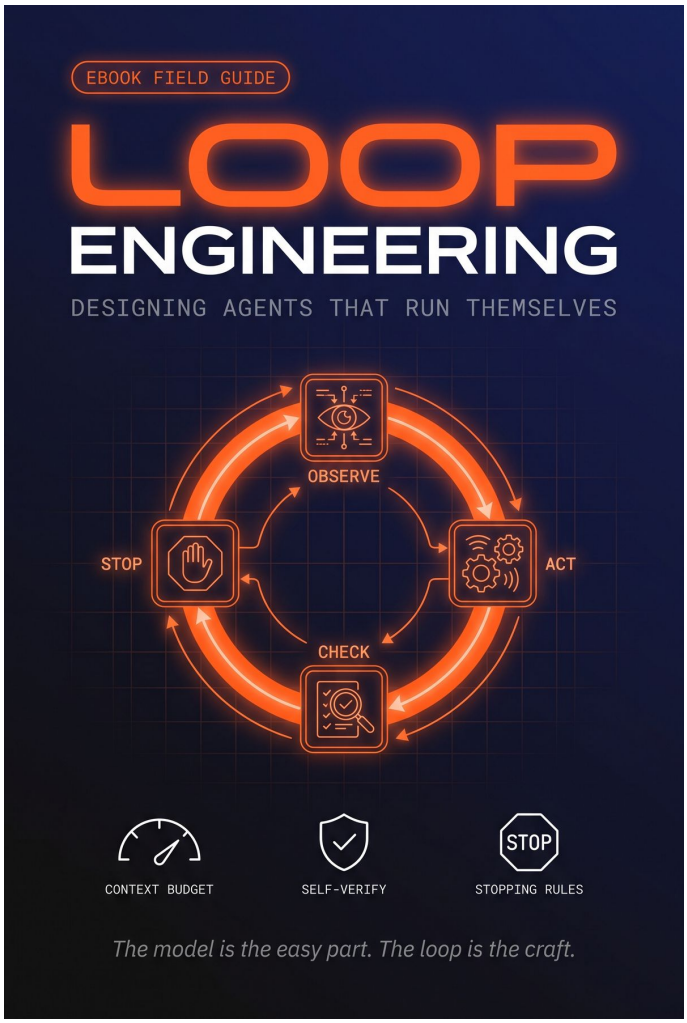

Loop Engineering: Designing Agents That Run Themselves

Everyone obsesses over which model to use. Almost nobody engineers the fifty lines of code wrapped around it — the loop that decides what the model sees, when it gets to act, when it's allowed to stop, and what happens when it does something dumb. That loop is the agent. The model is just an expensive function you call from inside it.

This is a build-along field guide to loop engineering: the discipline of designing, hardening, and shipping the control loop that turns a language model into an agent you can trust to run unattended. We start with the naive loop — the one that works in a demo and falls over in production — and harden it chapter by chapter into a small TypeScript runtime, with an on-call triage agent as the running example. Along the way: context budgeting, stopping conditions, retries and idempotency, the approval boundary, self-verification (the DOER/CHECKER split), observability, cost control, resumability, and composing loops out of sub-loops. The model is the

easy part. The loop is the craft — and it's where reliability is won or lost.

Roger Stringer · rogerstringer.com

July 11, 2026

Contents

Why the Loop Is the Real Program

Everyone obsesses over the model. Which one, how big, what it scored on some benchmark. Almost nobody talks about the fifty lines of code wrapped around it — the `while` loop that decides what the model sees, when it gets to act, when it's allowed to stop, and what happens when it does something dumb.

That loop is the agent. The model is just a very expensive function you call from inside it.

I learned this the way most people learn things that matter: by building something that worked in a demo and fell over in the real world. My first "agent" was a model call in a `while (true)`. It worked beautifully for about six turns, then it looped forever re-reading the same file, ran up a bill I actually noticed, and eventually "fixed" a bug by deleting the test that caught it. The model was fine. The model was great. The *loop* was garbage — no budget, no stop condition, no guardrail, no memory of what it had already tried.

Loop engineering is the discipline of building the part everyone skips. Not prompt engineering (what you say to the model) and not context engineering (what the model can see) — though both live inside the loop. This is the control structure itself: the cycle of gather !ḍecide !ȧct !ȯbserve !č̣heck, and every hard decision about how that cycle behaves when things go sideways.

The term caught on in 2026. Peter Steinberger put it bluntly — *"you shouldn't be prompting coding agents anymore; you should be designing loops that prompt your agents"* — and Addy Osmani gave it the name, framing a loop as six moving parts: automations, worktrees, skills, connectors, sub-agents, and memory. That's the *what*. There's a gentle, no-code way in — building your first loop with Claude Code's `/goal` command and a scheduled routine, no TypeScript required — but this guide takes the other road: we're going to *build the loop ourselves*, from scratch, and confront every hard part the managed tools handle for you — so that when you do use them, you know exactly what's happening under the hood and why it sometimes isn't.

This guide is a build-along. We start with the naive loop — the one that fails the way mine did — and harden it chapter by chapter into `agentloop`, a small TypeScript runtime you could actually put on a box and trust. The running example throughout is an **on-call triage agent**: it pulls alerts off a queue, gathers context, decides what to do, acts through tools, and moves to the next one. It's a good teacher because it hits every loop problem there is — runaway cost, dangerous side effects, transient failures, the need to stop and ask a human before it restarts production.

If you've read Roll Your Own Coding Agent (https://rogerstringer.com/guides/roll-your-own-coding-agent), some of the early scaffolding will feel familiar — that guide builds a coding agent to show you what's under Cursor and Claude Code. This one zooms out: the loop is the same shape whether the agent writes code, triages alerts, or answers support tickets, and *the loop is where reliability is won or lost.* We're going deep on the loop and nothing else.

Thirteen chapters, four parts:

- **Part 1 — The loop is the program.** Why the loop is the real unit of engineering, the anatomy of one turn, and how tools become the loop's hands.
- **Part 2 — Making the loop not suck.** Context budgeting, stopping conditions, retries and idempotency, and the approval boundary.
- **Part 3 — Trusting the loop.** Verification inside the loop, observability, and cost control.
- **Part 4 — Loops at scale.** Long-running and resumable loops, composing loops out of sub-loops, and shipping the whole thing to production.

Let's build.

Here's the reframe that changes how you build everything after it: **the model is a component, and the loop**

is the program.

A language model, on its own, does exactly one thing. You give it text, it gives you back text, and then it forgets you existed. That's it. It has no hands, no memory between calls, no ability to check its own work, no sense of when a job is done. Every capability you associate with "agents" — using tools, remembering what it tried, knowing when to stop, asking permission — is not in the model. It's in the code you wrap around the model. It's in the loop.

This is why two teams using the *identical* model get wildly different results. The one whose agent quietly resolves 40% of its queue and escalates the rest cleanly didn't get a better model. They engineered a better loop.

Say it plainly: a turn of an agent loop is five steps.

1. **Gather** — assemble what the model gets to see this turn (the task, relevant history, tool results from last turn).
2. **Decide** — call the model; it emits either a final answer or a request to use a tool.
3. **Act** — if it asked for a tool, run the tool.
4. **Observe** — feed the tool's result back in.
5. **Check** — decide whether to loop again or stop.

That's the whole thing. Everything in this guide is a hard question about one of those five steps. What do you *gather* when the history is too big to fit? How do you *check* for "done" when the model is bad at knowing it's done? What happens in *act* when the tool fails, or when the tool is `restart_production_database`?

The trap is that the naive version of this loop is genuinely easy to write — maybe fifteen lines — and it *works in a demo.* That's the dangerous part. It works well enough to convince you it's finished, then it meets reality: a task that takes forty turns instead of four, a tool that times out, a model that gets stuck in a rut repeating the same action. The demo loop has no answer for any of that, because the answers aren't in the model. They were supposed to be in the loop, and you didn't write them.

So the mental model for the rest of this guide: **you are not building an AI. You are building a control system that happens to call an AI.** That framing pulls in decades of boring, battle-tested engineering — retries, idempotency, budgets, circuit breakers, observability — and applies it to a component that happens to be a language model. The boring engineering is the point. The model is the easy part. It's already good. Your loop is what's holding it back.

If you take one thing from this chapter: stop debugging your prompts first. When an agent misbehaves, nine times out of ten the bug is in the loop — something you gathered, something you didn't stop, something you let it do — not in the words you sent the model.

Anatomy of a Loop: Building v0

Let's build the naive loop on purpose, so we know exactly what we're hardening. This is `agentloop` v0. Every later chapter fixes one thing it gets wrong.

We'll use TypeScript, one model client, and a made-up ops task: our **on-call triage agent** works a queue of alerts. For now it has one tool, `getAlert`, and its job is to look at an alert and summarize it.

The two data structures that matter are the **message list** (the running conversation, including tool results) and the **tool registry** (what the agent is allowed to do).

```
// agentloop v0 — the naive loop. Do not ship this.
type Msg = { role: "system" | "user" | "assistant" | "tool"; content: string; toolCall?:
ToolCall };
type ToolCall = { name: string; args: Record<string, unknown> };

const tools = {
  getAlert: async ({ id }: { id: string }) => db.alerts.find(id),
};

async function run(task: string) {
  const messages: Msg[] = [
    { role: "system", content: SYSTEM_PROMPT },
    { role: "user", content: task },
  ];

  while (true) {                                    // !problem #1
    const reply = await model.call(messages);       // DECIDE
    messages.push(reply);

    if (reply.toolCall) {                            // ACT
      const tool = tools[reply.toolCall.name];       // !problem #2 (no such tool?)
      const result = await tool(reply.toolCall.args); // !problem #3 (it throws?)
      messages.push({ role: "tool", content: JSON.stringify(result) }); // OBSERVE
      continue;
    }

    return reply.content;                            // CHECK: no tool call = done
  }
}
```

Read that loop against the five steps. **Gather** is "shove everything into `messages` and never remove anything." **Decide** is the model call. **Act** is `tools[name](args)`. **Observe** is pushing the tool result back. **Check** is "did the model stop asking for tools?"

Now count the ways it kills you in production, because these are literally the chapter list:

- `while (true)` with no budget. A confused model tool-calls forever and you find out from your bill. *(Ch. 5)*
- `messages` grows without bound. Turn 30 blows past the context window and the call fails — or worse, silently truncates the system prompt. *(Ch. 4)*
- `tools[reply.toolCall.name]` assumes the model named a real tool with valid args. It hallucinates `getAlertt` and you get `undefined is not a function`. *(Ch. 2/3, below)*
- `await tool(args)` assumes tools never throw and never need a retry. *(Ch. 6)*
- Nothing stops the agent before a destructive action. The moment a tool can *write*, this loop is a liability. *(Ch. 7)*
- "No tool call = done" trusts the model's judgment about completion, which is the thing it's worst at. *(Ch. 8)*
- No logs. When it does something weird, you have no idea why. *(Ch. 9)*

Before we leave v0, fix the cheapest bug — the one that's just hygiene, not a whole chapter. Never trust the tool name or that the tool succeeded structurally:

```
if (reply.toolCall) {
  const tool = tools[reply.toolCall.name];
  if (!tool) {
    // Don't crash — tell the model it was wrong and let it recover.
    messages.push({ role: "tool", content: `Error: no tool named "${reply.toolCall.name}".
Available: ${Object.keys(tools).join(", ")}` });
    continue;
  }
  // ...run it
}
```

That one pattern — **turn errors into observations the model can see and react to, instead of exceptions that kill the loop** — is the single most useful habit in loop engineering. Hold onto it; it comes back in every chapter of Part 2. A model that's told "that tool doesn't exist, here are the real ones" will just… use the right one. A model that triggers an unhandled exception takes your whole agent down.

v0 runs. v0 demos great. v0 is a trap. Let's harden it.

Tools Are the Loop's Hands

The model can't *do* anything. It can only emit text that says "I would like to do this." Tools are the bridge from that text to the actual world, and the tool boundary is where most agent bugs are born — not in the model's reasoning, but in the sloppy handoff between "the model asked for X" and "X happened."

A tool is three things: a **name**, a **schema** the model uses to call it correctly, and a **handler** that does the work. The schema is not paperwork — it's how you stop half your errors before they happen, because a good schema makes malformed calls impossible to express.

```
type Tool<A> = {
  name: string;
  description: string;         // the model reads this to decide when to use it
  schema: JSONSchema;          // args are validated against this BEFORE the handler runs
  handler: (args: A, ctx: LoopContext) => Promise<ToolResult>;
};

const getAlert: Tool<{ id: string }> = {
  name: "getAlert",
  description: "Fetch a single alert by its ID. Use before deciding any action.",
  schema: { type: "object", properties: { id: { type: "string" } }, required: ["id"] },
  handler: async ({ id }, ctx) => {
    const alert = await ctx.db.alerts.find(id);
    if (!alert) return { ok: false, error: `No alert with id ${id}` };
    return { ok: true, data: alert };
  },
};
```

Three design rules earn their keep every single time:

1. Validate args before the handler runs. The model *will* send you `{ id: 42 }` when you wanted a string, or omit a required field entirely. Validate against the schema at the loop boundary and, on failure, feed the validation error back as an observation — same pattern as Chapter 2. The handler should never receive garbage.

```
const parsed = validate(tool.schema, reply.toolCall.args);
if (!parsed.ok) {
  messages.push({ role: "tool", content: `Invalid args for ${tool.name}: ${parsed.error}
` });
  continue;   // let the model try again with the error in front of it
}
```

2. Tools return results, never throw. A tool that throws is a tool that can kill the loop. Give every tool a uniform result type — `{ ok: true, data }` or `{ ok: false, error }` — and let the loop decide what to do with a failure. The model handles `{ ok: false, error: "database timeout" }` gracefully; it cannot handle a stack trace, because it never sees one.

3. Write tool descriptions for the model, not for you. The description is a prompt. "Fetch a single alert by its ID. Use before deciding any action." tells the model *when* to reach for it. Vague descriptions produce agents that pick the wrong tool, and picking the wrong tool is the most expensive mistake in the loop — every wrong action costs a full turn (money) and pollutes the context (Chapter 4).

One more distinction that saves you later: separate **read tools** from **write tools** in your own head from day one. `getAlert`, `searchLogs`, `getRecentDeploys` are reads — safe, idempotent, cheap to retry. `restartService`, `escalateToHuman`, `resolveAlert` are writes — they change the world and demand budgets (Ch. 5), retry care (Ch. 6), and approval gates (Ch. 7). The loop should know which is which. We'll formalize it with a `sideEffect: "read" | "write"` flag on the tool and lean on it constantly.

Here's the loop with a proper tool registry — v0.5:

```
async function step(messages: Msg[], registry: Map<string, Tool<any>>, ctx: LoopContext) {
  const reply = await model.call(messages, { tools: describe(registry) });
  messages.push(reply);
  if (!reply.toolCall) return { done: true, output: reply.content };

  const tool = registry.get(reply.toolCall.name);
  if (!tool) { messages.push(obs(`Unknown tool "${reply.toolCall.name}"`)); return { done:
false }; }

  const parsed = validate(tool.schema, reply.toolCall.args);
  if (!parsed.ok) { messages.push(obs(`Invalid args: ${parsed.error}`)); return { done:
false }; }

  const result = await tool.handler(parsed.value, ctx);   // handler never throws
  messages.push(obs(JSON.stringify(result)));
  return { done: false };
}
```

Notice what we did: pulled one turn out into a `step` function. That refactor isn't cosmetic — once a turn is a value-returning function, the *loop around it* becomes the thing we engineer, and every hard decision (budget, stop, retry, approval) lives in the caller. Which is exactly where Part 2 begins.

Context Engineering Inside the Loop

The naive loop's fatal habit is that it never forgets. Every turn appends the model's reply and every tool result to `messages`, forever. By turn 30 of a real triage session — dozens of log dumps, deploy histories, half-read runbooks — you either blow past the context window (hard error) or you silently truncate and the model loses the plot without telling you.

Context is not free storage. It's a **budget you spend every turn**, and the "gather" step of your loop is where you decide how to spend it. This is the loop-side of the discipline I go deep on in the Context Engineering (https://rogerstringer.com/guides/context-engineering) guide — here we care specifically about managing it *turn to turn as the loop runs.*

The mental model: every turn, you reconstruct the window from scratch out of parts you control.

```
[ system prompt ]        !fixed, always present, cheap to keep
[ task / goal ]          !fixed, the thing it's working on
[ durable notes ]        !a running scratchpad the agent maintains (small)
[ recent turns ]         !the last N turns verbatim
[ compacted older turns ]!everything before that, summarized
[ this turn's new obs ]  !fresh tool results
```

The single highest-leverage move is a **sliding window with summarization**: keep the last N turns verbatim, and when history crosses a threshold, replace the older turns with a short summary the model writes itself.

```
async function gather(state: LoopState): Promise<Msg[]> {
  const { system, task, notes, history } = state;
  const RECENT = 8;

  if (history.length <= RECENT) {
    return [system, task, ...(notes ? [noteMsg(notes)] : []), ...history];
  }
  // Compact everything older than the recent window, once, and cache it.
  const older = history.slice(0, -RECENT);
  const recent = history.slice(-RECENT);
  state.summary ??= await summarize(older);   // model-written, cached so we pay once

  return [system, task, ...(notes ? [noteMsg(notes)] : []),
summaryMsg(state.summary), ...recent];
}
```

Two things make or break this:

Compact tool results, not just chat. In an ops agent the bloat isn't the model's prose — it's the 2,000-line log dump you fed back in during "observe." Store the raw result somewhere retrievable, but put a *reference plus a digest* in the window: `"searchLogs !2,314 lines, 3 ERROR (OOMKilled x3 on api-7). Full result: ref#a41."` The agent can ask for the full thing by reference if it needs it. You just cut 95% of the tokens with zero loss of decision-relevant information.

Give the agent a durable scratchpad. A tiny `notes` field the agent updates via a `writeNote` tool — "Hypothesis: memory leak in api-7 after the 14:02 deploy. Ruled out: DB, network." This survives compaction verbatim. It's the difference between an agent that remembers its own reasoning across 40 turns and one that re-derives it badly every time the window slides. Keep it small and force overwrites, or it becomes bloat of its own.

The rule of thumb I use: **the context window is the agent's working memory, and working memory is supposed to be small.** If you're not actively deciding what leaves the window each turn, the model is deciding for you by drowning — and a drowning model makes worse decisions two turns before you'd ever notice from the outside.

Stopping Conditions and Turn Budgets

`while (true)` is the line that funds my "things I regret" collection. A loop with no stopping condition is not an agent; it's an open-ended charge on your credit card that occasionally produces output.

Loops have to end. The catch is that there are two very different kinds of ending, and you need both:

Good endings — the agent decided it's done. The model emits a final answer with no tool call, or better, calls an explicit `finish` tool. Prefer the explicit signal: "no tool call" is ambiguous (did it finish, or did it just forget to act?), whereas a `finish({ resolution, summary })` tool is unambiguous and gives you structured output to check in Chapter 8.

Forced endings — the loop decided it's done, whether the agent likes it or not. These are your safety rails, and every single one is non-optional in production:

```
type Budget = {
  maxSteps: number;        // hard cap on turns
  maxWallClockMs: number;  // hard cap on time
  maxCostUsd: number;      // hard cap on spend (track token cost per call — see Ch. 10)
  maxToolCalls?: Record<string, number>; // per-tool caps for the dangerous ones
};

async function run(task: string, budget: Budget, ctx: LoopContext) {
  const started = Date.now();
  let steps = 0, cost = 0;

  while (true) {
    if (steps >= budget.maxSteps)                return halt("step budget exhausted");
    if (Date.now() - started > budget.maxWallClockMs) return halt("time budget
exhausted");
    if (cost >= budget.maxCostUsd)               return halt("cost budget exhausted");

    const { reply, callCost } = await step(state, registry, ctx);
    steps++; cost += callCost;

    if (reply.finish) return done(reply.finish);
  }
}
```

But raw caps aren't enough, because the most common runaway isn't "40 productive turns" — it's the agent doing **the same useless thing over and over.** Re-reading the same file. Calling `searchLogs` with the same query five times. The step budget will eventually catch it, but not before it burns twenty turns going nowhere.

So add a **loop / no-progress detector**:

```
// If the last 3 actions are identical, the agent is stuck. Interrupt it.
function detectStall(history: Action[]): boolean {
  const last3 = history.slice(-3);
  return last3.length === 3 && last3.every(a => sameAction(a, last3[0]));
}

if (detectStall(state.actions)) {
  messages.push(obs(
    "You have repeated the same action 3 times without new information. " +
    "Stop and either try a different approach or call finish/escalate."
  ));
}
```

Notice the fix isn't to kill the loop — it's to *tell the model it's stuck* and let it course-correct, escalating to a hard halt only if it keeps going. Same philosophy as everywhere: prefer an observation the agent can react to over an exception it can't.

The uncomfortable truth underneath this chapter: **models are bad at knowing when they're done.** They'll declare victory three steps early or grind three steps past the point of usefulness. Your loop's stopping logic

is not a backstop for a rare edge case — it's a core feature that runs every single turn. Budget every loop like it's going to try to bankrupt you, because the day it malfunctions, it will.

Retries, Idempotency, and Error Recovery

Tools fail. The database times out, the API 503s, the deploy service is briefly down. In the naive loop, a single transient failure either throws (killing the loop) or gets fed back as a permanent-looking error (the agent gives up on a problem that would've worked on retry). Both are wrong. Recovering from failure *inside* the loop is what separates an agent you can leave alone from one you have to babysit.

The first move is a distinction the model shouldn't have to make: **transient vs. terminal failures.**

- **Transient** — timeouts, 429s, 503s, connection resets. Retry with backoff. The agent never needs to know it happened.
- **Terminal** — "no alert with that id," "permission denied," "invalid argument." Don't retry; feed it back as an observation so the agent adapts.

Bake this into the tool-execution layer, below the model, so every tool gets it for free:

```
async function runTool(tool: Tool<any>, args: any, ctx: LoopContext): Promise<ToolResult>
{
  const RETRIES = 3;
  for (let attempt = 1; attempt <= RETRIES; attempt++) {
    try {
      return await tool.handler(args, ctx);          // returns {ok:false,...} for
terminal errors
    } catch (err) {
      if (!isTransient(err) || attempt === RETRIES) {
        return { ok: false, error: `${tool.name} failed: ${String(err)}` }; //
observation, not a crash
      }
      await sleep(backoff(attempt));                  // 250ms, 1s, 4s...
    }
  }
  return { ok: false, error: "unreachable" };
}
```

Now the part people skip and regret: **idempotency for write tools.** Retries are lovely for reads and terrifying for writes. If `restartService` times out, did the restart happen or not? Retry blindly and you might restart twice. The loop itself can also replay a write — imagine the process crashes mid-turn and resumes (Chapter 11) — and now you've escalated the same alert to PagerDuty three times at 3am. Ask me how I know.

The fix is **idempotency keys**: every write action carries a stable key, and the tool (or the service behind it) refuses to perform the same keyed action twice.

```
const escalate: Tool<{ alertId: string; reason: string }> = {
  name: "escalate", sideEffect: "write",
  handler: async ({ alertId, reason }, ctx) => {
    const key = `escalate:${alertId}`;              // stable per alert, not per attempt
    if (await ctx.store.seen(key)) return { ok: true, data: "already escalated (idempotent
no-op)" };
    await pagerduty.trigger(alertId, reason);
    await ctx.store.mark(key);
    return { ok: true, data: "escalated" };
  },
};
```

The habit to internalize: **assume every write will be attempted more than once** — by a retry, by a resumed loop, by an over-eager model — and make the second attempt harmless. This is the same discipline I hammer in Background Jobs & Queues (https://rogerstringer.com/guides/background-jobs-and-queues), and it's not a coincidence: an agent loop *is* a job queue where the model writes the jobs. Everything the queue people learned the hard way about retries and idempotency applies directly, and you get to skip learning it the hard way yourself.

A loop that retries transient failures silently, surfaces terminal ones as observations, and makes every write idempotent is a loop you can walk away from. That's the whole goal of Part 2 in one sentence.

Guardrails and the Approval Boundary

Everything so far assumed the agent's actions were, at worst, wasteful. This chapter is about the actions that are *dangerous* — the ones where being wrong isn't a wasted turn, it's a restarted production database at peak traffic. The moment a tool can change the world, your loop needs an approval boundary, and getting this boundary right is the difference between an agent you can deploy and a résumé-generating incident.

Start from the `sideEffect` flag we put on tools back in Chapter 3. It was foreshadowing. Reads run freely. Writes pass through a gate.

```
async function act(tool: Tool<any>, args: any, ctx: LoopContext): Promise<ToolResult> {
  if (tool.sideEffect === "read") return runTool(tool, args, ctx);

  // Write: classify risk and route accordingly.
  const risk = ctx.policy.assess(tool, args);   // "low" | "high"
  if (risk === "high") {
    const decision = await ctx.approvals.request({ tool: tool.name, args, reason:
ctx.lastReasoning });
    if (decision.status !== "approved") {
      return { ok: false, error: `Action ${tool.name} was ${decision.status} by a human.
Do not retry; choose another path.` };
    }
  }
  return runTool(tool, args, ctx);
}
```

The three patterns that make this real:

1. Plan-then-act. For anything consequential, make the agent *propose the full plan before executing any of it.* A `finish`-style `proposePlan` tool emits the ordered list of write actions; a human (or a policy) approves the plan; only then does the loop execute. This beats approving actions one-by-one because the reviewer sees the whole blast radius at once — "restart api-7, then clear the cache, then resolve the alert" reads very differently as a plan than as three separate pings.

2. Dry-run by default. Give dangerous tools a dry-run mode and run it first. `restartService({ id, dryRun: true })` returns "would restart api-7, currently serving 1,240 req/s" — the agent (and the human) sees the consequence before committing. Half the time the dry-run output changes the decision.

3. The approval boundary is asynchronous, and your loop must handle that. A human isn't going to answer in 200ms. The loop has to be able to **pause, persist its state, and resume when the approval arrives** — which is exactly the resumability machinery from Chapter 11, and the reason these chapters are ordered the way they are. In v0 you can block; in production you serialize the loop state, fire a notification, and rehydrate when the human clicks approve.

One principle sits above all the mechanics: **the agent should never be one hallucination away from a catastrophe.** Don't rely on the prompt to keep it safe ("please don't restart production without asking") — prompts are suggestions, and a model that misreads the situation will cheerfully ignore them. Rely on the loop. The gate is code. Code doesn't get talked out of things.

This connects directly to the safety posture in [Building Your Agentic OS](https://rogerstringer.com/guides/building-your-agentic-os): the agent can be trusted with reads and low-risk writes precisely *because* the loop makes high-risk writes structurally require a human. You earn autonomy by making the dangerous stuff impossible to do unilaterally — not by hoping the model behaves.

Verification Inside the Loop

Back in Chapter 5 I said models are bad at knowing when they're done. This chapter is the fix: **stop trusting the agent's word that it finished, and make the loop check.**

Here's the line I keep coming back to: **if you can't say what "done" looks like, you don't have a loop — you have a wish.** Every loop has a **DOER** (the part that makes the thing) and a **CHECKER** (the part that decides if it's good). The model — the DOER — was never the hard part. The CHECKER is. Almost everyone builds the DOER and skips the CHECKER, then wonders why their agent is confidently wrong.

The naive loop's stopping condition — "the model stopped asking for tools" — is faith-based engineering. The model says "I've resolved the alert," and you believe it. Sometimes it did. Sometimes it marked an alert resolved without doing the thing that resolves it, because it *pattern-matched to done* rather than *verified done.* An agent that can lie to itself about completion will lie to you about completion. That's the DOER grading its own homework, and the DOER always grades generously.

The move is to put a verification step inside the loop, between the agent claiming done and the loop accepting it. I call it the check gate. When the agent calls `finish`, don't return — **verify the claim, and if it fails, feed the failure back and keep looping.**

```
if (reply.finish) {
  const verdict = await verify(reply.finish, state, ctx);
  if (verdict.ok) return done(reply.finish);

  // Rejected: the agent isn't actually done. Tell it why and continue.
  messages.push(obs(`Not done yet: ${verdict.reason}. Address this before finishing.`));
  continue;
}
```

`verify` can be as cheap or as rigorous as the stakes demand:

- **Assertions.** Deterministic checks. Agent claims the alert is resolved? Query the alert's status directly — is it actually `resolved`? Claims it restarted api-7? Check the service's start time is recent. These are the highest-value checks because they can't be argued with. Code doesn't hallucinate.
- **A verifier model call.** A second, cheap model call whose only job is to judge: "Given the task and the transcript, is this actually complete? Answer with a reason." A fresh model that didn't do the work is a much harsher grader than the one that's motivated to be finished.
- **An eval gate.** For agents that run a known class of task, run the same evals you'd run in CI, *inside the loop.* This is where loop engineering meets Eval-Driven Development (https://rogerstringer.com/guides/eval-driven-development) head-on: the evals you write to test the agent offline are the same checks you run online to gate completion. Write them once, use them in both places.

This is the **DOER/CHECKER split** made concrete: the part of the loop that *does the work* and the part that *checks the work* should be separate, and ideally not the same context talking to itself. Give the CHECKER a clean slate, the original task, and the result — nothing else — and it catches things the DOER was blind to precisely because it isn't invested in being done. I run exactly this on a real support agent: one loop auto-resolves the day's tickets, and a separate checker agent independently reviews each closure and *re-opens* any that should've gone to a human instead. The checker isn't a nicety bolted on top — it's the half of the system that makes auto-resolution safe to turn on at all.

Verification costs turns and money, so scale it to the stakes: a read-only summarization agent barely needs it; an agent that resolves production alerts needs a hard assertion gate on every `resolve`. But the principle holds across all of them — **"done" is a claim to be verified, not a state to be trusted.** The loop is where you verify it, because the loop is the only part of the system that isn't the model.

Observability: Making the Loop Legible

The naive loop is a black box. It does forty turns, produces an answer, and when the answer is wrong you have exactly one debugging tool: reading the final output and guessing. That's not good enough for anything you run unattended. **If you can't replay what the loop did, you can't fix it — you can only pray it doesn't happen again.**

Agent loops are genuinely hard to observe because the interesting state isn't in any one place — it's the *sequence*: what the model saw, what it decided, why, what the tool returned, how the context changed. So you instrument the loop to emit a structured trace of every turn. Not logs you `grep`. A timeline you can replay.

The unit is the **turn record**:

```
type TurnRecord = {
  loopId: string;
  step: number;
  timestamp: string;
  contextTokens: number;              // what the window cost this turn
  modelInput: Msg[];                  // exactly what the model saw (post-gather/
compaction)
  reasoning?: string;                 // the model's stated rationale, if it emitted one
  action: { tool: string; args: unknown } | { finish: unknown };
  toolResult?: ToolResult;
  costUsd: number;
  durationMs: number;
};
```

Emit one of these every turn, to somewhere durable, and three things become possible that were impossible before:

Replay. You can reconstruct the exact sequence that led to a bad action — see precisely what the model saw at step 22 when it decided to escalate. Nine times out of ten the bug is obvious the moment you can see the input: the context got compacted too aggressively and dropped the fact that ruled the action out, or a tool returned something malformed and the model ran with it. You stop guessing.

Live legibility. For anything with an approval boundary (Chapter 7), a human needs to see *why* the agent wants to do the scary thing. `reasoning` + the recent turns rendered as a timeline turns "the agent wants to restart api-7" into "the agent wants to restart api-7 *because it found three OOMKills after the 14:02 deploy and confirmed memory climbing* — approve?" Same action, completely different decision quality for the reviewer.

Aggregate metrics. Once every turn is a record, you can ask questions across thousands of runs: average turns-to-resolution, cost per run, which tool fails most, where loops stall (Chapter 5's detector, but now you can *see* the stalls). This is what tells you whether last week's prompt change actually helped or just felt like it did.

One concrete tip that pays for itself: **log the post-gather `modelInput`, not the raw history.** The bug is almost always in what the model *actually saw* after your context engineering did its thing — the compaction, the sliding window, the reference-swapping from Chapter 4 — not in the raw event log. If you only log the raw history, you're debugging a different program than the one that ran.

You don't need a fancy platform to start. A turn record appended to a JSONL file and a fifty-line HTML viewer that renders the timeline will take you remarkably far — the same "boring and legible beats clever and opaque" instinct from The Boring Stack (https://rogerstringer.com/guides/the-boring-stack). The goal isn't dashboards. It's that when the loop does something baffling at 3am, you can sit down in the morning and *watch it happen.*

Cost and Token Control

Every turn of the loop is a model call, and every model call is money. The naive loop treats tokens as free, which is how a "cheap little agent" quietly becomes a line item you have to explain. Loop engineering is where cost is actually controlled — not by picking a cheaper model, but by spending fewer, smaller turns. You already built most of the levers in earlier chapters; this chapter is about pointing them at the bill.

First, **measure per turn, or you're flying blind.** You cannot control what you don't count. Track token cost on every call and thread it through the loop — it's the same `callCost` that fed the budget in Chapter 5 and the `costUsd` on the turn record in Chapter 9. That's not a coincidence; cost control reuses the machinery you already have.

```
function priceOf(usage: Usage, model: ModelSpec): number {
  return usage.inputTokens  * model.inputPricePerTok
       + usage.outputTokens * model.outputPricePerTok;
}
```

Now the levers, roughly in order of payoff:

1. Shrink the input. In a multi-turn loop, input tokens dwarf output tokens — you re-send the growing context *every single turn.* This is why Chapter 4 is secretly the most important cost chapter in the guide: aggressive context compaction doesn't just keep the model sharp, it's the biggest line-item reduction you have. Every 2,000-line log dump you swap for a 40-token digest saves that cost *on every subsequent turn*, not just once.

2. Cache the fixed prefix. Your system prompt and tool definitions are identical every turn. Most providers offer prompt caching that charges a fraction for the cached prefix. Structure the window so the stable stuff (system prompt, tool schemas, task) comes first and the volatile stuff (recent turns) comes last — cache hits on the prefix can cut input cost dramatically for free. Loop structure directly determines cache efficiency.

3. Route by difficulty. Not every turn needs your best model. "Summarize this log dump" and "decide whether to restart production" are wildly different in stakes, and you can route them to different models. The verifier from Chapter 8, the summarizer from Chapter 4 — those are cheap-model jobs. Reserve the expensive model for the actual decisions.

```
function pickModel(intent: TurnIntent): ModelSpec {
  if (intent === "summarize" || intent === "verify") return CHEAP;   // small, fast
  return CAPABLE;                                                     // the decision-
maker
}
```

4. Cap it, always. The `maxCostUsd` budget from Chapter 5 is your last line of defense. Set it per run *and* track spend per agent per day. A bug that turns a $0.10 run into a $0.10 loop-forever run is caught by the budget; a bug that turns every run slightly more expensive is caught by the daily aggregate from Chapter 9. You want both.

The framing that keeps this sane: **cost is a design constraint, not an afterthought.** The cheapest agent isn't the one on the cheapest model — it's the one with the tightest loop: minimal context, cached prefixes, cheap models for cheap jobs, hard budgets, and no wasted turns. Every one of those is a loop decision. Get the loop right and the bill takes care of itself; get it wrong and no model is cheap enough to save you.

Compaction and Long-Running Loops

Everything so far assumed the loop runs start to finish in one process, in one sitting. Real agents don't get that luxury. The triage agent runs for hours. It waits on a human approval (Chapter 7) that arrives twenty minutes later. The process gets deployed over, the machine reboots, the run crashes at turn 34. A loop that can't survive any of that isn't production-ready — it's a demo that hasn't met an outage yet.

The requirement is **resumability**: the loop's entire state must be serializable, so you can stop at any turn boundary and pick up exactly where you left off. This is why, all the way back in Chapter 2, we pulled a turn into a `step` function operating on an explicit `state` object instead of hiding state in local variables and a call stack. Local variables can't be serialized. An explicit state object can.

```
type LoopState = {
  loopId: string;
  task: string;
  system: Msg;
  history: Msg[];         // full event log (source of truth)
  summary?: string;       // cached compaction (Ch. 4)
  notes?: string;         // durable scratchpad (Ch. 4)
  steps: number;
  costUsd: number;
  status: "running" | "awaiting_approval" | "done" | "halted";
  pendingApproval?: { tool: string; args: unknown; key: string };
};
```

Everything the loop needs lives in that object. Which means the loop becomes: **load state → run one turn → persist state → repeat.**

```
async function tick(loopId: string, ctx: LoopContext) {
  const state = await ctx.store.load(loopId);
  if (state.status !== "running") return state;      // paused/done/halted — nothing to do

  const result = await step(state, ctx.registry, ctx);
  applyBudget(state, ctx.budget);                    // may flip status to "halted"
  await ctx.store.save(state);                       // ←durable at every turn boundary
  return state;
}
```

Now the hard-won details:

Persist at turn boundaries, never mid-turn. A turn is your atomic unit. If you crash mid-turn, resume from the *start* of that turn — which is safe only because your writes are idempotent (Chapter 6). This is the payoff for the idempotency work: a resumed loop *will* occasionally replay a turn, and idempotency is what makes the replay harmless instead of a double-escalation at 3am.

Compaction is now permanent, not just a context trick. In a long-running loop the summary from Chapter 4 becomes durable memory — the agent's record of the first 100 turns it will never see verbatim again. Treat it with care: a lossy summary means a lossy agent. When something decision-critical happens, push it to the durable `notes` scratchpad, which survives compaction verbatim, rather than trusting it to survive summarization.

Pausing is just a status. The approval boundary from Chapter 7 is trivial once state is durable: set `status: "awaiting_approval"`, save, stop ticking. When the human approves, flip back to `running` and resume. The loop doesn't hold a thread open for twenty minutes — it holds a row in a database. A thousand agents can be "waiting" at once because waiting costs a row, not a process.

This is the chapter where an agent stops being a script you run and becomes **a durable process that happens to think** — one you can deploy over, scale horizontally, and leave running for days. The loop became a state machine. That's not extra complexity for its own sake; it's the minimum shape that survives

contact with production.

Composing Loops

One loop, no matter how well-engineered, has a ceiling: it does one thing at a time, and its context is a single stream. When the triage agent has forty alerts waiting and each needs its own multi-turn investigation, a single loop grinds through them one by one, and every alert's logs pollute the next alert's context. The answer isn't a bigger loop. It's **more loops** — composed.

The core move is the **sub-loop**: a turn in the parent loop that spawns a fresh child loop with its own task, its own budget, and — crucially — **its own clean context.** The child runs to completion and returns only its result to the parent. The parent never sees the child's forty turns of log-diving; it sees "api-7: resolved, OOM after 14:02 deploy, restarted." This is the single most powerful pattern in the chapter, because clean context isolation is what keeps composed agents from drowning each other.

```
const investigateAlert: Tool<{ alertId: string }> = {
  name: "investigateAlert", sideEffect: "read",
  handler: async ({ alertId }, ctx) => {
    // Spawn a child loop with its OWN state, budget, and context.
    const child = await runLoop({
      task: `Investigate alert ${alertId}. Return root cause + recommended action.`,
      budget: { maxSteps: 15, maxCostUsd: 0.50, maxWallClockMs: 60_000 },
      registry: readOnlyTools,        // child can look but not touch
    }, ctx.fork());
    return { ok: true, data: child.output };   // only the conclusion crosses the boundary
  },
};
```

Two shapes cover almost everything:

Orchestrator / worker. A parent loop that plans and delegates; child loops that execute. The parent triage agent decides *which* alerts matter and in what order; a worker sub-loop investigates each one. The parent stays strategic and small; the workers get dirty in the details. Clean separation of "deciding what to do" from "doing it."

Parallel fan-out. When sub-tasks are independent, run the children *concurrently* and collapse a ten-alert backlog from ten sequential investigations into one wall-clock batch. This is where composition earns its cost — throughput.

```
const results = await Promise.all(
  alerts.map(a => runLoop({ task: `Investigate ${a.id}...`, budget: CHILD_BUDGET,
registry: readOnlyTools }, ctx.fork()))
);
```

But composition is where the failure modes get spicy, so a few hard rules:

- **Budgets nest.** The parent's budget must account for its children, or you've just built a fork bomb with a credit card. A child that can spawn children needs a depth limit. Unbounded recursion is even more fun when each call costs money.
- **Isolate write authority.** Give sub-loops read-only tools by default (note `readOnlyTools` above). Writes and approvals belong to the parent, or to a human — you do not want six parallel children all deciding to restart the same service. Concentrate side effects; distribute reads.
- **Results cross the boundary; context does not.** The entire benefit evaporates if you dump the child's full transcript back into the parent. Return the conclusion, keep the mess.

This is the on-ramp to full multi-agent systems, which is a whole guide of its own — the one I'm building toward next. Everything there (specialized agents, shared memory, orchestration) is *this* pattern with more structure and more governance. But the foundation is unglamorous and worth saying plainly: **a multi-agent system is just well-composed loops, and if your single loop isn't solid, composing it only multiplies the**

mess. Get one loop right first. Then make more of them. It connects straight back to the multi-agent direction teased in Building Your Agentic OS (https://rogerstringer.com/guides/building-your-agentic-os) — composition is the bridge from one agent to many.

Shipping a Loop to Production

You've got a loop that budgets itself, manages its context, retries transient failures, gates dangerous writes behind humans, verifies its own work, logs every turn, controls its cost, survives restarts, and composes into sub-loops. Now you have to *run* it somewhere, unattended, and sleep at night. This chapter is the operational layer — the difference between a loop that works on your laptop and one that works at 3am when you're not watching.

Where the loop runs. A durable, resumable loop (Chapter 11) wants to run as a **worker off a queue**, not a web request. Alerts land on a queue; workers pull them, run the loop tick by tick, persist state, and pick up the next. This is the exact architecture from Background Jobs & Queues (https://rogerstringer.com/guides/background-jobs-and-queues), and it's not a coincidence I keep saying that — **an agent runtime is a job system where the model decides the next job.** Everything that guide teaches about workers, concurrency, and graceful shutdown applies unchanged. Deploy it the boring way: a worker process on a single VPS, exactly like The Boring Stack (https://rogerstringer.com/guides/the-boring-stack). You don't need Kubernetes to run an agent. You need a queue, a process manager, and a database for loop state.

The pre-flight checklist. Before a loop goes near production, walk this list — every item is a chapter you already built:

- **Budgets set and enforced** — step, wall-clock, and cost caps, per run and per day. *(Ch. 5, 10)*
- **Every write tool is idempotent** — because resume and retry *will* replay them. *(Ch. 6)*
- **High-risk actions gated** — approval boundary live, not just a prompt asking nicely. *(Ch. 7)*
- **A verification gate on completion** — "done" is checked, not trusted. *(Ch. 8)*
- **Turn-level tracing on** — you can replay any run before you need to. *(Ch. 9)*
- **State persists at turn boundaries** — kill the process any time and lose nothing. *(Ch. 11)*
- **A kill switch** — one flag that halts a running loop, and a global one that halts all of them.

That last one deserves its own sentence: **build the stop button before you build anything fancy.** The ability to instantly halt a misbehaving agent — a status flip the running loop checks each tick — is the cheapest insurance you'll ever write, and the one you'll be most grateful for the day you need it.

The failure playbook. When a loop misbehaves in production — and it will — the drill is: *halt* (kill switch), *replay* (turn trace from Chapter 9 to see what it saw), *diagnose* (context bug? tool bug? missing guardrail?), *fix the loop* (not usually the prompt), *add the check that would've caught it* (an assertion, a tighter budget, a new gate). This is ordinary incident response — the same muscle from any on-call rotation — and treating agent incidents as *engineering* incidents rather than "the AI did a weird thing" is most of the maturity.

Here's the thing I want you to walk away with. **A production agent is a boring, well-instrumented control system that calls a language model.** The model is the exciting part and the *least* of your engineering. Everything that determines whether you can trust the thing — the budgets, the idempotency, the gates, the verification, the tracing, the resumability — lives in the loop. That's the craft. Not prompting the model better. Engineering the loop around it so well that a fallible model, wrapped in a loop that assumes it's fallible, adds up to a system you can actually rely on.

Two skills decide how far this takes you, and they're the two halves of the same formula I keep on a sticky note: **AI Leverage = Your Clarity × Your Skill.** *Clarity* is your ability to define what "done" actually looks like — the CHECKER from Chapter 8, the thing most people can't articulate and so never build. *Skill* is your ability to read what the loop did and make it better — the tracing from Chapter 9 turned into judgment. A model can't give you either. They're what *you* bring to the loop, and they're why the loop, not the model, is the craft.

Build the loop like the model is going to try to embarrass you. Then it can't.

About Roger

I'm Roger Stringer — I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I take on a handful of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) engagements — helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. If you're wrestling with the kind of problem this guide covers and want someone in your corner who's done it before, that's exactly what I help with. Drop me a line. (mailto:roger.stringer@hey.com)

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email** — roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X** — @freekrai (https://x.com/freekrai)
- **GitHub** — github.com/freekrai (https://github.com/freekrai)
- **LinkedIn** — linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting — follow along wherever suits you.