3 Without tests, complexity disintegrates

Agents forget; tests don't. Every new change risks breaking what was working. Tests are what holds large projects together.

Test pain is design pain. If tests fight the code, the design is wrong.

IN PLAYBOOK — TESTING IS A POLICY, NOT A SUGGESTION

Encoded in the `task.md` template: every Build gate is paired with a Test gate.
The agent writes tests as it builds, not after. The Stop hook blocks closing the task while a Test gate is open.

Tests answer the buyer's question — what is the risk surface of code I did not write myself?

MIND_MAP.md: a graph format that LLMs already know how to read and write

Six self-referential nodes. Every [N] link is real — generated and followed by the same agent.

1 Routing index

Entry point for the whole map. Routing nodes `[1-5]` form the table of contents — read these first on bootstrap. Inline citation format `[2]` makes the syntax LLM-native, grep and git make it tool-friendly `[3]`, wiki-style linking `[4]` gives the mental model, single-line node format `[5]` enables cheap edits, and progressive disclosure `[6]` keeps bootstrap cost low.

2 Inline citation format

Every node uses `[N]` references inlined in prose. LLMs already write this — academic papers trained the pattern in. The agent generates the link and the surrounding text in the same sentence; no separate metadata layer, no graph schema. Round-trip works because the same agent reads `[3]` what it wrote earlier.

3 Grep, git, agent-friendly

Plain text, line-oriented. `grep '\[7'\'` finds every reference to node 7. `git diff` shows exactly which nodes changed in a commit. No parser, no database, no embedding model. Same agent that updates `[5]` a node also reads it next session — the format is the protocol.

4 Wiki-style cross-linking

Any node references any other node. Not a tree, not a list — a graph. The same node `[2]` gets cited from multiple entry points: the routing index references it as syntax, `[3]` references it as the thing grep operates on, `[6]` references it for progressive disclosure. Many-to-many is the default.

5 Single-line nodes, line-oriented edits

Each node fits on one logical line: `[N]` `**Title**` — body with `[M][K]` inlined. Overwrite the line, you've updated the node. No JSON schema, no validation pass. The cost of changing a node is the cost of editing one line, which is why the agent actually maintains `[6]` the map rather than letting it drift.

6 Progressive disclosure

Bootstrap loads routing nodes `[1-5]` only — under 5KB. Body content fetched on demand by node ID. Same pattern as skill frontmatter: cheap to scan, full content lazy-loaded. The mind map can grow to hundreds of nodes without inflating bootstrap cost, because the agent navigates `[4]` rather than slurping.

The format is its own documentation — you are reading a working mind map of the mind map format.