ANTHROPIC CLAUDE & WORKSPACE STUDIOS • FIELD GUIDE

Claude Code Autonomous Loops

Turning interactive coding chats into self-verifying engineering loops—mastering `/goal` completion contracts, Goal + Rider specs, `claude agents` fleet supervision, atomic `/rewind` recovery, and live Code Artifacts.

`/goal` Evaluator Loops | Goal + Rider Specs | `claude agents` Fleet UI | `/fork` • `/branch` • `/rewind` | `--worktree` Isolation | Live Code Artifacts



Outcome-Driven Autonomy

Replace turn-by-turn babysitting with a `/goal` contract checked after every turn by an independent evaluator model.

Parallel Fleet Control

Dispatch isolated `--worktree` runs, monitor workers with `claude agents`, and jump into Desktop via `/resume`.

Live Shareable Proofs

Publish dynamic PR walkthroughs, diff maps, and release checklists directly to private `claude.ai` Code Artifacts.

From Chat Turns to Self-Verifying Engineering Loops

Traditional AI coding is conversational: you type a prompt, watch Claude edit three files, spot a failing test, and nudge it again. Autonomous Loops flip that rhythm—you define the finish line once with `/goal`, and an independent evaluator keeps Claude working until objective proof confirms the job is done.

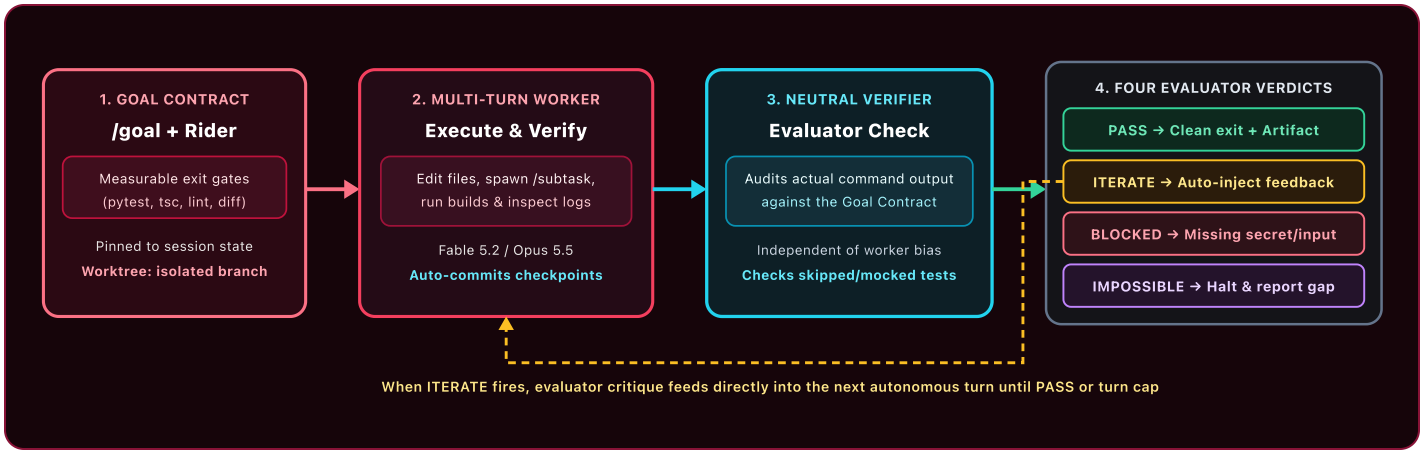
Quick Navigation

1. Loop Mental Model & Evaluator SVG p. 2
2. Prerequisites & Command Matrix p. 3
3. Annotated Fleet & Desktop UI Tour p. 4
4. Engineering `/goal` & Rider Specs p. 5
5. Color-Coded Contract Anatomy p. 6
6. Fleet Supervision & Safe Navigation p. 7
7. Live Code Artifacts & Sharing p. 8
8. 5 Copy-Paste Loop & Fleet Recipes p. 9
9. Guardrails for Long-Horizon Runs p. 10
10. Cross-Tool Synergy Pipeline p. 11
11. Troubleshooting & Pre-Flight p. 12
Golden Rule: Proof Before Completion All

The Three Pillars of Autonomous Claude Code Workflows

Reliable autonomy rests on separating **doing the work** from **judging whether the work is complete**. When you trigger `/goal`, the primary coding model (such as `Opus 5.5` or `Fable 5.2`) edits code and runs tests across multiple turns. After each turn settles, a fast, neutral evaluator inspects the transcript, tool outputs, and workspace state—returning one of four verdicts instead of blindly trusting the worker's optimism. Meanwhile, `claude agents` lets you supervise multiple loops concurrently, and **Code Artifacts** package the finished diff into a live visual walkthrough.

Autonomous Execution & Independent Evaluator Cycle



Why Separate Worker & Evaluator Models Prevent Premature "Done!" Claims

Coding models naturally want to please you after fixing the obvious bug. Because the `/goal` evaluator only sees your exit criteria and raw command evidence, it catches unverified assumptions, skipped suites, or lingering type errors and automatically triggers the next turn.

Prerequisites, Model Tiers & Command Matrix

Autonomous loops work out of the box across both the **Claude Code CLI** and **Claude Code Desktop**. Pairing isolated Git worktrees with right-sized model tiers and subagent definitions keeps multi-hour runs fast, affordable, and cleanly reversible.

1 CLI & Desktop Sync

Install or update with `claude update`. Every CLI session shares history with Claude Code Desktop so you can launch overnight loops in terminal and review diffs visually via `/resume`.

2 Worktrees & Agents

Launch isolated branches via `claude --worktree <name>`. Store reusable specialist personas (reviewer, test-runner, security-auditor) as Markdown files inside `.claude/agents/`.

3 1M Context & Effort

Use `Opus 5.5` (1M context window) for high-speed daily engineering loops, or step up to `Fable 5.2` on High/Max effort for deep architectural refactors and multi-hour research goals.

Essential Workspace Paths & CLI Launch Flags

```
# Launch an isolated autonomous worktree session with Opus 5.5 (1M context) and budget cap
claude --worktree feat-billing-v2 --model claude-opus-5-5 --max-budget-usd 12

# Open the interactive fleet monitor across all active background loops in this repo
claude agents --cwd .           # Add --json --all to script status dashboards in CI

# Define project-wide specialist subagents in Markdown (committed to Git for your team)
.claude/agents/test-verifier.md   # Read-only verifier persona with strict Bash test-only rules
.claude/goals/billing-rider.md    # Deep phased spec referenced by your concise /goal contract
```

Session Control & Multi-Agent Comparison Matrix

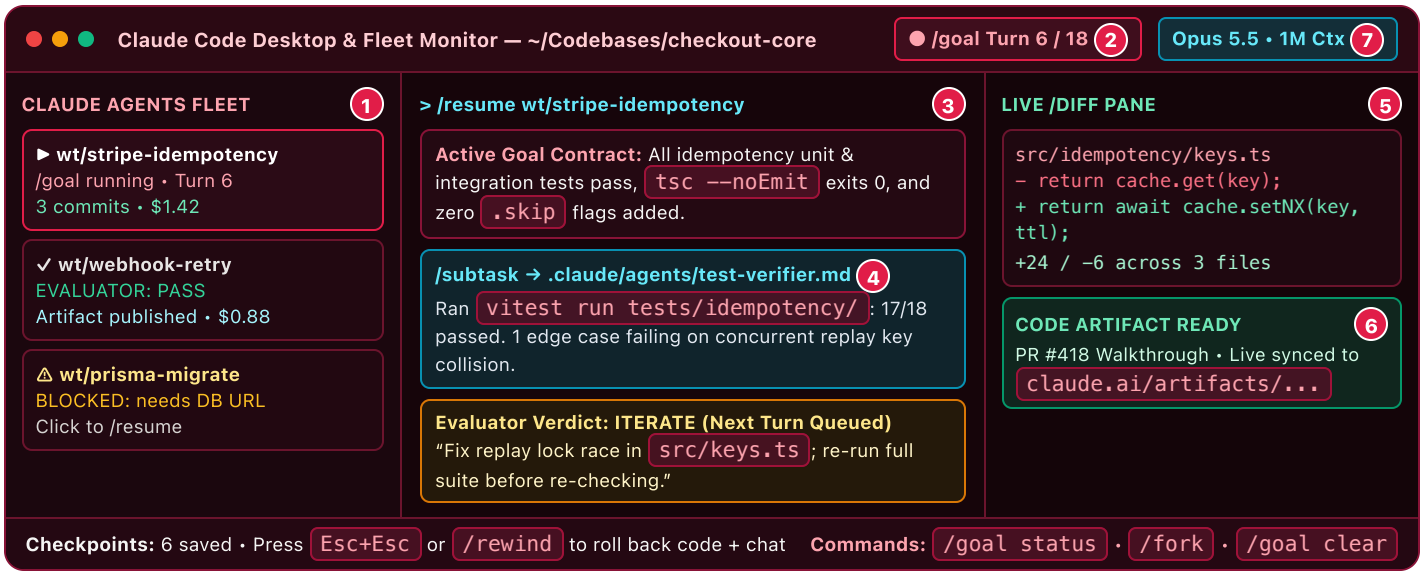
Command	What It Creates	Workspace & Context Effect	Best Used When...
`/goal`	Persistent evaluator contract attached to current session	Loops automatically across turns until exit criteria pass or `/goal clear`.	You want hands-free execution verified by tests, type-checks, or build output before stopping.
`/subtask`	Scoped in-session child worker	Runs isolated work and returns a concise summary without bloating parent context.	Delegating file exploration, log triage, or parallel test shards inside one active goal loop.
`/fork`	Background copy of current session	Clones full context into an independent background session while you stay in place.	Spinning off a side experiment or documentation pass while keeping your main loop uninterrupted.
`/branch`	Named parallel conversation branch	Clones context and switches your active terminal directly into the new branch.	Comparing two rival architectures from a clean baseline without losing your original thread.
`/rewind`	Atomic time-travel rollback	Restores *both* conversation history and workspace file edits to an earlier turn.	An autonomous loop chased a dead end, broke tests, or filled context with noisy stack traces.
`/resume`	Interactive or named session switcher	Reopens any CLI or background session inside terminal or Claude Code Desktop.	Picking up a finished overnight `/goal` run in Desktop's side-by-side `/diff` viewer.

Choosing Between `Opus 5.5` and `Fable 5.2` for Autonomous Loops

Start 80% of `/goal` runs on **Opus 5.5** (1M context): its fast token throughput lowers loop latency and cost while handling multi-file migrations effortlessly. Reserve **Fable 5.2** with High effort for tangled race conditions, cross-service schema redesigns, or strict zero-regression audits.

Annotated Tour: Fleet Dashboard & Desktop Review

Power users rarely babysit a single terminal prompt. Instead, they supervise parallel worktrees in `claude agents` and jump straight into Claude Code Desktop via `/resume` to inspect live diffs and loop telemetry.



Interface Legend: The 7 Autonomous Cockpit Touchpoints

1. `claude agents` **Fleet Sidebar:** Lists every active, passed, or blocked background loop with branch name, turn progress, and live spend.
2. `/goal` **Turn & Evaluator Badge:** Tracks current autonomous iteration against your turn cap and displays the last evaluator verdict.
3. **Cross-Surface** `/resume` **Handoff:** Opens any headless CLI worktree session inside Claude Code Desktop without restarting or losing state.
4. `/subtask` **Specialist Delegation:** Dispatches scoped helpers from `.claude/agents/` so verbose test logs stay out of the main loop context.
5. **Live Desktop** `/diff` **Inspector:** Streams line-by-line edits inside the isolated worktree so humans can audit changes before merging.
6. **Live Code Artifact Card:** Publishes interactive PR walkthroughs, architecture maps, and release checklists to private `claude.ai` pages.
7. **Model, 1M Context & Atomic** `/rewind` **Bar:** Confirms active tier (`Opus 5.5` / `Fable 5.2`), remaining context headroom, and checkpoint count for instant one-command rollback of both files and chat history.

Engineering `/goal` Contracts & Goal + Rider Specs

A loop is only as sharp as its exit condition. Vague prompts like *"improve performance"* confuse the evaluator, whereas concrete contracts backed by a **Goal + Rider** Markdown pair power hours of disciplined, self-correcting work.

1 How the Evaluator Judges Each Turn

- **Satisfied (`PASS`):** Every named test command, type check, and deliverable file shows explicit passing evidence in tool output.
- **Needs Work (`ITERATE`):** Output shows failing tests, partial diffs, or unverified claims; evaluator sends targeted feedback into the next turn.
- **Blocked / Impossible:** Missing API credentials, broken external dependency, or contradictory constraints halt the loop cleanly so you can intervene.

2 The Goal + Rider Pattern

- **Goal File (~4,000-char spine):** Ultra-crisp contract passed to `/goal`—holding the target outcome, read-first paths, exit commands, and stop rules.
- **Rider File (unlimited depth):** Detailed spec committed in `.claude/goals/<slug>-rider.md` with phased steps, SQL/API schemas, edge cases, and non-goals.
- **Phased Commits:** Instruct the loop to commit after each passing phase so `/rewind` or `git reset` never loses progress.

```
# Invoke /goal referencing a concise contract + deep committed Rider spec
/goal Read .claude/goals/webhook-rider.md first. Implement Phases 1–3 on this worktree, commit after each green phase, and
stop only when pnpm test:webhooks, pnpm typecheck, and pnpm lint all exit 0 with zero skipped tests.
```



Continuous Execution Meets Deterministic Verification: Every coding turn spins through the crimson execution ring, passes across the neutral sapphire evaluator gate to verify real command outputs, and advances to the emerald completion beacon only when zero exit conditions remain unmet.

Anatomy of a Bulletproof `/goal` & Rider Contract

Whether you write a one-line `/goal` command or a multi-phase Rider file, packing these five color-coded clauses eliminates drift, endless loops, and silent corner-cutting.

[1. Target Outcome] [2. Verifiable Exit Gate] [3. Boundary / Non-Negotiables] [4. Checkpoint / Commit Cadence] [5. Escalation / Impossible Stop Rule]

`/goal` Migrate all legacy REST billing endpoints in src/routes/v1/billing/ to idempotent Zod-validated handlers following .claude/goals/billing-rider.md. Exit gate: npm run typecheck, npm run lint, and npm test -- billing all run fresh after the final edit and exit 0 with 100% pass rate. Non-negotiables: do not skip, delete, or weaken existing assertions; do not touch src/routes/v2/ or production env files. Create a clean git commit after each endpoint passes its integration test suite, and summarize changed files at the end. If Stripe sandbox credentials are missing or a schema migration requires manual DBA approval, mark the goal blocked/impossible immediately and list the exact gap.

Vague Goals vs. Evaluator-Ready Contracts

Weak / Flaky `/goal`	Why the Evaluator Struggles	Hardened Replacement
“Refactor the auth module to be cleaner.”	No objective pass signal; evaluator either passes too early or loops forever nitpicking style.	“Split `auth.ts` into `session.ts` and `jwt.ts` per `auth-rider.md`; all 28 unit tests and `tsc --noEmit` pass.”
“Fix the flaky CI test.”	Passing once by luck fools the loop unless stress repetition is required.	“Eliminate race condition in `checkout.spec.ts`; run `npx playwright test --repeat-each=10` with 0 failures.”
“Make the dashboard load faster.”	“Faster” cannot be checked without a baseline metric and benchmark script.	“Reduce `dist/app.js` bundle under 180 KB gzipped and keep `npm run build && npm test` green.”

Rider Sizing Rule: Keep the `/goal` Spine Under ~4,000 Characters

Because the evaluator re-reads your `/goal` contract on every turn, keep the active goal string compact and link to your detailed Rider Markdown file on disk. That saves evaluator tokens and ensures critical non-negotiables never get truncated.

Parallel Fleet Supervision & Safe Session Navigation

Long autonomous runs shine when you parallelize cleanly and recover fearlessly. Combining specialist `.claude/agents/`, `/subtask`, `claude agents`, and atomic `/rewind` lets one engineer direct an entire squad of isolated workers.

A Fleet Orchestration: `claude agents` & `/subtask`

- **Project Personas (`.claude/agents/`):** Define scoped subagents (e.g., `reviewer.md`, `migrator.md`) with explicit model tiers and allowed tools.
- `/subtask` **Delegation:** Offload verbose log searches or test shards so raw 20k-token stack traces never pollute your primary `/goal` context.
- `claude agents` **Dashboard:** Inspect every background worker across worktrees from one terminal UI, or run `claude agents --json --all` for CI polling.

B Safe Navigation: `/resume`, `/fork`, `/branch`, `/rewind`

- **CLI-to-Desktop `/resume`:** Kick off three loops from your terminal, then call `/resume` inside Claude Code Desktop to inspect diffs and file trees visually.
- `/fork` **vs.** `/branch`**:** Use `/fork` to spawn a background side-experiment while staying put; use `/branch <name>` when you want to switch into the new fork immediately.
- **Atomic `/rewind` (`Esc + Esc`):** Rolls back *both* conversation turns and Claude's file edits in lockstep to any prior checkpoint.

Anatomy of a Reusable Verifier Persona in `.claude/agents/test-verifier.md`

```
---
name: test-verifier
description: Neutral test and type-check runner invoked by /goal loops via /subtask
model: claude-opus-5-5
tools: Read, Grep, Glob, Bash
---
Run only read-only checks and test commands. Never modify source files or add .skip/.only flags.
Return exact exit codes, failed test names, and minimal stack lines back to the parent session.
```

Pro Recovery Playbook: When a `/goal` Loop Drifts on Turn 8

Never argue with an autonomous loop that has filled 70% of its context window with broken attempts. Hit `/goal clear`, run `/rewind` back to Turn 3 (where the baseline was clean), tighten one constraint in your Rider file, and relaunch `/goal` with fresh context.

Claude Code Artifacts: Live Shareable Engineering Output

When an autonomous loop finishes 400 lines of changes across twelve files, dumping a wall of Markdown into Slack wastes your team's time. **Claude Code Artifacts** publish rich, interactive engineering pages directly from your session to private, live-updating `claude.ai` URLs.

1 What You Can Publish From Terminal

- **PR Walkthroughs with Inline Diffs:** Group changes by architectural layer, explain *why* each module changed, and attach live test pass tables.
- **Debug & Incident Timelines:** Pair stack traces, log excerpts, and root-cause diagrams from an autonomous triage loop.
- **Architecture & Flow Diagrams:** Interactive HTML/SVG maps showing data flow across services updated by your branch.

2 Live Sync & MCP-Backed Checklists

- **Live-Updating Pages:** Keep iterating in your CLI worktree—as tests go green or commits land, ask Claude to refresh the published artifact in place at the same private URL.
- **MCP Release Checklists:** Combine local git diffs with GitHub, Linear, or Sentry MCP tools inside one interactive pre-merge checklist for reviewers.
- **Private Workspace Sharing:** Share securely with teammates on `claude.ai` without pasting secrets into public pastebins.



From Raw CLI Loop to Interactive Review Page: Claude Code Artifacts project terminal diffs, verifier logs, and MCP release checks onto frosted, shareable web panes on `claude.ai`—giving code reviewers instant clarity before opening a pull request.

5 Real-World Copy-Paste Loop & Fleet Recipes

Drop any recipe below into a fresh `claude --worktree <branch>` session. Each prompt pairs a verifiable exit gate with clear boundaries so the evaluator knows precisely when to advance or stop.

Workflow Recipe	Features Combined	Copy-Paste Prompt / `/goal` Command

1. Overnight Test-Fix Sprint
Zero-Flake CI
	`/goal`
	`--worktree`
	`test-verifier`
		`/goal Fix every failing unit test in tests/payments/. Delegate full suite runs via /subtask to test-verifier, commit after each module turns green, never mark tests .skip, and stop only when npm test -- payments and npm run typecheck exit 0.`

2. Phased Rider Migration Multi-Hour Spec	`/goal` + Rider `Fable 5.2` Phased commits	`/goal Execute .claude/goals/orm-v4-rider.md Phases 1-4 in order. Create one commit per phase with passing migration dry-run logs. Exit only when schema diff is empty, pnpm test:integration passes, and zero legacy v3 imports remain.`

3. Parallel 3-Branch Spike
Fleet Exploration
	`/branch`
	`/fork`
	`claude agents`
		`Baseline tests are green. Create two branches via /branch cache-redis and /fork cache-sqlite, benchmark both implementations with 1,000 concurrent reads, monitor both in claude agents, and summarize p95 latency + diff size side-by-side.`

4. Live PR Walkthrough Artifact Reviewer Handoff	Code Artifact `/diff` GitHub MCP	`Inspect git diff main...HEAD and our latest test logs, then publish a private Claude Code Artifact titled "PR #418 Auth Refactor Walkthrough" containing an architecture SVG, per-file inline diff notes, and an interactive pre-merge checklist.`
5. Live Incident Triage & Hotfix Production Rescue	`/goal` Sentry MCP `/rewind` safety	`/goal Pull Sentry issue #8812, write a failing regression test reproducing the null payload crash first, implement the minimal fix, and verify full suite pass. Publish an incident timeline Code Artifact once all tests exit 0.`

Chaining Recipe #2 + Recipe #4: The Complete Unattended-to-PR Workflow

Run your phased migration inside a `--worktree` overnight using Recipe #2, wake up and verify progress in `claude agents`, then trigger Recipe #4 to generate a polished Code Artifact walkthrough your teammates can review in two minutes.

Guardrails for Long-Running Autonomous Loops

Giving an agent thirty turns of autonomy demands engineering discipline. Without hard spend limits, isolated worktrees, and anti-cheating rules, an eager model can burn budget, mutate shared branches, or quiet a failing test by relaxing assertions.

Four Hard Boundaries Every Loop Needs

- **Always Isolate Disk State:** Launch unattended runs with `claude --worktree` so uncommitted edits never collide with your main checkout.
- **Cap Spend & Turns:** Pass `--max-budget-usd` (or turn limits) on every headless or overnight run so a stubborn bug cannot spin endlessly.
- **Ban Test Dilution:** State explicitly in both `CLAUDE.md` and your Rider: zero `.skip`, zero lowered coverage thresholds, zero mocked-away failures.
- **Sandbox Secrets & Prod:** Deny production database URLs and force-pushes via Claude Code permissions or `sec-default` mods.

Catching Subtle Loop Failure Modes

- **Silent Mock Relaxation:** If a worker edits a test expectation just to make Jest pass, your evaluator prompt and PR diff check must reject the turn.
- **Context Poisoning:** After 10+ failed compile logs, context quality degrades. Use `/subtask` for noisy runs and `/rewind` if the main thread gets stuck.
- **Unverified “Should Pass” Claims:** Require fresh post-edit command execution—never allow the loop to cite a test run from three edits ago.
- **Human Merge Gate:** Autonomous loops open drafts and Code Artifacts; humans approve merges to `main`.

Drop-In Safety Clauses for Your Repo’s `CLAUDE.md`

```
## Autonomous /goal Loop Policy
1. Never add .skip, .only, @pytest.mark.skip, or relaxed assertions to make a /goal check pass.
2. Always run the full target test and typecheck commands AFTER your final file edit before finishing.
3. Commit working milestones per phase; never push directly to main or run destructive database resets.
4. If external secrets, staging access, or ambiguous product decisions block progress, halt and report BLOCKED.
```

Pair Loops With Deterministic Hooks or Mods for 100% Enforcement

Prompt rules inside `CLAUDE.md` cover 95% of cases, while deterministic pre-tool hooks (blocking `git push --force` or edits to `.env.production`) make critical boundaries unbreakable even on turn twenty.

Cross-Tool Synergy: From Spec to Live PR Artifact

Autonomous loops shine brightest when every stage plays its natural role: Markdown specs anchor intent, isolated worktrees and specialist subagents execute safely, Desktop `/resume` speeds human review, and Code Artifacts communicate the result.



Example Day-in-the-Life: Shipping a Feature Across Three Parallel Loops

1 Morning Launch & Midday Audit

You commit `search-rider.md` and launch two `claude --worktree` sessions on `Opus 5.5` with `/goal` contracts. At lunch, `claude agents` shows one green and one iterating; you open the green session via `/resume` in Claude Code Desktop to review its `/diff` pane.

2 One-Click Artifact Handoff

When the second loop turns green, you ask Claude to publish a live **Code Artifact** summarizing API route changes, benchmark numbers, and GitHub MCP release checks—dropping one clean private link into your PR description.

Always Treat `claude agents` + `/rewind` as Your Safety Net

Running parallel loops only feels effortless when reversing a misbehaving branch takes two seconds. Because every loop lives in its own worktree and saves turn checkpoints, bad runs cost pennies to rewind and zero time to clean up.

Troubleshooting Clinic & Pre-Flight Checklist

Almost every autonomous loop hiccup stems from ambiguous exit criteria, noisy context buildup, or missing environment isolation. Keep this ten-second triage table beside your terminal before kicking off unattended runs.

Symptom / Failure	Root Cause	10-Second Fix
1. Evaluator stops early and declares PASS on broken code	`/goal` lacked an explicit command check (e.g., only said "finish refactor").	Include exact shell exit gates (`npm test && tsc --noEmit`) and require fresh output after the final edit.
2. Loop repeats the exact same failing edit across turns	Context window is saturated with stale error logs or conflicting hypotheses.	Run `/goal clear`, hit `/rewind` to the last green checkpoint, add the missing clue to your Rider, and restart.
3. Main session hits 80% context after two test runs	Full build/test stdout was printed straight into the parent conversation.	Delegate test execution through `/subtask` using a read-only `.claude/agents/` verifier that returns only failures.
4. Loop immediately halts with `BLOCKED` / `IMPOSSIBLE`	Missing `.env` secrets, offline dependency, or contradictory constraint in Rider.	Inspect the evaluator's gap report, supply the sandbox env var or relax the conflicting rule, and re-arm `/goal`.
5. Two parallel loops overwrite each other's files	Both sessions launched in the same working tree instead of isolated Git worktrees.	Always start parallel sessions with `claude --worktree <task-name>` and track them in `claude agents`.
6. Worker silenced a failing test to pass the gate	No anti-dilution guardrail in `CLAUDE.md` or `/goal` non-negotiables.	Add "zero `.skip`/weakened assertions" to `CLAUDE.md`, review `/diff` in Desktop, and `/rewind` the cheat turn.
7. Published Code Artifact shows outdated diff or logs	Additional commits landed after the artifact was first generated.	Run a quick follow-up prompt asking Claude to sync the existing Code Artifact with current `HEAD` and test results.

6-Point Pre-Flight Checklist Before Launching an Autonomous `/goal` Loop

- ☑ **Isolated Worktree Active:** Launched with `claude --worktree <slug>` on a clean base commit.
- ☑ **Budget & Model Sized:** Selected `Opus 5.5` (or `Fable 5.2` High effort) with `--max-budget-usd` set.
- ☑ **Compact Goal + Deep Rider:** Kept `/goal` under ~4,000 chars and moved phased detail into `.claude/goals/`.
- ☑ **Objective Exit Gate Set:** Named exact test, lint, and type-check commands that must pass after final edits.
- ☑ **Anti-Cheating Rules Pinned:** Disallowed `.skip`, deleted assertions, or unapproved dependency changes.
- ☑ **Review & Artifact Plan Ready:** Configured `claude agents` monitoring, Desktop `/resume` diff check, and final Code Artifact export.