SIDEBAR

AI

Loop Engineering

Sidebar

AI · Loop Engineering

Sidebar is where this site keeps content that's genuinely worth a real, standalone write-up, but too narrow to justify a multi-chapter course of its own — link collections, interactive tools, and one-off lessons like this one, organized by subject rather than by a fixed outline. This lesson covers **loop engineering**, a real, current shift in how AI coding agents are actually used.

What "Loop Engineering" Actually Means

In June 2026, Addy Osmani and Boris Cherny — the engineer who built Claude Code — put a name to a shift that had already been happening in practice: **loop engineering**. Cherny's own line has become the unofficial founding statement of the idea: *"I don't prompt Claude anymore. I have loops that are running. They're the ones that are prompting Claude and figuring out what to do. My job is to write loops."*

The distinction is about where the effort actually goes. **Prompt engineering** optimizes a single exchange: you write an instruction, the model responds, you read the output, and if it's wrong, you write a better instruction — you hold the pen for the entire conversation. **Loop engineering** optimizes a system that runs that conversation on your behalf, repeatedly, without you reading every single reply. A loop, in this sense, is a goal-driven cycle: receive a goal, gather context, decide on an action, use a tool or call another agent, observe what happened, update state, and repeat.

This Isn't Entirely New Ground for This Site

> **AN HONEST NOTE BEFORE GOING FURTHER**
>
> Most of the actual mechanics behind "loop engineering" were already taught in *Claude Code Agents: Advanced Orchestration* — under different names, before the term itself existed. That course's own Chapter 2 covered the Agent SDK and the Tool Runner's automated loop; Chapter 3 covered parallel vs. sequential orchestration; Chapter 4 covered background and autonomous agents; Chapter 7 covered agent-to-agent handoffs and shared state; Chapter 9 covered cost/context/efficiency tradeoffs. Loop engineering is best understood as a fresh, useful *framing* for that same underlying architecture — not a wholly new mechanism replacing it.

The Five Core Techniques

Osmani's own framing identifies five concrete techniques loop engineering is actually built from — each with a real, working implementation in tools like Claude Code today:

TECHNIQUE	WHAT IT DOES	ALREADY COVERED ON THIS SITE
Automations	Scheduled discovery and triage work, running independently of an active conversation	*Claude Code Agents: Advanced Orchestration 4* — background & autonomous agents
Worktrees	Isolated working directories so parallel agents don't collide on the same files	Not yet — genuinely new to this site, covered in depth below
Skills	Reusable project knowledge, loaded on demand instead of re-derived every time	Implicit throughout this site's own skill-driven workflows (e.g. `/links`, `/my-tools`), not previously named as its own concept
Plugins/Connectors	Integration with external tools and services via the Model Context Protocol (MCP)	Touched on briefly in *Claude Code Agents: Fundamentals*'s own tool-permission material
Sub-agents	Separating the agent that implements a task from the agent that verifies it	*Claude Code Agents: Fundamentals 1* and *Advanced Orchestration 5* — the security/compliance review agent is exactly this pattern

> **A WELL-DEFINED PATTERN, NOT SPRAWLING CHAOS**
>
> Loop engineering maps cleanly onto five specific, nameable techniques — it's an emerging engineering discipline with real boundaries, not a vague buzzword covering "anything automated." That's part of why it's a genuinely useful lens even where it overlaps with material this site already taught.

What's Actually New: Worktrees for Parallel Isolation

Claude Code Agents: Advanced Orchestration 3 already taught the deciding question between parallel and sequential agent work. What it didn't cover is the specific mechanism that makes running several agents on the *same codebase* at the *same time* safe: a **git worktree**.

A worktree is an additional working directory linked to the same underlying git repository — a second (or third, or fourth) checkout, each with its own files on disk, all sharing the same commit history and git objects underneath. Two agents working in two separate worktrees can edit files freely without either one ever seeing, or overwriting, the other's uncommitted changes — because each agent's own file edits genuinely live in a different folder.

```
# create a second, isolated working directory for a parallel agent,
# checked out on its own branch, sharing the same repo history
git worktree add ../repo-agent-2 -b agent-2-feature

# the second agent now works entirely inside ../repo-agent-2 —
# its file edits cannot collide with anything happening in the original folder
```

> **WITHOUT A WORKTREE, "PARALLEL" QUIETLY BECOMES "COLLISION-PRONE"**
>
> Running two agents in parallel inside the *same* working directory — the default, if nothing explicit is done about it — means both are reading and writing the same files on disk. One agent's edit can silently overwrite the other's, or one agent can read a half-finished intermediate state left by the other. A worktree isn't an optimization here; it's what makes "parallel" actually mean parallel rather than "two agents racing over the same files."

A Real, Concrete Loop

Cherny's own "my job is to write loops" line is a useful test: what does actually writing one look like? A simple, realistic example — a loop that watches for new work, triages it, and only surfaces what's genuinely actionable:

1. A scheduled trigger wakes the loop (an **automation**)
2. It gathers context — new issues, failed test runs, whatever the goal is scoped to
3. For each item, it spawns a dedicated **sub-agent** to investigate, isolated in its own **worktree** if the investigation involves editing code
4. Each sub-agent reports back a distilled result — not its full exploratory process, just the finding
5. The loop decides: escalate to a human, take an automatic action, or discard as not actionable
6. It goes back to sleep until the next trigger, and repeats

THIS ISN'T HYPOTHETICAL — THIS VERY SESSION HAS A VERSION OF IT

Claude Code's own `/loop` skill, used in dynamic-pacing mode, is a working instance of exactly this shape: it schedules its own next wake-up based on what it's actually waiting for, rather than a human re-prompting it turn by turn. The gap between "a real loop" and "an interesting idea from a video" is smaller than it might sound — the mechanism already exists in the tool being used to read this lesson.

The Honest Caveat: Verification Is Still On You

Loop engineering doesn't remove the need for human judgment — it relocates it. Instead of reviewing every individual response, the real work shifts to designing good stop conditions, sensible escalation rules, and periodic verification that the loop is actually doing what it's supposed to. The genuine risk, named directly in the original writing on this topic, is **cognitive surrender** — letting a loop's own fluent output stand in for actually checking its work. A loop that runs unattended for longer, on more consequential tasks, needs *more* deliberate verification design, not less, echoing exactly the `/loop` skill's own guidance on scheduling meaningful check-ins rather than optimistically assuming everything will go right.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, the difference between prompt engineering and loop engineering, specifically in terms of who reads each individual model response.

View solution

EXERCISE 2

Two agents are launched to work on the same codebase at the same time, in the same working directory, with no worktrees set up. Explain the specific risk this chapter describes, and how a worktree would fix it.

View solution

EXERCISE 3

A developer sets up an unattended loop to auto-merge any pull request an agent marks "looks good," with no periodic human review. Explain why this chapter would call this a case of "cognitive surrender."

View solution

QUICK REFERENCE

- **Loop engineering** — designing a system that runs the prompt/response cycle on your behalf, repeatedly, instead of hand-prompting turn by turn; coined June 2026 by Addy Osmani and Boris Cherny
- The generic loop shape: goal → gather context → decide action → use a tool/call an agent → observe → update state → repeat
- **Five core techniques:** automations, worktrees, skills, plugins/connectors (MCP), sub-agents — most already covered on this site under different names in *Claude Code Agents: Advanced Orchestration*
- **Worktrees** are the genuinely new piece: isolated working directories letting parallel agents edit the same repository without colliding on files
- Loop engineering relocates human judgment to designing stop conditions and verification, rather than eliminating the need for it — "cognitive surrender" is the real risk of skipping that design work