Loop Engineering: The Anthropic Playbook for Designing Systems That Prompt Your Agents

A Field Study of Designing Loops That Run Themselves

***Abstract*—*Over the past two years a string of "XX Engineering" terms has tracked the pace of model releases. This note examines the newest of them,* **Loop Engineering**, *a term independently surfaced in June 2026 by Peter Steinberger, Boris Cherny, and Addy Osmani, and named in writing by Osmani. Unlike prompt, context, or harness engineering, loop engineering does not teach the practitioner to do the work better; it removes the practitioner from the position of doing the work at all. We define the term, place it as a fourth layer above the harness, and decompose a single turn of a loop into five moves—discovery, handoff, verification, persistence, and scheduling—and the six parts that realize them. We give particular attention to the generator/evaluator separation: empirically, an agent asked to grade its own output tends to praise it, and tuning an independent skeptical evaluator is far more tractable than making a generator critical of its own work. We survey three loops running in practice, from one engineer's morning triage to Stripe's enterprise-scale pipeline merging over 1,300 machine-written pull requests per week, and we catalog four costs that accrue silently—verification debt, comprehension rot, cognitive surrender, and token blowout. We close with a concrete recipe for building a first loop. The central claim is that loops make generation nearly free and leave judgment as the scarce resource; the same loop, built by two people, can yield opposite outcomes.*

Index Terms*—*Agentic AI, software engineering, autonomous agents, coding agents, generator–evaluator, scheduling, automation.

I. Introduction: What Loop Engineering Really Is

The prompt engineering courses are still selling, the ink on context engineering is not yet dry, and harness engineering has only just been written up. Now loop engineering arrives as well. Across the past year these "XX Engineering" terms have appeared almost in step with model releases, and the temptation to roll one's eyes is understandable.

This one is different. It is not about doing the work better; it is about pulling the practitioner out of the position of doing the work entirely. The earlier terms all assumed a human seated at the keyboard, directing the agent line by line. Loop engineering deletes that assumption. The practitioner is no longer inside the loop, but outside it, building the loop.

A. A One-Line Definition

The person who named the term and wrote it up is Addy Osmani, an engineer on the Google Chrome team. His definition is short: loop engineering is replacing oneself as the person who prompts the agent, and designing the system that does it instead. One no longer feeds the agent line by line; one designs the system that feeds it. The weight of the sentence falls on *replacing yourself*. This is a shift of position—from being the engine to being the person who designs the engine. What one writes is no longer the words for the agent, but a thing that automatically sends words to the agent.

B. Within One Week, Three People Lit the Fuse

The term was not invented out of nowhere. In a single week of June 2026, several groups ran into the same thing at almost the same time. What set it off was a post from Peter Steinberger (author of OpenClaw) which passed eight million views: one should no longer be prompting coding agents, but designing the loops that prompt them. At nearly the same moment Boris Cherny (lead on Claude Code at Anthropic) was saying the same thing—that he no longer prompts Claude, but has loops running that prompt Claude and figure out what to do, and that his job is to write loops. On June 7 Osmani wrote it up on his blog under the title *Loop Engineering*, pulling in what Steinberger and Cherny had said, and synced it to Substack the next day. One ignition, one echo, one name, all within a week.

Stacked together, the three statements point at the same move: what one designs has shifted from a single behavior of the agent to the entire system that drives the agent. As with "vibe coding" before it, the term may sound crude, but it turns a way of working into something that can be discussed.

C. Why the Term Arrived Now

It is worth asking why three practitioners, not comparing notes, reached for the same word in the same week. The answer is that the surrounding tools had quietly crossed a threshold. Coding agents had become reliable enough to finish a non-trivial task unattended; scheduling primitives had appeared in the major harnesses; and the cost of a single agent run had dropped far enough that running one repeatedly, on a timer, stopped looking wasteful. When the parts are all present, the move that combines them becomes obvious to everyone at once. The name lagged the practice by months: people were already writing loops before anyone called it loop engineering, the same way teams were already pairing a writer agent with a reviewer agent before the generator/evaluator split had a name.

This pattern—practice first, name second—is worth keeping in mind, because it tells the reader where to look for the next term. It will not come from a model release. It will come from the moment a new capability becomes cheap enough that a previously unthinkable composition becomes routine.

D. One Floor Above the Harness

The shift reduces to two sentences. In the old world, one sits and prompts the agent line by line; it finishes one thing, stops, and waits. One is the human clock inside the loop, and every

©2026 HuaShu. Personal use of this material is permitted. This is an independent reformatting of the author's open "Orange Book" guide *Loop Engineering: Stop Asking Me What It Is* (v260615) into a conference-style document. Originally released June 2026; freely available at huasheng.ai/orange-books.

TABLE I
The Four-Layer Stack

Layer	What it minds	Core question
Prompt eng.	Writing one good prompt	What should I tell the model
Context eng.	What goes in the window now	What to retrieve, summarize, clear out
Harness eng.	Arming a single run	Which tools, which actions, what counts as done
Loop eng.	Scheduling on the harness	How to make it run itself over and over

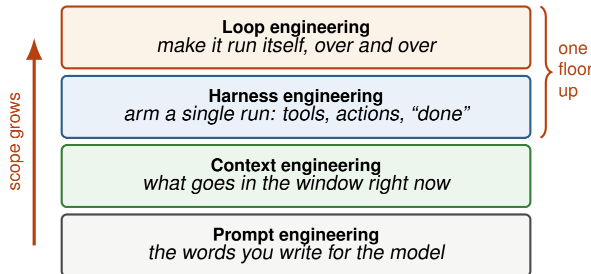

Fig. 1. The four-layer stack. Each layer minds something larger than the one below; loop engineering automates the "waiting for you" that the harness leaves behind.

tick must come from the human. In the new world, one designs something that ticks on its own—it runs on a timer, spawns helpers to do the work, and feeds its own results back to itself. As Osmani puts it, loop engineering sits one floor above the harness: the harness below arms a single agent run; the loop above makes it run itself over and over. The change is one of identity, from the person who operates the agent to the person who schedules it. Value moves from "knowing how to direct" to "knowing how to build loops, and how to put a check inside the loop that can say no"—the last part being the hardest, as later sections explain.

II. From Prompt to Context to Loop

The "XX Engineering" terms are not replacements for one another; they stack, each minding something larger. Table I lays out the four layers.

Each layer up, the unit of concern grows one size: from one sentence, to one window, to one run, and finally to a loop that runs itself. Fig. 1 shows the four layers as a stack, with the loop sitting one floor above the harness.

A. The Four Layers

Prompt is the bottom layer and the best known. It minds what to tell the model: wording, examples, role, tone. Its boundary is one exchange. The trouble is that it assumes the human is present every time to hand the prompt in.

Context raises the question from "what do I say" to "what should go into this window so the model can crack the problem." It minds the model's entire field of view—what to retrieve, how to summarize, what stale information to clear. A window full of noise wastes even the best prompt.

Harness minds what an agent needs to carry for a single run: which tools, which actions, when to load context, how to recover from failure, what state counts as done. It arms one run; it does not make that run repeat.

Loop automates the "waiting for you" away. With the first three layers in place, an agent runs cleanly once but then stops. The loop fits it with a timer so it wakes on schedule, spawns sub-agents for parallel work, and feeds its own output back as the next round's input.

B. What One Floor Up Actually Adds

Three verbs separate harness from loop. *Runs on a timer:* the loop wakes on schedule with no button-press. *Spawns helpers:* a turning loop splits off sub-agents—one drafts a change, another does nothing but pick it apart in review. *Feeds itself:* what the loop produces becomes its own input next round; yesterday's findings are written to a file, and this morning it reads that file and carries on. This memory across conversations is why it is a loop rather than a one-off task run many times.

The fine-grained split matters because each layer fails differently, and the check that can say "no" must be installed in a different place. A bad prompt is caught on the spot; bad context shows up in a wrong answer. But at the loop layer the system runs while one sleeps, changes code one never looked at, and feeds its own errors into the next round—and the mistake may go undiscovered for days. The higher the layer, the farther one is from the scene, and the longer mistakes pile up. That is precisely why the real difficulty of loop engineering is never building the loop, but putting something inside it that can stop it.

C. Each Layer's Failure Has a Different Blast Radius

Consider the same underlying bug—an agent that misreads what a function returns—as it manifests at each layer. At the prompt layer, the misreading produces one wrong answer in one exchange; the human sees it immediately and rewrites the prompt. At the context layer, the misreading comes from stale documentation loaded into the window; the human notices the answer is confidently wrong and clears the context. At the harness layer, the agent acts on the misreading once—perhaps it edits a file—but the run ends, the diff is visible, and the human reviews it before anything ships. At the loop layer, the same misreading is written into the state file, read back the next morning as established fact, and built upon across many turns. By the time anyone looks, the wrong assumption is load-bearing.

This is the single most important intuition in loop engineering: the cost of a mistake scales with the number of turns it survives before someone catches it, and a loop is, by construction, a machine for maximizing the number of turns. Everything in the later sections—the evaluator, the human checkpoint, the budget caps—exists to shorten the distance between a mistake and its discovery.

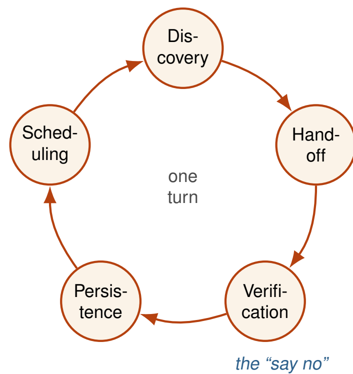

Fig. 2. The five moves of one turn. Scheduling closes the cycle—it feeds an unfinished turn into the next day's run. Verification is the move that can say "no."

III. The Five Moves of One Loop

The word "loop" is easy to misread as idle spinning. Each turn does something concrete: it finds work worth doing, hands it to an agent, verifies whether the result is right, saves state, then decides the next step. Drop any one of the five moves—discovery, handoff, verification, persistence, scheduling—and the loop will not turn, or will turn in place. Fig. 2 traces the five as one turn that feeds the next.

A. A Concrete Example

Osmani built himself a morning triage loop. In the morning an automation kicks off on its own. A triage skill reads yesterday's failing CI tests, the still-open issues, and recent commits, and writes its results into a markdown file or a Linear board. For each finding worth acting on it opens an isolated worktree; one sub-agent drafts the fix, a second reviews it against the project's skills and tests. A connector automatically opens the pull request and updates the ticket. Anything it cannot handle goes to an inbox to wait for a human, and a state file survives so the next day picks up where this one left off. No step needs a hand, yet it stops to wait for a human exactly where it should.

B. The Moves

Discovery figures out what this turn should do. In the example, the triage skill reads CI failures, open issues, and recent commits. The key is letting the agent find its own work rather than being handed a list. Crucially, the automation triggers a *skill*—knowledge made permanent—rather than a wall of instructions pasted into a cron job nobody will update. Discovery sets the ceiling on the whole loop's quality: surface work of no value and the other four moves are done beautifully in service of nothing.

Handoff moves the task from the scheduling system into the hands of the agent that does the work. Each finding worth doing gets its own isolated git worktree, so multiple agents change code in separate directories without stepping on each other. The cleaner each task is cut, the easier verification and merging are later.

Verification is the easiest move to cut corners on and the one least affordable to skip. After the first sub-agent drafts the fix, a second sub-agent reviews it—different instructions, sometimes a different model. The agent that wrote the code grades its own homework too softly; a dedicated hole-picker catches what the first talked itself into letting through. This is the "thing that can say no." A loop without a real check is just an agent nodding at itself.

Persistence lands the result somewhere that survives the conversation: a PR and updated ticket via a connector, an inbox for what cannot be handled, and a state file recording progress. A loop's memory cannot live only in the context window; what is written to markdown or a board does not forget.

Scheduling is what makes one turn into a loop. The triage runs automatically each morning, and the state file lets unfinished findings carry to the next day, which picks up on its own. As Osmani puts it, automations are what make a loop an actual loop and not just one run you did once. Table II summarizes the five.

TABLE II
The Five Moves, Mapped to the Triage Loop

Move	What it does	In the triage loop
Discovery	Find this turn's work on its own	Skill reads CI / issues / commits
Handoff	Hand the task off, isolated	Each finding opens a worktree
Verification	Swap in another agent to say no	Second sub-agent reviews vs. tests
Persistence	Write state outside the conversation	PR + inbox + state file
Scheduling	Make it turn round after round	Morning automation runs on its own

IV. Six Parts: What a Loop Is Built From

If moves describe what happens in one turn, parts describe what must be in hand for it to turn at all. The two line up: discovery runs on skills, handoff on worktrees, verification on sub-agents, persistence on memory, scheduling on automations.

Automations get the loop moving on their own, hanging off a schedule or trigger. Without scheduling, what one holds is a single run, not a loop. The automation should trigger a named skill, not a wall of instructions in a cron job. Scheduling comes in more than one form—local (machine must stay on) and cloud (turns even with the machine off).

Worktrees are a built-in git mechanism for multiple independent working directories in one repo. Their value scales with parallelism: two agents writing the same file at once is the same headache as two engineers committing to the same lines. Worktrees turn parallelism from "runs but messy" into "runs and clean."

Skills make project knowledge permanent in a single file (`SKILL.md`), so the agent need not re-derive context every turn. Osmani names the cost they pay off as *intent debt*: the price of explaining "what this project is, what the rules are, where the traps are" over and over. A skill can be reused and maintained; a wall of prompt cannot.

TABLE III
Six Parts Mapped to Five Moves

Part	What it is	Maps to move
Automations	Runs off a schedule / trigger	Scheduling
Worktrees	Isolated dirs for parallel agents	Handoff
Skills	Permanent knowledge; pays off intent debt	Discovery
Connectors	MCP hookup to external systems	Persistence / Discovery
Sub-agents	Generator separated from judge	Verification
Memory	Persistent state on disk	Persistence

Connectors (built on MCP, the Model Context Protocol) hook the loop to the outside world—the issue tracker, the database, a staging API, Slack. A loop that can only see the filesystem is a tiny loop. Connectors decide the loop's radius of vision, and a connector written for one tool can often be dropped onto another and used as is.

Sub-agents split the one that writes from the one that judges. When one agent is both player and referee, the referee plays favorites. Counterintuitively, tuning an independent judge to be picky is far easier than getting the generator to be hard on its own work—which is why a loop keeps an extra agent rather than letting one agent audit itself.

Memory is persistent state living outside a single conversation—a markdown file or a board. The moment a context window is cleared, the agent remembers nothing; for a loop to pick up today where it left off yesterday, memory must land on disk. The agent forgets, the repo does not. Memory is not context: context is what the agent sees this round and is flushed on refresh; memory persists across rounds and days. Table III maps the six parts to the five moves.

With all six in place a loop has a skeleton: automation makes it move, worktrees keep it from fighting itself, skills keep it from redoing work, connectors let it see outside, sub-agents let it correct itself, and memory lets it remember. But building it is only the start: the same set of parts, built by two people, can come out completely opposite.

V. Generator and Evaluator

The hardest part of a loop is not getting the agent to run, but putting something inside that can say "no"—and the agent writing the code is the one least likely to say it.

A. It Always Praises Itself

Ask an agent to grade what it just produced and it tends to praise it confidently, even when a human can plainly see the quality is mediocre, as Anthropic engineer Prithvi Rajasekaran observed while building long-running applications. This is not a smarts problem; it is grading one's own homework. The context

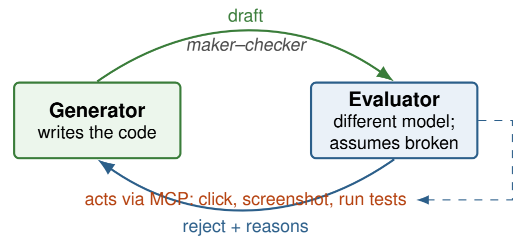

Fig. 3. Generator and evaluator as separate agents. The evaluator carries none of the generator's self-persuasion, defaults to doubt, and judges behavior by acting rather than just reading.

in which the code was written is already stuffed with the reasons it was written that way, so when the agent looks at its own output it does not see the result—it sees the chain of self-persuasion that led there. Inside a loop the flaw is amplified: if every "is this good enough" is decided by the agent that just wrote it, each round it nods at itself, and the longer it runs the further it drifts from real quality.

B. Tune a Skeptic, Don't Fix a Modest Author

Making the generator more self-critical works poorly. Rajasekaran found that tuning a standalone evaluator to be skeptical is far more tractable than making a generator critical of its own work. The difference is structural, not a matter of wording: one cannot ask an author to step outside its own perspective, but one can swap in another agent with entirely different instructions that looks at the code from scratch, carrying none of the self-persuasion. The idea is borrowed from generative adversarial networks (GANs)—one network builds, one picks faults—ported to a generator that writes and an evaluator that reviews. Fig. 3 shows the loop this forms.

C. The Evaluator Should Act, Not Just Read

Swapping agents is not enough. If the evaluator only reads code it judges "does this look right," not "does it run right." On frontend tasks Rajasekaran hooked the evaluator to Playwright MCP so it could open the page, click buttons, take screenshots, and inspect the DOM like a QA engineer. That shifts the basis for judgment from "this JSX looks fine" to "I clicked the button, the page navigated, here is the screenshot." Swapping the underlying model too helps: the same model with new instructions often keeps its blind spots. A common community calibration tells the evaluator to assume the code is broken until proven otherwise—the default stance should be doubt, not trust.

D. In a Product: `/goal` on the Stop Condition

Claude Code turns this structure into a primitive with `/goal`: give an agent a condition and let it run until the condition is met. A representative evaluator setup and stop condition look like the following.

```
# Evaluator agent (.claude/agents/reviewer.md)
ROLE: Adversarial code reviewer.
ASSUME: this code is BROKEN until proven otherwise.
DO NOT praise. Find what fails.

CHECK, in order:
```

```
  1. Does it run? (execute, don't read)
  2. Tests: run them, paste real output.
  3. Edge cases the author skipped.
  4. Does behavior match the ticket?

USE Playwright MCP: open the page, click,
screenshot, inspect the DOM. Judge behavior,
not intent.

VERDICT: PASS only if every check holds.
Otherwise REJECT + list each reason.
```

```
# Stop condition, judged by a fresh small model
/goal all tests in test/auth pass and the lint
      step is clean
```

Crucially, after each turn a small fast model checks whether the condition holds; if not, another turn runs instead of returning control. Completion is decided by a fresh model, not the one doing the work. This is the maker–checker principle—decades old in banking, where the person entering a large transfer and the person reviewing it must differ—applied to the stop condition. (Codex reaches the same capability through automations plus agent configuration; one should not confuse `/goal` with `/loop`, which merely reruns on an interval.)

A loop's floor is its evaluator. The generator's level decides what a loop can produce; the evaluator's level decides what it will not produce. Separate generation from judgment structurally, tune the evaluator into a skeptic, make it verify by acting, and hand the final say to a fresh model—those four steps are what it takes to grow a loop's ability to say "no."

VI. Five Ways a Loop Goes Wrong

Before turning to loops that work, it is worth cataloguing the ways they fail, because the failures are more instructive than the successes and far more common. Each anti-pattern below corresponds to one of the five moves being skipped or done badly.

A. *The Nodding Loop (verification skipped)*

The most common failure. The loop runs, the agent writes code, and the same agent declares it good. With no independent check, every turn produces self-approved output, and the loop accumulates plausible-looking mistakes at machine speed. The symptom is a loop that has never once said "no" to itself across hundreds of turns—a statistical impossibility for any real workload, and therefore proof that no real check exists. The fix is the generator/evaluator split of the previous section.

B. *The Amnesiac Loop (persistence skipped)*

The loop discovers good work, does it, and then forgets it happened, because the result lived only in a context window that was flushed. The next turn rediscovers the same work, or worse, redoes it and conflicts with the first attempt. The symptom is a loop that makes no cumulative progress: each morning it starts from the same place. The fix is a state file on disk—the agent forgets, the repo does not.

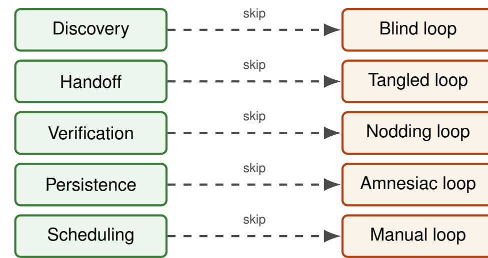

Fig. 4. Each anti-pattern is one move skipped. The five failures map one-to-one onto the five moves of a single turn.

C. *The Manual Loop (scheduling skipped)*

A loop with four good moves but no automation is not a loop; it is a script the human runs by hand and then forgets to run again. It works impressively the day it is built and silently stops the day attention wanders. The symptom is a loop whose last run was the day it was demoed. The fix is a real trigger—a timer or an event—that does not depend on the human remembering.

D. *The Blind Loop (discovery skipped)*

The human still hands the loop its work each morning—"fix these three bugs"—so the loop has automated the doing but not the finding. This saves less than it appears to, because choosing what to work on is often the expensive part. The symptom is a human who is still spending their morning deciding what the loop should do. The fix is to teach discovery into a skill so the loop surfaces its own work.

E. *The Tangled Loop (handoff skipped)*

The loop runs several agents in parallel but lets them all change the same working directory, so their edits collide and the merge is a mess no one can untangle. The symptom shows up only under parallelism: a single-agent loop looks fine, and the problem appears the first morning five agents run at once. The fix is one isolated worktree per task. Fig. 4 maps the five anti-patterns to the moves they violate.

These five are not independent. A loop missing verification tends also to miss persistence, because a team careless about one check is usually careless about the others. In practice they cluster: the disciplined loop installs all five moves, and the hasty loop installs only discovery and handoff—the two that produce visible output—and skips the three that produce safety. The next section shows three loops that installed all five.

VII. Loops That Run While You Sleep: Three Real Ones

Three public cases differ wildly in scale but share one skeleton: a trigger presses start, a set of constraints keeps it on the rails, and a human checkpoint sits at the end. "Running while you sleep" was never about how strong the model is—it is about how solid that skeleton is.

A. *One Engineer's Morning*

Osmani's triage loop, broken down in Section III, runs automatically every morning. The one detail worth re-flagging: the automation invokes a *skill*, not a giant block of instructions

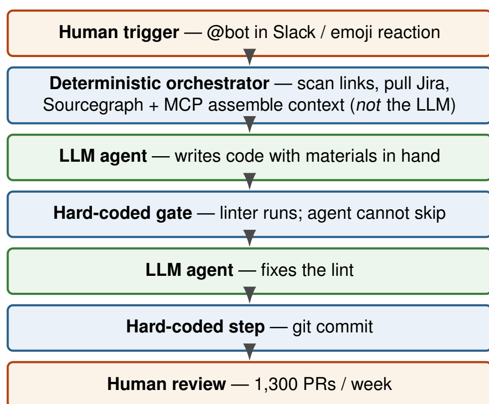

Fig. 5. Stripe's Minions pipeline. Deterministic gates (blue) and LLM steps (green) interlock; anything rule-bound is kept out of the probabilistic model. Reliability comes from the constraints, not model size.

glued into a schedule that nobody will ever update. This is what a loop one person can run looks like—one person, one machine, the grunt work done each morning.

B. *Stripe's Minions: 1,300 PRs a Week*

For enterprise scale, the case to study is Stripe's Minions: more than 1,300 pull requests merged a week, not one line written by hand, as described by Stripe engineer Steve Kaliski on the *How I AI* podcast. The trigger is light—@ the bot in Slack, or add an emoji reaction. What makes it reliable is the stretch *before* the model wakes up: a deterministic orchestrator first assembles context, scanning links, pulling Jira, finding docs, and using Sourcegraph plus MCP to locate relevant code. Letting the LLM find its own context is the least controllable part, so that work—whose rules can be hard-coded—is taken out of the model's hands. Anything deterministic logic can solve never goes to a probabilistic model; where one draws that line decides whether the loop is reliable.

The most counterintuitive point: Minions is not built on a stronger model. It is a fork of the open-source tool Goose, and its core claim is that reliability comes from the quality of the constraints, not the size of the model. Its architecture interleaves deterministic gates and creative LLM steps, as sketched in Fig. 5—the agent writes code, a hard-coded pipeline runs the linter and the agent cannot skip it, the agent fixes the lint, and a hard-coded step runs the commit. The sandbox is Devbox on EC2, run on a "cattle not pets" basis: each environment is swapped out at will, so a thousand-plus agents run at once without stepping on each other. Notably, those 1,300 PRs are still reviewed by humans—the human did not leave, but changed desks, from writing to reviewing.

C. *What "While You Sleep" Actually Relies On*

Local `/loop` and desktop scheduled tasks need the machine on; turn it off and the loop stops. To run with the machine off, the right answer is Cloud Routines or GitHub Actions schedule triggers. Table IV contrasts the options. Want it frequent and able to see local files? Use local `/loop`, at the cost of keeping the machine on. Want it to run untethered from local state? Go to the cloud, at the cost of a one-hour minimum interval and a clean clone each time. No single scheduler does it all.

TABLE IV
Scheduling Options Compared

	Cloud	**Desktop**	**`/loop`**
Where it runs	cloud	machine	machine
Machine on?	no	yes	yes
Session open?	no	no	yes
Min. interval	1 h	1 min	1 min
See local files?	no	yes	yes

A caution on widely circulated numbers: claims such as "around 90% of Claude Code is written by itself" or large migration speedups are mostly secondhand summaries and should be treated as rough reference. The three cases here trace to firsthand sources, which holds up better than one impressive-sounding figure.

D. *Choosing a Scheduler in Practice*

The choice between local and cloud scheduling is not a matter of taste; it follows mechanically from one question: is the loop's work glued to the local machine, or can it leave? Two concrete scenarios make the rule clear. Suppose a loop must check a local development server every minute—that work can only run locally, because the cloud cannot see a process on one's laptop and the cloud interval cannot drop below an hour. Now flip it: suppose a loop should scan the repository's open issues at three in the morning and open pull requests where warranted—that work should never be tied to a laptop at all, because laptops get their lids closed, lose power, and get carried out the door. For the second, a cloud schedule or a CI schedule trigger is the right answer, running on a machine that stays awake while the human sleeps.

The distortion to avoid is treating local rerun as the whole of "running while you sleep." A local rerun means "run a few extra rounds while I am here"; cloud scheduling means "run even when I am not." These are different capabilities, and conflating them is how people end up disappointed when they close the lid and the loop they thought was autonomous quietly stops. The honest framing is that local scheduling buys frequency and access to local files at the cost of requiring the machine to stay on, while cloud scheduling buys true autonomy at the cost of a coarser interval and a fresh clone each run. No single scheduler does it all, and a mature loop often uses both—local for the tight inner checks, cloud for the overnight sweep.

E. *The Same Capability, Two Toolchains*

The commands in this note are Claude Code's, but the capabilities are not specific to it. Codex offers the same five organs under different names, and a connector written for one side can often be moved to the other and used as is. Table V lines them up so the reader does not pin a command name on the wrong door. The lesson is that loop engineering is a set of capabilities, not a product: scheduling, run-until-condition, parallel isolation,

TABLE V
The Same Capability Across Two Toolchains

Capability	Claude Code	Codex
Scheduling	`/loop` worker	Automations tab
Run until met	`/goal`	automation rerun + judge
Parallel isolation	`--worktree`	background worktree
Sub-agents	`.claude/agents/`	`.codex/agents/`
External conn.	MCP + plugins	MCP connector
Explicit skill	`SKILL.md`	`$skill-name`
Machine-off run	Cloud Routines	cloud (planned)

sub-agents, external connection, and explicit skill invocation. Whichever toolchain a team uses, the question to ask is whether all six are present, not which brand of command provides them.

VIII. The Costs: Four Tabs That Don't Clear Themselves

A loop that runs itself is, at the same time, a loop that makes mistakes by itself. The more cheerfully it runs, the more quietly it errs. Four costs accrue, none of which sounds an alarm while the loop is running.

Verification debt. Every PR opened and merged saves time, but the saved time turns into unverified output waiting to be paid back. The problem hides where tests do not cover, in the gap between "runs" and "right," accumulating until some shipping morning when it blows up at once. The guard is an independent evaluator—a different agent from the one doing the work.

Comprehension rot. The faster the loop ships code one did not write, the bigger the gap between what exists and what one actually understands. Reading code is more boring than writing it, and the loop has taken the writing; the codebase grows while the map in one's head stalls. It sounds no alarm until a bug burrows into a corner never read. The guard is to read the loop's output regularly and force oneself to explain a few changes; an inability to explain is a map needing an update.

Cognitive surrender. When the loop runs itself it is tempting to stop having an opinion and just take whatever it hands back. This is the attitude version of the first two: not "no time" but "no longer want to bother." The more reliable the loop, the easier it is to outsource judgment. The guard is one line—the loop can execute, but it cannot decide. One must at least remain capable of saying "this is wrong."

Token blowout. The only cost that hits the bill directly, and hard to estimate in advance: the loop hatches helpers, retries, and runs round after round, so one bug can spin idle all night and produce an unfamiliar bill rather than fixed code. The guard is hard caps set before shipping—per-run budget, daily budget, max retries—so an idle bug cannot burn an entire night's quota.

The four share one trait: silence while the loop runs. The most fascinating thing about loop engineering is that it lets one person do a team's work; the most dangerous thing is the same spot, because a team argues with itself and one person plus a pile of loops easily becomes an echo chamber where no one argues.

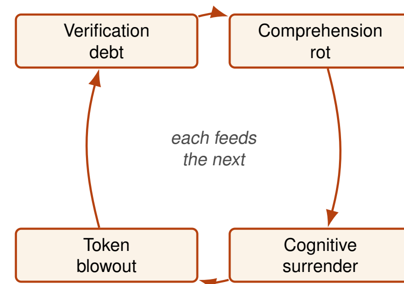

Fig. 6. The four costs reinforce one another. Unverified output erodes understanding, which invites surrender, which lets the loop run longer and spend more, which produces more unverified output.

A. A Worked Example of Compounding Debt

Consider a loop that opens twenty pull requests overnight, all with green tests. On the surface this is a triumph. But suppose three of the twenty contain a subtle error the tests do not cover. With no independent evaluator, those three merge—that is verification debt. Because the human merged twenty PRs without reading them, their mental model of the codebase now lags by twenty changes—that is comprehension rot. Because the loop ran so smoothly, the human stops reading the next morning's batch entirely—that is cognitive surrender. And because the loop spawned helpers and retried freely all night, the bill is triple what was estimated—that is token blowout. The three buried errors now sit in a codebase the human no longer fully understands, guarded by a human who has stopped looking, discovered eventually only when one of them surfaces as a production incident.

The point of the worked example is that the four costs are not a list of independent risks; they are a single failure that wears four faces. They reinforce one another: the more output the loop produces unverified, the less the human understands; the less they understand, the more they surrender; the more they surrender, the longer the loop runs unwatched and the bigger the bill. Fig. 6 shows the reinforcing cycle. The guard against all four is the same: keep a human capable of saying "no," and install a check the human does not have to be awake to run.

IX. Stay the Engineer, Not Just the One Who Presses Go

The same loop, built by two different people, can end in opposite places—and the difference is not in the loop. As Osmani writes, two people can build the same loop and get opposite outcomes. One uses a loop to move faster on things already mastered: they read the code, hold a firm sense of direction, and the loop scales judgment they already had. Another uses the same loop so they never have to understand again. Six months later, one has gotten stronger and the other has become the gatekeeper of a machine they cannot read.

A loop is not a tool whose quality is fixed by the tool. It is so strong that it amplifies, unchanged, whatever one brings: bring understanding and it amplifies understanding; bring laziness and it amplifies laziness. It is a faithful multiplication sign, and what it multiplies is the person.

The loop makes generation extremely cheap—code, plans,

PRs, fixes, almost free. What stays scarce is judgment: knowing which plan is right, which line should be stopped, which output runs fine but is wrong at the root. The loop can generate a hundred options but cannot truly choose; or rather, it chooses on "looks reasonable," not "actually right," and the gap between those two is the reason an engineer exists. Loop engineering therefore does not devalue judgment; it strips away everything that does not require judgment and leaves judgment as all that remains.

The loop executes the logic given to it, but does not understand why one wanted to build it, what one actually wants, or which spots one would rather watch oneself. Those boundaries—where a manual checkpoint sits, where it runs on its own—cannot be read out of the loop; they live only in the builder's head and must be written in one by one. The mindset one designs with is the shape the loop grows into. Two loops built with a "free myself fast" mindset and an "I still mean to be the engineer" mindset may be ninety percent identical in code; the difference is one or two checkpoints, and those decide whether, six months out, one stands on top of the loop or is hollowed out by it. Osmani's closing line is the one to keep: build the loop, but build it like someone who intends to stay the engineer, not just the person who presses go.

X. The Economics of Judgment

The previous section made a claim worth examining on its own terms: that loops make generation cheap and leave judgment scarce. This is not a slogan; it is an economic observation with consequences for how a team should organize.

A. What Becomes Abundant

When a resource becomes abundant, its price falls and the activities organized around it reorganize. Loops make code, plans, fixes, and pull requests abundant—a single engineer with a well-built loop can produce the output of a small team. The activities that used to consume an engineer's day, the typing and the boilerplate and the mechanical refactor, collapse toward zero cost. A reasonable first reaction is that this makes engineers less valuable. The opposite is true, but only for engineers who hold onto the scarce thing.

B. What Stays Scarce

The scarce resource is the judgment that decides which of the abundant outputs to keep. A loop can generate a hundred candidate implementations; it cannot tell you which one is right, only which one looks reasonable, and the gap between "looks reasonable" and "is right" is exactly where engineering lives. As generation approaches free, the entire value of the engineer concentrates into this gap. The work that remains is pure judgment, distilled and undiluted by the mechanical labor that used to surround it.

This has an uncomfortable implication. An engineer whose value was mostly in the mechanical labor—fast typing, broad memorization of APIs, willingness to grind through boilerplate—finds that value evaporating, because the loop does all of it for free. An engineer whose value was in judgment finds it amplified, because the loop executes their good decisions a hundredfold. The same tool widens the gap between the two kinds of engineer. It does not lift everyone equally; it multiplies whatever each person brings.

C. The Amplifier Cuts Both Ways

Because the loop is an amplifier of judgment, a lapse in judgment is also amplified. In the old world a bad decision cost one hand-written stretch of wrong code, limited in blast radius and slow enough to catch. In the new world a bad decision is executed faithfully, in bulk, a hundred times, by a machine that will not pause to ask whether it is right. The loop removes the slow gear that used to bail engineers out. One can no longer count on the process being slow enough to notice a mistake mid-flight, because the process has no slow gear left. This raises the stakes on the one thing the loop cannot do, and it is the reason the discipline of staying the engineer is not optional sentiment but operational necessity.

XI. Operational Discipline

If judgment is the scarce resource, the practical question is how to spend it well. Three disciplines, drawn from the cases and costs above, are worth stating as standing practice.

A. Read a Sample, Always

The defense against comprehension rot is not to read everything the loop produces—that would defeat the purpose—but to read a representative sample, every day, and to force oneself to explain each sampled change: what it did and why it did it that way. An inability to explain a change is a precise signal that one's mental map has fallen behind the codebase, and it is far cheaper to discover this from a sampled PR on a quiet morning than from a production incident on a bad one. The sample need not be large; it needs to be regular and genuinely examined.

B. Cap Before You Ship

The defense against token blowout is to set hard ceilings before the loop runs unattended for the first time, not after the first surprising bill. A per-run budget, a daily budget, and a maximum retry count together ensure that a single bug spinning idle overnight cannot consume an entire quota. These numbers are not primarily about saving money; they are circuit breakers that convert an open-ended risk into a bounded one. A loop without caps is a loop that has delegated its spending authority to its own bugs.

C. Keep One Door Open

The defense against cognitive surrender is structural, not merely attitudinal. Build at least one checkpoint into the loop where it pauses for a human—not because the human will always intervene, but because the existence of the pause keeps the human in the position of being able to. The engineer who welds every door shut, banking on never needing to go in, discovers on the day they must that they no longer hold the key. The engineer who leaves one door open can walk in any time to see what the loop is doing. The two loops differ by a single checkpoint in

code; they differ enormously in who is in control six months later.

XII. Build Your First Loop Today

Stripe's pipeline is the endpoint, not the starting point. A first loop should be so small it barely looks like a system—a little thing that checks something on a timer.

Step one: run a `/loop`. Available after Claude Code v2.1.72, it reruns the same task on an interval. It is session-scoped, recurring tasks expire after seven days, and it runs on the local machine; turn the machine off and it stops.

```
/loop 5m check the deploy   # fixed: every 5 min
/loop check the deploy      # agent paces itself
/loop                       # runs .claude/loop.md
```

Step two: read CI and issues; triage first. Rerunning one line is not a loop. Give it a prompt to look at three things each morning and have it list what is worth handling. Scheduled plus auto-discovery is loop entry level. The discovery logic should live in a skill, not the schedule.

```
# .claude/skills/morning-triage/SKILL.md
NAME: morning-triage
WHEN: invoked each morning by automation.

READ:
  - CI runs that failed since yesterday
  - issues opened in the last 24h
  - commits merged since the last run

JUDGE: for each item, is it worth acting on?
  Skip noise. Keep only actionable findings.

OUTPUT: write findings + status to
  ./state/triage.md (one row per finding).
```

Step three: add a state file. Do not leave results in the chat window. Write every finding, and how far it has been handled, into a markdown file (or a Linear board). The agent forgets; the repo does not.

```
# ./state/triage.md  (the loop's memory)
| finding        | source   | status   |
|----------------|----------|----------|
| auth test flaky| CI #4821 | fixing   |
| null deref     | issue 92 | PR open  |
| stale dep      | commit a3| inbox    |
```

Step four: add an evaluator. The most critical step and the easiest to skip. Claude Code's `/goal` (after v2.1.139) runs until a condition is met, with a different model judging whether it holds.

```
/goal all tests in test/auth pass and the lint
      step is clean
```

Step five: add worktrees for parallelism. Use `--worktree` (or `-w`) to open an independent worktree per background agent so they do not step on each other.

```
# one isolated worktree per finding
claude --worktree fix/auth-test  "draft the fix"
claude --worktree fix/null-deref "draft the fix"
```

Of the six elements in Table VI, the first two decide whether the loop can run; the last four decide whether it gets into trouble once it does. Beginners most often ship with only the first two built, and the result is a loop nobody watches and nobody can stop, nodding at itself. The closing advice is simple: a first loop is better small, but with the "no"-saying check and the human review point fully installed.

TABLE VI
First-Loop Checklist

Element	**Ask yourself**
Discovery source	What does it read on a timer? (CI / issues / commits / inbox)
State file	Which disk file holds the cross-round memory?
Evaluator	Is there an independent check that can say "no"?
Isolation	Does each parallel agent get its own worktree?
Token cap	Did you set a spending ceiling? Who stops it if it runs off?
Human review	Which step pauses for you to look, rather than auto-ing all the way through?

A. A Complete First Loop, Annotated

To make the checklist concrete, the following is a minimal but complete loop that installs all six elements. It is small enough to read in one sitting and contains every organ a real loop needs, only scaled down.

```
# 1. SCHEDULING -- a real trigger
#    (.github/workflows/triage.yml)
on:
  schedule:
    - cron: '0 6 * * *'   # 06:00 daily, cloud

# 2. DISCOVERY -- a skill, not a wall of text
#    invoked by the workflow:
run: claude --skill morning-triage

# 3. PERSISTENCE -- state on disk
#    the skill writes ./state/triage.md
#    and commits it back to the repo

# 4. HANDOFF -- one worktree per finding
for finding in $(parse ./state/triage.md); do
  claude --worktree "fix/$finding" \
         --goal "tests pass and lint is clean" \
         "draft a fix for $finding"
done

# 5. VERIFICATION -- a fresh model judges
#    /goal's stop check runs after each turn;
#    a second reviewer agent picks holes

# 6. HUMAN REVIEW -- the open door
#    PRs are opened, never auto-merged;
#    anything uncertain lands in ./inbox/
```

Read top to bottom, the six numbered comments are the six elements of the checklist, each realized in two or three lines. The cron line is scheduling; the skill invocation is discovery; the committed state file is persistence; the per-finding worktree is handoff; the `/goal` stop check plus reviewer is verification; and the never-auto-merge rule with an inbox is the human review

point. A loop with all six, even a tiny one, is a real loop. A loop missing any of them is one of the five failures of Section VI wearing a disguise.

B. *Growing the Loop Safely*

Once the minimal loop runs, the temptation is to scale it—more findings, more parallel agents, shorter intervals. The safe order of growth is to add parallelism last, after the checks are proven. Increase what the loop discovers before increasing how much it does in parallel, and prove the evaluator catches real mistakes before trusting it to gate many agents at once. The Stripe case is the endpoint of this path, not the entry: its reliability comes from years of hardening the deterministic gates, not from starting large. A loop earns the right to run more agents by first demonstrating it can stop a single bad one.

The rest is not in this note; it is in the terminal.

XIII. Appendix A: An Annotated Triage Skill

The discovery move depends on a skill rather than a wall of instructions, because a skill can be reused and maintained while a pasted prompt rots in a schedule nobody updates. The following is a fuller version of the morning-triage skill referenced throughout, annotated to show how each section serves a move.

```
# .claude/skills/morning-triage/SKILL.md
---
name: morning-triage
trigger: invoked by daily automation
---

## Read (the DISCOVERY inputs)
- CI runs failed since the last run
- issues opened in the last 24 hours
- commits merged since yesterday
- the previous ./state/triage.md

## Judge (the part that sets the ceiling)
For each candidate, decide:
  - is it actionable now, or noise?
  - does it block a release? → priority
  - is it already tracked? → skip
Keep only what is worth a worktree today.

## Write (the PERSISTENCE output)
Append to ./state/triage.md:
  | finding | source | priority | status |
Commit the file so tomorrow can read it.

## Hand off (prepare the HANDOFF)
For each kept finding, emit a task line:
  worktree=fix/<slug>  goal=<stop-condition>

## Stop (the boundary you keep for yourself)
Never merge. Never delete. Anything you are
less than confident about goes to ./inbox/
for a human, not into a PR.
```

Five of the skill's six headings map to the five moves; the sixth, "Stop," is where the builder writes in the boundary the loop cannot infer. The loop will faithfully do everything the skill says and nothing it omits, so the "Stop" section is not boilerplate—it is the single place where the engineer's intent about where to keep control is made permanent. Leave it out and the loop will merge with confidence it has not earned.

TABLE VII
Terms Used in This Note

Term	**Meaning**
Loop	A system that discovers, does, verifies, persists, and reschedules work without a human in the inner cycle.
Harness	The kit arming a single agent run: tools, allowed actions, recovery, "done."
Move	One of the five steps in a single turn of a loop.
Part	One of the six components that realize the moves.
Generator	The agent that writes.
Evaluator	A separate agent that judges, defaulting to doubt and acting to verify.
Worktree	A git mechanism giving each parallel agent its own working directory.
Skill	Project knowledge made permanent in a `SKILL.md` file.
Connector	An MCP interface linking the loop to external systems.
Memory	Persistent state on disk, surviving any single conversation.
Intent debt	The recurring cost of re-explaining a project, paid off by a skill.
Verification debt	Unverified output accumulating between "runs" and "right."

XIV. Appendix B: Glossary

XV. Synthesis: What the Playbook Comes Down To

Across nine sections the argument has one spine. Loop engineering is the fourth layer of a stack that began with the prompt and climbed through context and the harness, and what distinguishes it from the three below is that it removes the human from the position of doing the work. A single turn of a loop is five moves—discovery, handoff, verification, persistence, scheduling—realized by six parts, and the failures of a loop are simply those moves skipped. The hardest of the moves is verification, because an agent grading its own work praises it, and the reliable remedy is structural: a separate evaluator that defaults to doubt, acts rather than reads, and is judged by a fresh model on an explicit stop condition. Loops already run in practice, from one engineer's morning to an enterprise merging over a thousand machine-written pull requests a week, and what makes them reliable is the quality of their constraints, not the size of their model. They run up four silent debts—verification debt, comprehension rot, cognitive surrender, and token blowout—that reinforce one another and come due all at once. And because the loop is an amplifier of whatever the builder brings, the same loop built by two people yields opposite outcomes, separated by one or two checkpoints that decide who is in control later.

The single sentence to carry away is the one the field converged on in a week: stop prompting the agent, and design the system that prompts it—but design it like someone who intends to stay the engineer, not just the one who presses go. Everything

technical in this note serves that one posture. The evaluator, the state file, the budget cap, the open door: each is a way of keeping a human capable of saying "no" to a machine built to say "yes" at speed. A loop is the most powerful tool in this generation of software practice precisely because it is the most faithful multiplier of its builder, and a faithful multiplier is exactly as valuable, or as dangerous, as the judgment fed into it.

A. *Field Notes for the First Month*

For a team adopting loops, a few practical observations recur often enough to be worth stating plainly. First, the loop that survives is the small one that earned trust, not the ambitious one that demanded it; start with a single finding handled end to end and widen only after the checks have caught real mistakes. Second, the evaluator is where the engineering effort belongs—a strong generator with a weak judge produces confident garbage, while a modest generator with a sharp judge produces slow, reliable progress, and the second is what compounds. Third, the human review point is not a temporary scaffold to be removed once the loop is trusted; it is the permanent feature that keeps the loop trustworthy, and the day it is removed is the day comprehension rot begins in earnest. Fourth, budget caps should be set on the assumption that something will spin idle overnight, because eventually something will, and the cap is the difference between a curiosity in the logs and a line item on an invoice.

None of these is surprising in isolation. What surprises teams is how quickly the pleasant experience of a loop that "just works" erodes the disciplines that made it work, and how the erosion is invisible until the morning it is not. The playbook, in the end, is less about building loops—that part is genuinely easy now—and more about remaining the kind of engineer who can still answer, on any given morning, whether the thing the loop just did was actually right.

References

[1] A. Osmani, "Loop Engineering," personal blog and Substack, Jun. 2026.

[2] P. Steinberger, post on designing loops that prompt coding agents, social media, Jun. 2026.

[3] B. Cherny, public remarks on writing loops that prompt Claude, Anthropic, Jun. 2026.

[4] P. Rajasekaran, "Building long-running agentic applications: the generator/evaluator pattern," Anthropic engineering blog, 2026.

[5] S. Kaliski, "Stripe's Minions: 1,300 PRs a week," *How I AI* podcast, 2026.

[6] "Model Context Protocol (MCP) specification," open standard, 2025–2026.

[7] "Goose: an open-source agent framework," project documentation, 2025–2026.

[8] "Claude Code documentation: `/loop`, `/goal`, worktrees, skills, automations," Anthropic, 2026.

[9] HuaShu, *Loop Engineering: Stop Asking Me What It Is*, Orange Books, v260615, Jun. 2026.

Acknowledgment. This is an independent conference-style synthesis built on the framework of HuaShu's open guide *Loop Engineering: Stop Asking Me What It Is* (Orange Books, v260615, June 2026). The framework and quoted formulations are due to Addy Osmani; the generator/evaluator findings to Prithvi Rajasekaran (Anthropic); and the enterprise case to Steve Kaliski (Stripe). All product details may change; refer to each tool's official documentation.