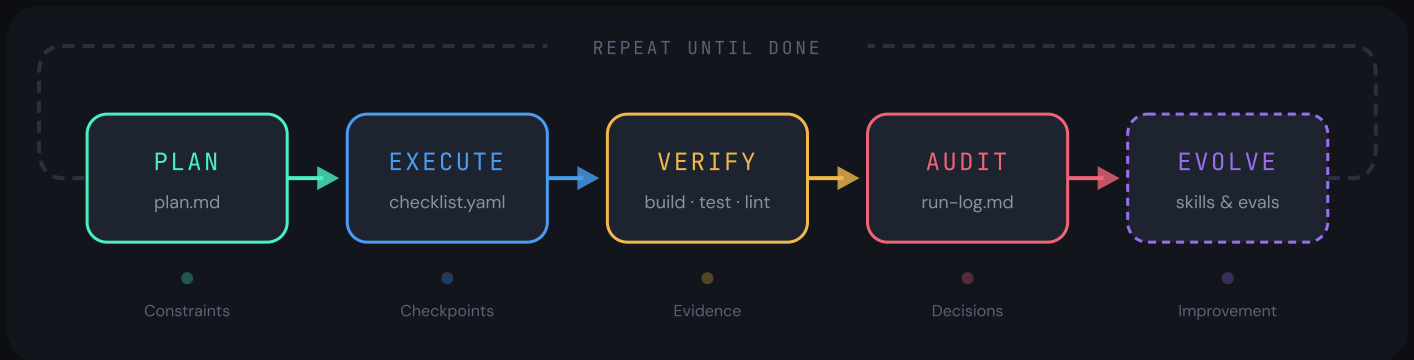


PLAYBOOK V3 • FEBRUARY 2026

Long-Running *Agents*

An operating model for handing off real engineering work to autonomous coding agents—and shipping the result safely.

Commits • Tests • Diffs you can ship



THE SHORT VERSION — IF YOU ONLY REMEMBER A FEW THINGS

- 1 **Start with Codex CLI + GPT-5.2 (high/xhigh)** — this is the current frontier. Use other stacks when you have a clear reason.
- 2 **Make “done” measurable.** A build-verify loop is non-negotiable.
- 3 **Give the agent durable memory.** `plan.md`, `checklist.yaml`, and `run-log.md`.
- 4 **Keep the loop cheap.** Fast tests often, full suite at milestones, CI always running in the background.
- 5 **Checkpoint constantly.** Small commits, reversible steps, separate checkouts for parallel work.
- 6 **Steer early.** A 30-second correction can save hours.
- 7 **Review like an operator.** Agent-first review, then human review by risk.

Contents

01	Who this is for	13	Parallel task management
02	How to use this playbook	14	Running this in a team
03	What a long-running agent is	15	Skills, MCP, and CLI tools
04	Where they shine	16	Planning
05	Where they are a bad fit	17	Templates
06	The operating system	18	Execution
07	Common failure modes	19	Compaction
08	Models	20	Steering
09	Harnesses (agent runners)	21	Review
10	Tracking the frontier	22	Conclusion
11	Cost management	23	Glossary
12	Safety and permissions	24	References

Who This Is For

This playbook is written for three audiences: **founders and execs** who want more engineering throughput, **engineers and tech leads** running large refactors, migrations, and platform work, and **platform teams** hardening CI, tests, and developer experience.

How to Use This Playbook

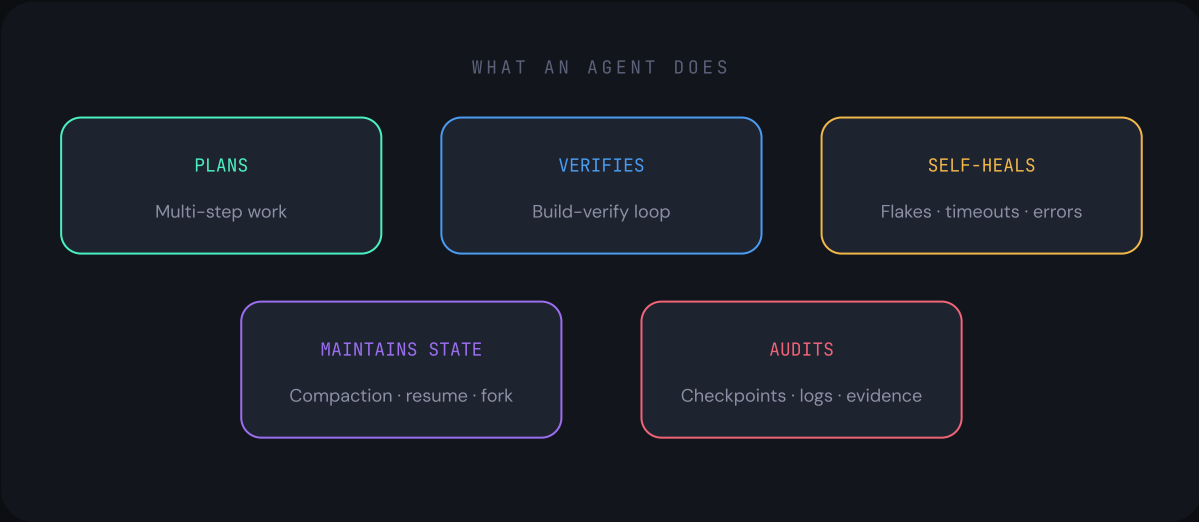
1. Skim **The Operating System, Models, and Cost Management** so you pick a stack and a workflow.
2. Copy the templates into your repo.
3. Run one small long job first, in a safe environment, with a strict verification loop.
4. Only then scale to overnight runs and parallel worktrees.

What a Long-Running Agent Is

A long-running agent is a coding agent that can run for **hours or days** with minimal intervention while it plans and executes multi-step work, keeps score with a build-verify loop, self-heals when it hits failure, maintains state across a long session, and leaves a paper trail you can audit.

Cursor shared runs where agents worked close to a week on a single codebase. One example: a web browser built from scratch with 1M+ lines across about 1,000 files. Another: a Java LSP with 2.5M+ lines across about 10,000 files. They also reported peaking around **10 million tool calls** in a week, with bursts around 1,000 commits per hour.

This playbook is an operating model for how I achieved similar results. It is opinionated. It assumes you want throughput and you still want to ship safely: green CI, reviewable diffs, rollback paths, and no surprises in production.



Where They Shine

Long-running agents are a great fit for work that is **wide, mechanical, and verifiable**:

- **Large refactors** — API changes, renames, file moves, staged migrations
- **Migrations** — framework upgrades, dependency bumps, schema changes and backfills
- **Reliability work** — harden CI, shrink flakes, add guardrails
- **Performance work** — benchmarks, profiling loops, regressions, perf budgets
- **Test work** — build suites, fix flakes, raise coverage where it matters
- **Security hygiene** — dependency audits, secret scanning, permissions cleanup
- **Data analysis** — repeated queries, synthesis, polished writeups
- **Feature work** — wide-but-shallow features with crisp specs, usually behind a flag
- **Internal tools** — dashboards, admin panels, backoffice workflows with clear “done”
- **Incident work** — log triage, repros, mitigations, postmortems and follow-ups

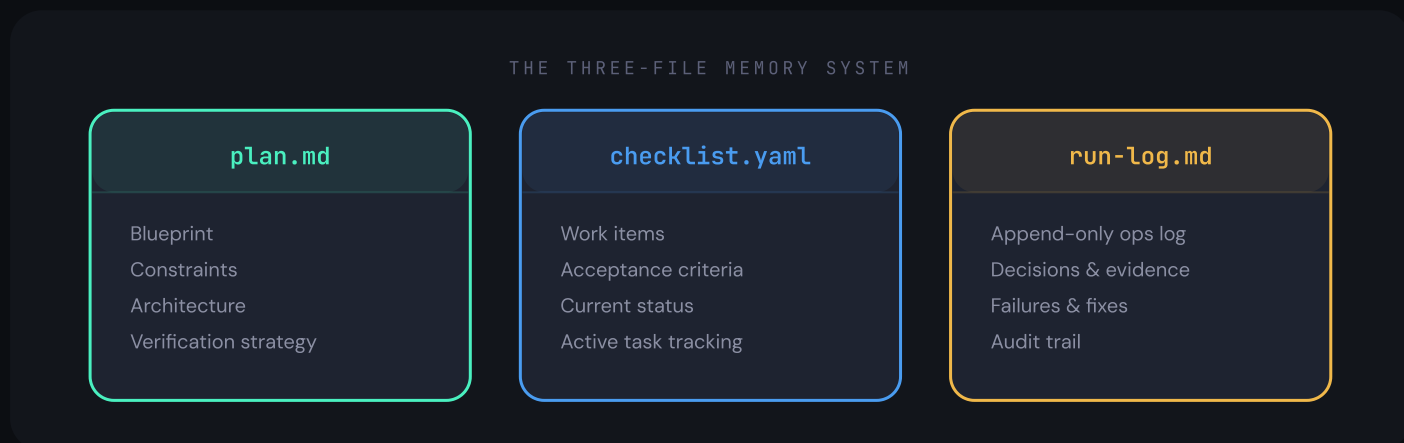
Where They Are a Bad Fit

Prefer interactive work when:

- The problem is ill-defined and you need rapid back-and-forth
 - The best solution requires taste and fast iteration (UX polish, naming, API design)
 - The acceptance criteria is subjective, not measurable
 - The blast radius is high and the rollback story is weak
 - The work requires frequent human decisions (product tradeoffs, stakeholder input)
-

The Operating System

Long-running agents succeed when you treat them like **production systems**. You need a build-verify loop, a durable memory system, checkpointing, steering, and review workflows that scale.



The three-file memory system

For long runs, keep three files at the root of the repo. These files do two jobs: **durable memory** (they survive compaction and session forks) and **observability** (they let you understand what the agent is doing without reading a 12-hour chat log).

Common Failure Modes

These are the problems that waste the most time. Each one has a recognizable pattern and a fix.

01 No objective verification

SYMPTOMS

The agent keeps “making progress” but you can’t prove it. You find out it broke something at the end.

FIX

Make “done” measurable. Tests, lint, typecheck, CI, screenshots, queries. If the repo doesn’t have a verification loop, **build that first**.

02 Flaky or slow verification loops

SYMPTOMS

The agent keeps retrying tests. It takes 30 minutes to learn one thing.

FIX

Create a fast loop for iteration, then run the full suite at milestones. Run targeted test subsets. Prefer persistent test environments when setup time is the bottleneck. Quarantine flakes—make them someone’s problem, or the agent will burn hours.

03 Tool output floods the context

SYMPTOMS

The agent pastes huge logs into chat. Compaction triggers constantly and quality drops.

FIX

Write long outputs to files (`logs/` , `tmp/` , `artifacts/`) and reference them. Teach the agent to summarize, not paste.

04 Context drift after compaction

SYMPTOMS

It forgets a constraint and starts changing forbidden areas. It redoes work it already did.

FIX

Use the three-file memory system. Make rereading those files part of the execution loop, especially after compaction.

05 Infinite loops and false progress

SYMPTOMS

It keeps trying the same fix. It bounces between two approaches.

FIX

Add a loop breaker rule: after N failed attempts, stop and change strategy. Force a short diagnosis before continuing. Use stop hooks to prevent “give up” behavior.

06 Oversized diffs

SYMPTOMS

One PR has everything. Review is impossible.

FIX

Commit checkpoints early and often. Prefer milestone commits or stacked PRs. If the end state is right but the commit history is messy, rebuild the branch with a clean narrative history.

Models

What you want in a long-running model

Look for strong tool use (especially shell output parsing), endurance on long boring sequences, good recovery after failures, refactor discipline, and low hallucination rate around file paths, APIs, and CLI usage.

Point-in-time take (2026-02-20)

If you want one default: **Codex CLI + GPT-5.2 (high or xhigh)**. This is the frontier stack right now. It runs tool loops for hours, compaction stays coherent, and you get enough visibility to steer before the run drifts.

Claude notes (Opus 4.6)

Opus 4.6 is strong and still useful for interactive work. For long-running background runs, it's not the default. Cursor put it bluntly:

"Opus 4.5 tends to stop earlier and take shortcuts when convenient."

— Cursor, Scaling long-running autonomous coding

If you're in a Claude-first stack, you can still run long jobs. You'll usually want a stronger harness loop (hooks and a "Ralph loop"), plus the same fundamentals in this playbook.

Reasoning effort (thinking levels)

Most serious harnesses expose a reasoning effort setting like `low`, `medium`, `high`, `xhigh`.

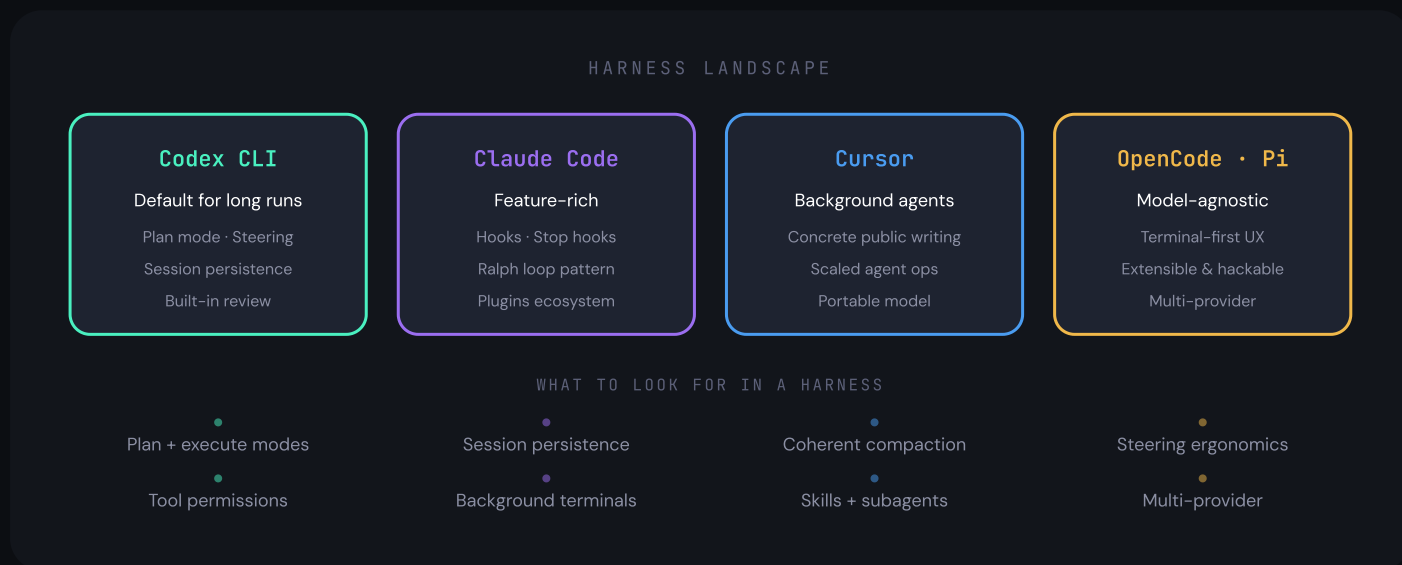
Practical rule: **plan with high or xhigh, execute with medium by default**. If execution feels slow, switch to a faster model before you drop effort.

Open-weight models as execution engines

Open-weight models are now good enough to matter. With a strict verification loop, they can do a lot of the execution work at a fraction of the cost. Plan and review with a frontier model at higher effort. Execute with an open-weight or cheaper model once the plan and test loop are locked.

Harnesses

A harness is the software around the model. It is the difference between “chat with a model” and “operate an agent.”



Codex (CLI)

This is the default harness for long-running coding work. Tool loops are the default—it runs tests, builds, and scripts for hours. Sessions live on disk so resume and fork are cheap. Compaction stays usable with good memory files. Thinking blocks plus tool traces make steering effective.

Claude Code

Claude Code is one of the most feature-rich harnesses today. To make it behave like a long-running runner, you need a “keep going” loop: **hooks** (scripts that run on lifecycle events), **stop hooks** (triggered when the agent is about to stop), and the **Ralph loop** (a stop-hook pattern that continues or restarts the run). If running Opus for hours, external memory is the crux.

Cursor background agents

Cursor has some of the most concrete public writing on long-running agents at scale. The operating model is portable: treat agent runs like running software. Monitor them, recover from errors, and expect loops that can span a full day.

OpenCode & Pi

OpenCode offers a model-agnostic harness with a great terminal UI, multiple providers, granular permissions, plugins, subagents, and a server mode for automation. Pi is small and extensible—a good fit when you want to bake in your own workflows and memory system.

Tracking the Frontier

This space moves fast. The meta-skill is switching stacks without chaos. Keep a default stack, follow release notes and key people, maintain a small “agent eval” with 10–20 real tasks in your repo, and rerun it when a new model drops.

Key metrics to track

- Time to first green fast loop
- Interventions (how often you had to steer)
- Retry storms (flakes, timeouts, tool failures)
- Cost
- Diff quality: noise, correctness, reviewability, safety

People to follow

- Peter Steinberger (steipete)
- Simon Willison
- Paul Gauthier (Aider)
- Cursor engineers and power users
- The SWE-bench and agent-eval community

Cost Management

Long-running agents can burn a lot of tokens. Set a budget. Pick a routing strategy. Then run your verification loop often so you don't pay to learn the same thing twice.

A real run (token scale)

METRIC		TOKENS
Total		52,146,126
Input		46,053,693
Cached input		2,164,821,248
Output		6,092,433
Reasoning		3,542,400

Cached input is the story. That's repeated context. Prompt caching changes the economics by an order of magnitude.

Cost comparison (this run)

OPTION	INPUT \$/1M	CACHED \$/1M	OUTPUT \$/1M	EST. COST
ChatGPT Pro	—	—	—	\$200
GPT-5.2	\$1.75	\$0.175	\$14.00	\$545
Claude Opus 4.6	\$5.00	\$0.50	\$25.00	\$1,465
Kimi K2.5	\$0.60	\$0.10	\$3.00	\$262
GLM-4.7	\$0.60	\$0.11	\$2.20	\$279
GLM-5	\$1.00	\$0.20	\$3.20	\$499

ChatGPT Pro vs Claude Max

If you want to run long background agents today, **ChatGPT Pro is the best subscription** (\$200/month as of Feb 2026). The limits have been extremely generous in practice for long-running Codex work.

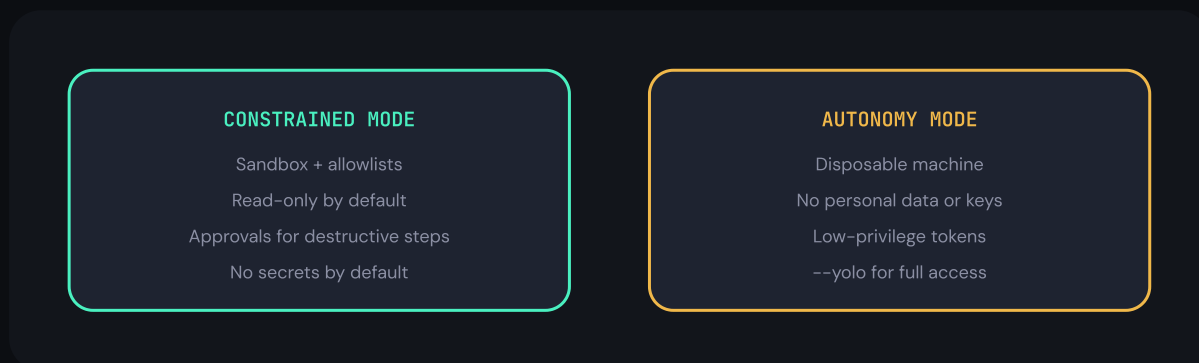
In one refactor run that consumed 52M+ tokens (and 2B+ cached tokens), about 35% of the weekly limit was still remaining on Pro. Claude Max has strict five-hour windows and message caps—fine for interactive work, a bottleneck for multi-hour background runs.

Where to spend tokens

Spend generously on planning and design, writing tests and hardening verification, debugging failures, and review and risk analysis. **Be stingy with** repetitive mechanical edits, long tool output pasted into chat, and re-running expensive steps without narrowing the problem.

Safety and Permissions

Long-running agents work because they can run unattended. That also means they can **fail unattended**. There are two sane ways to operate.



Secrets and data

Never paste secrets into prompts. Use dedicated low-privilege credentials, prefer short-lived tokens, hydrate secrets via a secrets manager, treat tool output as log data, and add secret scanning to CI.

Isolation options

If you want full autonomy without fear: run agents in a fresh VM, use a disposable dev environment platform (Daytona, sprites.dev, Codespaces, or your own VM templates), and keep clean separation between agent workspaces and your real machine.

Parallel Task Management

Parallel agents are a force multiplier when the work streams are independent. Give each agent its own checkout. The rules: **one agent per checkout, one goal per agent, one PR per checkout.**

```
# Worktrees (fast, shared .git)
git worktree add ../repo-agent-refactor -b agent/refactor
git worktree add ../repo-agent-migration -b agent/migration

# Separate clones (cleaner isolation)
git clone "$(pwd)" ../repo-agent-review
git clone "$(pwd)" ../repo-agent-perf
```

Avoid parallelizing interdependent work. When tasks depend on each other, you get merge conflicts, conflicting design decisions, and two agents fixing the same test failures in different ways. The hidden cost is lost learning. Sequential wins here —let one agent carry the context forward.

Checkpointing via git

Ask the agent to commit early and often. You get: a rollback path, easier review for big diffs, and a commit story you can scan quickly.

Clean-commit (narrative rebuild)

When agents land the right end state with a messy branch, rebuild it with a clean commit storyline: validate the source branch, study the full diff, create a new branch off `main`, plan a commit storyline, reimplement commit-by-commit, and verify the final state matches.

Tools that help

- [try](#) — a CLI for organizing experiments and creating dated worktrees
 - [Conductor](#) — a UI for coordinating parallel tasks and agent runs
-

Running This in a Team

Long-running agents work best when the team agrees on a few defaults:

- Standardize the three memory files across repos
- Standardize your safety baseline (sandbox by default, “yolo” only in isolated environments)
- Build a small shared skill library for your stack (CI triage, migrations, releases, clean-commit, review)
- Treat golden examples as artifacts—turn them into skills, not tribal knowledge
- Use agent-first review as policy. Humans do the final high-risk pass

If you don't standardize, every long run becomes a bespoke ops problem.

Skills, MCP, and CLI Tools

Tooling is the difference between a clever model and a useful worker.

Skills (small, reusable playbooks)

Skills are small instruction bundles that teach your agent how to do a specific thing. Typical structure: a `SKILL.md` with YAML metadata, optional `scripts/`, optional `references/`, and optional `assets/`. The key trick is **progressive disclosure**—load only skill names and descriptions by default, pull the full body only when needed.

A strong pattern: do a small slice interactively, turn that into a skill with golden examples, then run the long job using the skill.

Subagents (specialists)

Subagents are specialized agents with their own prompts and tool access. Good defaults: a “review” subagent, an “explore” subagent that only reads and maps the repo, and an “executor” subagent that runs a strict checklist loop.

MCP servers

MCP (Model Context Protocol) connects agents to external tools and data sources. High-leverage examples include **DeepWiki** for repo-grounded answers and **Postgres MCP** for validating assumptions against real data. Tradeoff: MCP can bloat the context. Use it when the task needs live data; use CLI tools when you want cheap, composable output.

Browser automation

Browser automation is a great verification loop when you don't trust the test suite yet, the change is UI-heavy, or the bug only reproduces end-to-end. Tools: Chrome DevTools MCP, Playwright MCP, Playwriter. Keep flows short and deterministic, and save screenshots to files.

CLI tools are underrated

CLI tools are perfect for long-running agents: deterministic, parsable output, composable (Unix philosophy). If a tool is well known, the agent will figure it out. If it's obscure, turn the workflow into a skill first.

Planning

Long runs are won in the planning phase. Ask the agent to ask you questions until scope is crisp: What is the target state? What does “done” mean? What should never change? What’s the rollout and rollback plan? What are the failure modes?

Example kickoff prompt

You are going to run a long background job in this repo.

First, ask me clarifying questions until you can write a plan that is crisp, testable, and safe. Do not start implementation until I answer.

When you have enough info, produce:

- 1) plan.md (architecture + constraints + verification + rollout)
- 2) checklist.yaml (summary/instructions/guidelines/tasks)
- 3) run-log.md (use the format from the playbook templates)

Rules:

- Treat the build-verify loop as the source of truth.
- Propose specific verification commands.
- Keep diffs reviewable: small commits, reversible steps.
- Use skills when a workflow will be repeated.
- Keep checklist.yaml updated. Never delete tasks.
- Record assumptions and evidence in run-log.md.

Define your verification loop

Pick the loop, then make it explicit in the plan. Common loops:

- Unit and integration tests

- API tests (contract, smoke, real API calls in a sandbox)
- Typecheck and lint
- Build artifacts
- End-to-end browser flows with screenshots
- Data validations (queries, metrics checks, diffs)

If you don't have a loop, **build the loop first**.

Golden examples (especially for refactors)

Golden examples work best when encoded into skills. Ask the agent to scan the repo and propose 10–20 candidate files representing pattern diversity. Pick 3–5, refactor them interactively, verify, and lock the pattern. Turn it into a skill and checklist section. Now the long-running agent can replicate safely.

Standard overnight refactor recipe

Pre-flight: Make sure `main` is green, create a checkout for the run, select golden examples interactively, and convert the pattern into a skill. **Kickoff:** Generate `checklist.yaml` with real acceptance criteria, open a draft PR early, and start the long run. **During:** Check in early, steer when you see drift, and don't let it fight flakes for hours. **Landing:** Run full verification, agent-first review, then human review by risk.

Templates

Copy these into your repo. These files are the memory and observability layer for long-running agents.

plan.md template

Plan: <project name>

Last updated: <YYYY-MM-DD HH:MM>

Purpose / Big picture

In 3 to 6 sentences: Why this matters, what changes, how to see it working.

Context and orientation

System overview, key files (repo-relative paths), glossary of non-obvious terms.

Scope

Goal – 1 paragraph, concrete and testable

Non-goals

Acceptance criteria (behavior)

Constraints and invariants

Must not change · Safety · Compatibility · Performance

Plan of work (milestones)

Outcome + Proof per milestone

Implementation map

Concrete edits: file, location, what to change/add.

Verification

Fast loop (iteration) · Full suite (milestones) · Evidence

Rollout and rollback

Feature flags · Migrations · Rollback plan

Idempotence and recovery

What can be rerun · What can fail halfway · How to retry

Skills and tools

Skills to use · MCP servers (if any)

Golden examples

Links to commits/files that define the pattern

```
## Open questions
Questions that block execution
```

checklist.yaml template

```
summary:
  last_updated: "<YYYY-MM-DD HH:MM>"
  active_task_id: ""
  status: "not_started"
  blockers: []
  next_actions: []
  verification_loop:
    fast: []
    full: []

instructions: |-
  Implement tasks until terminal state.
  Do not mark complete until acceptance criteria verified.
  Record evidence in task notes and run-log.md.
  Add new tasks when discovered. Never delete.

guidelines:
- "Do not change public APIs without updating callers."
- "Do not add dependencies without justification."
- "Prefer small, reversible steps."

tasks:
- task_id: T001
  title: "Establish baseline"
  description: "Run fast loop and full suite on main"
  acceptance_criteria:
    - "Fast loop passes"
    - "Full suite passes"
  status: not_started
  note: ""
```

run-log.md template

```
# Run log
```

```
## Context
```

```
Repo · Branch/worktree · Start time
```

```
## Events (append-only)
```

- <timestamp> decision: ...
- <timestamp> evidence: ...
- <timestamp> failure: ...
- <timestamp> fix: ...
- <timestamp> note: ...

```
## Follow-ups
```

```
Skills to extract · Docs to update · Tech debt spotted
```

Execution

Once `plan.md` and `checklist.yaml` look good, execution should feel **boring**.

The default execution loop



Implement tasks until they reach a terminal state (`complete` , `failed` , or `archived_as_irrelevant`). Don't mark `complete` until acceptance criteria is met and verified. Add new tasks when discovered. Never delete tasks. Keep the summary updated so a human can see current status in 30 seconds.

Compaction

Every long run eventually hits the context window. Harnesses handle this with compaction: they summarize history, prune tool output, and keep going. Done well, it feels invisible. Done poorly, the agent forgets constraints and starts making reasonable, wrong decisions.

How to make compaction a non-issue: put critical decisions in `plan.md`, put current truth in `checklist.yaml`, put surprises and evidence in `run-log.md`, and tell the agent to reread these files after compaction and at the start of each new “shift.”

If a run goes sideways, don't salvage it for hours. Fork the session and restart from a clean checkpoint.

Steering

Steering is how you correct the agent mid-run so the next cycle adapts fast. It works best when you have observability: `checklist.yaml` tells you what it thinks it's doing, `run-log.md` tells you what went wrong, and thinking blocks give you real-time peek into the thought process.

Check-in cadence

- A few minutes after the run starts
- When the agent encounters failures and subsequent recovery
- Before you go offline for the night

What to look for

- Is it running the verification loop?
- Is it making reversible steps?
- Is it stuck in a loop?
- Is it changing the right files?
- Is it drifting from the plan?

When it drifts, steer quickly

Restate the constraint it violated. Offer hints or direction. Narrow the next action to the smallest safe step. Tell it what evidence to produce before continuing.

Review

Review is where long-running work becomes shippable.

Agent-first review

Start by asking an agent to review the diff against your acceptance criteria. In Codex, use `/review` with `xhigh` effort. The review should produce a risk-ranked list of files, bugs, security issues, concurrency risks, missing tests, migration risks, and suggested follow-up tasks.

Human review by risk

After agent review, do a human pass biased toward high blast-radius areas: auth and permissions, money movement, core domain business logic, data migrations, concurrency and retries, and public APIs.

Use CI as an active loop

Open a draft PR early and let CI run repeatedly while the agent continues. The agent runs targeted subsets locally for speed, CI runs the full suite in the background, and the agent fixes failures as they appear.

Land it safely

For big refactors and migrations: ship behind a flag when you can, prefer incremental rollouts, add rollback steps to the plan, and prefer additive migrations first, destructive migrations later.

Conclusion

Long-running agents change the shape of engineering work. You spend less time on mechanical execution. You spend more time on direction, constraints, and review.

The dream output is a PR you can ship: green CI, a reviewable diff, and a `run-log.md` that explains what happened with links to evidence. Mine the run log for repeatable workflows, then turn them into skills.

You get there with boring discipline: Define “done” up front. Make the build-verify loop non-negotiable. Keep durable memory outside the chat. Make the run observable. Commit checkpoints. Pick your safety mode before you start.

Models, pricing, and limits will keep moving. Build a repeatable system from first principles: the agent makes a change, runs tools to verify, records evidence, and repeats—while you have full observability to direct the agent as needed. If that loop is solid, switching models is a routing decision.

A good run feels calm. The checklist stays accurate. The agent keeps producing evidence. Your interventions are short and early. Landing the PR feels routine.

Glossary

Harness

The runner around the model (tools, permissions, sessions, compaction, UI).

Build-verify loop

The commands and checks that prove the agent improved the system.

Compaction

Summarizing and pruning old context so the run can continue past the context window.

Golden examples

A small, human-verified slice that becomes the pattern for a large refactor.

Worktree

A separate checkout of the same git repo so each agent works in isolation.

Steering

Corrective instructions mid-run to prevent drift and wasted work.

Skills

Reusable procedures that teach the agent how to work in your environment.

MCP

A standard way to connect agents to external tools and data sources.

Ralph loop

A keep-going pattern built on stop hooks in Claude Code.

Stacked PRs

A chain of small PRs that build on each other so review stays tractable.

References

- Cursor: [Scaling long-running autonomous coding](#)
- Cursor: [Long-running agents](#)
- OpenAI: [Skills and shell tips](#)
- OpenAI: [Codex CLI getting started](#)
- OpenAI: [API pricing](#)
- Anthropic: [Pricing](#)
- Anthropic: [Claude Opus 4.6](#)
- Anthropic: [Claude Code hooks](#)
- Anthropic: [Usage limit best practices](#)
- Moonshot: [Kimi K2.5](#)
- Moonshot: [Model pricing](#)
- Z.ai: [GLM LLM guide](#)
- Z.ai: [Pricing](#)
- MiniMax: [MiniMax M2.5](#)
- DeepWiki MCP: [Docs](#)
- Agent Skills: [Standard](#)
- OpenCode: [Docs](#)
- Pi: [Repo](#)
- OpenClaw: [Repo](#)
- OpenRouter: [Router + model catalog](#)
- Benchmarks: [SWE-bench](#) · [TerminalBench](#)
- Chrome DevTools MCP: [Repo](#)
- Playwright MCP: [Repo](#)
- Playwriter: [Repo](#)

Build a repeatable system from first principles.