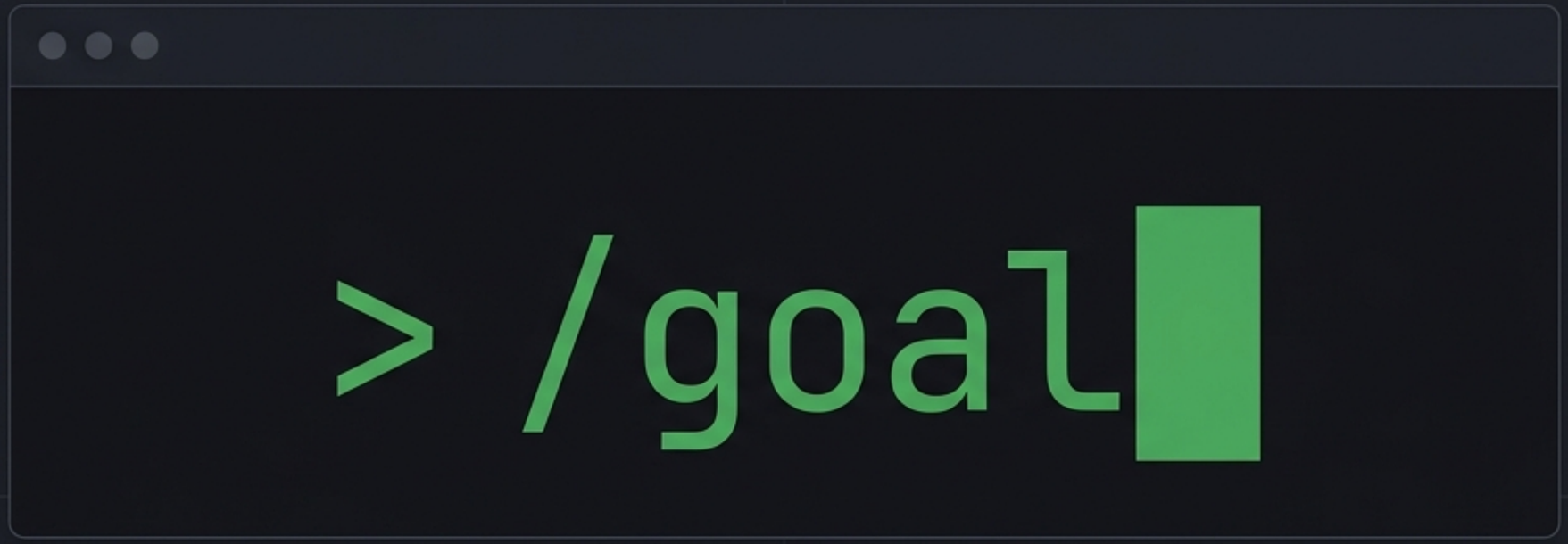
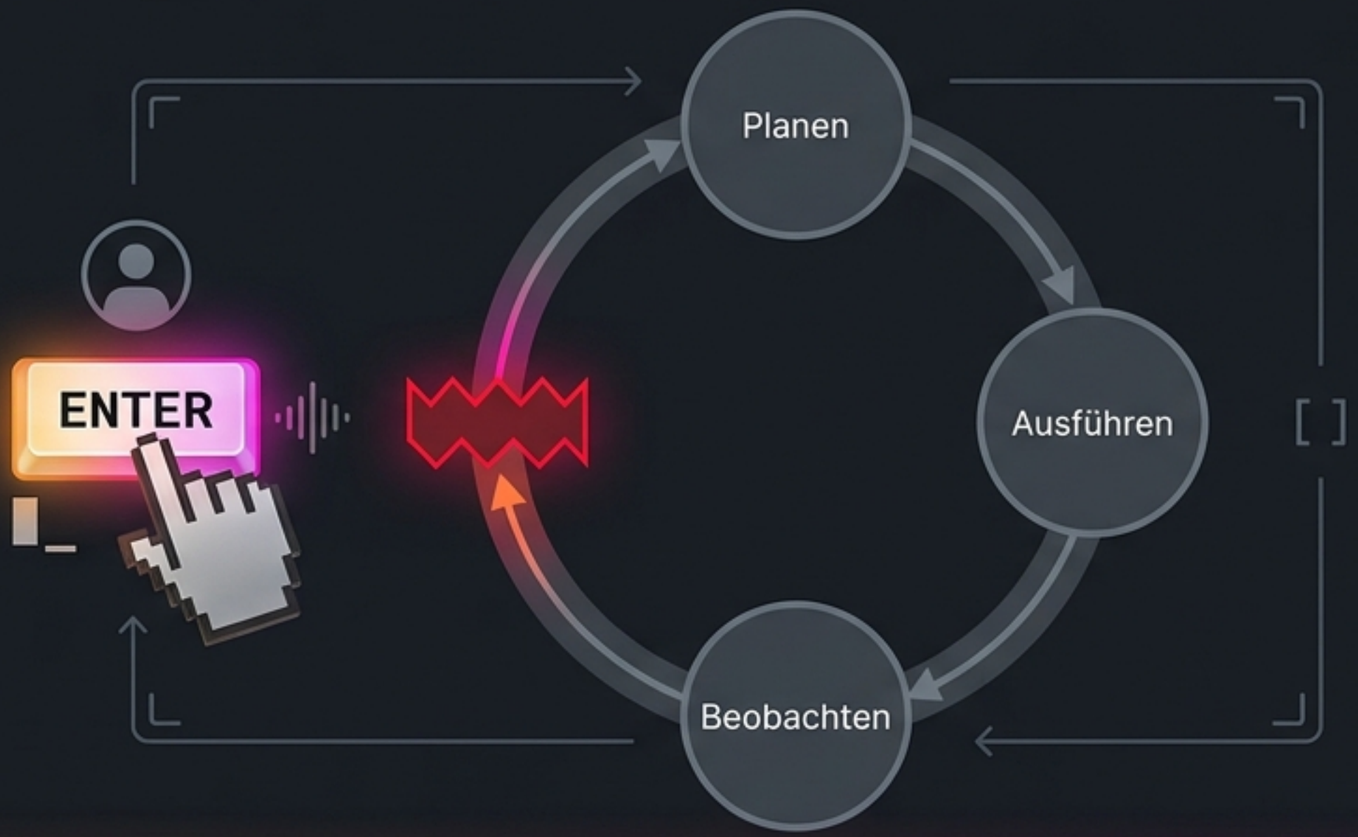


/goal: Der Befehl, der Code-Assistenten zu Agenten macht

Wie OpenAI und Anthropic in 11 Tagen den **De-facto-Standard** für **autonomes Coding** definierten.

Das Problem vor /goal (Die Ralph Loop)



>_ Problem 1

Modelle sind darauf trainiert, aus Höflichkeit zu stoppen und auf den nächsten Prompt zu warten.

>_ Problem 2

Entwickler erzwangen Autonomie künstlich durch Community-Hacks (Bash-Skripte, Cron-Jobs, manuelle Stop-Hooks).

Das Grundproblem war das Fehlen einer nativen, sauberen Instrumentierung für echte Autonomie.

Die 11-Tage-Revolution (Mai 2026)

30. April

OpenAI veröffentlicht Codex CLI 0.128.0. Der offizielle Startschuss für persistierte Goal-Workflows.

[/goal >_]

Zwischenphase

Open-Source-by-Pressure. Ein Community-Entwickler (jthack) baut innerhalb einer Woche einen funktionsfähigen Klon.

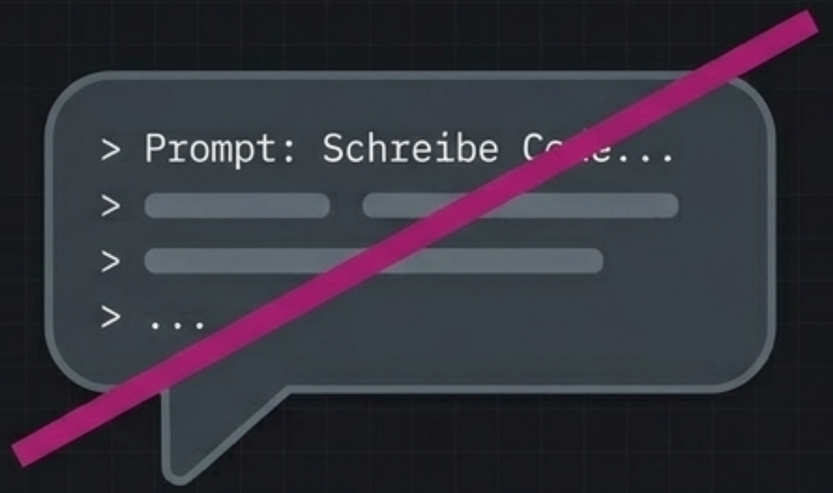
[<jthack/>]

11. Mai

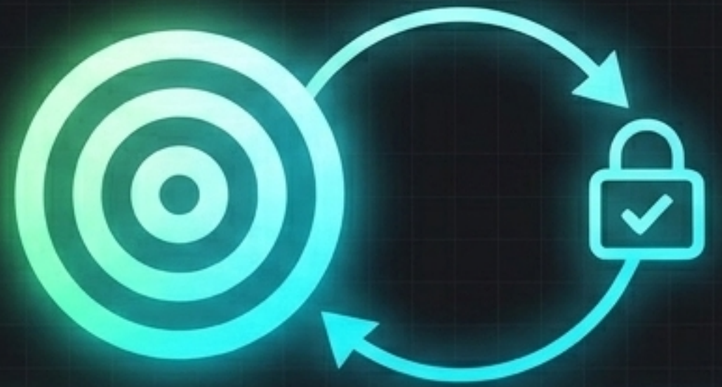
Anthropic zieht nach und veröffentlicht Claude Code 2.1.139 mit nativer /goal Unterstützung.

[/goal >_]

Die kürzeste Definition



Kein einzelner Prompt



> **Ein Ziel mit überprüfbarer Endbedingung**

> _ Der Agent arbeitet eigenständig Runde für Runde.

[ ] Ein Prüfmechanismus entscheidet nach jedem Turn: Ziel erreicht?

[ ] Nein -> Nächste autonome Runde. / Ja -> Terminal zurück an den Nutzer.

Unter der Haube – Die Codex-Architektur

Schicht 5: TUI UX
Fortschrittsanzeige und injizierte Prompts nach jedem Turn.

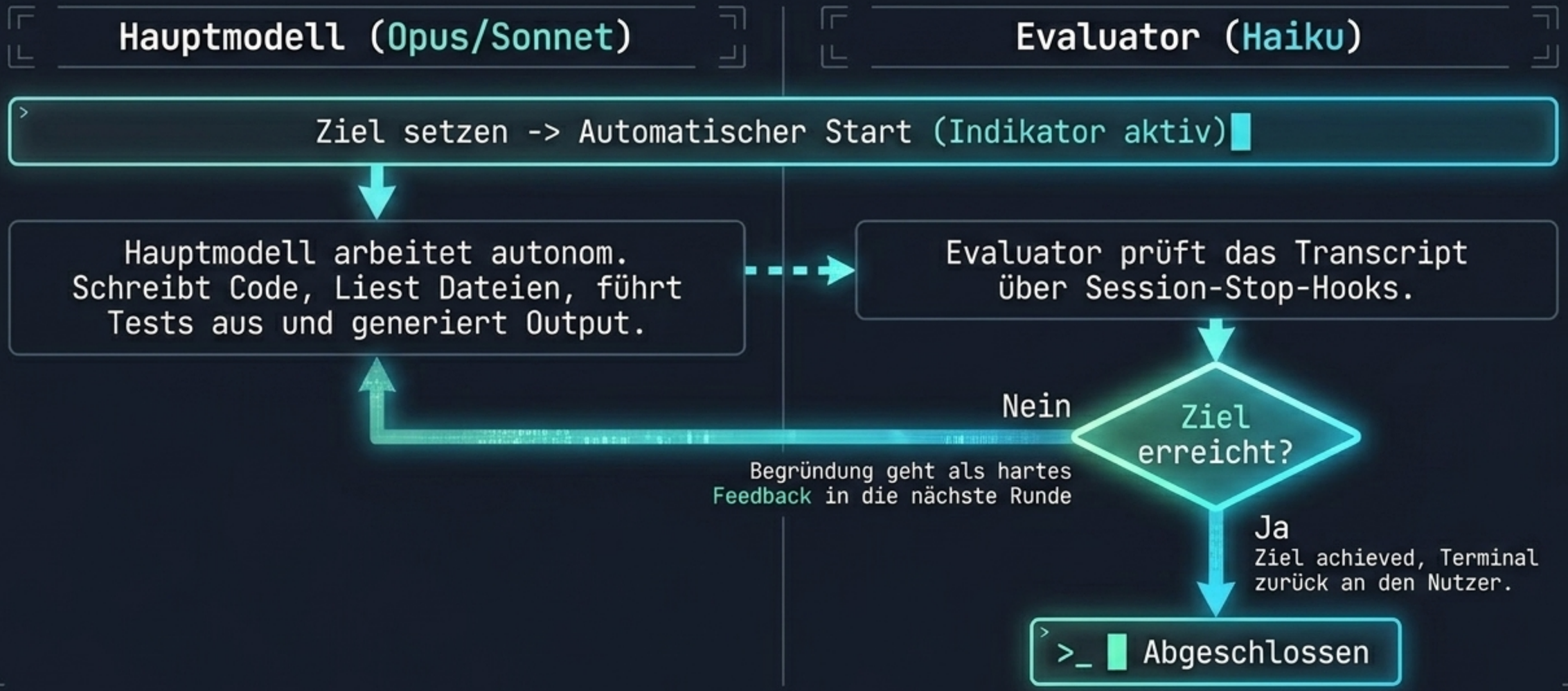
Schicht 4: Core Runtime
Event-Bus und Token-Abrechnung. Continuation Suppression killt Endlos-Höflichkeitsschleifen.

Schicht 3: Model-Tools
Dem Modell stehen exklusiv create_goal, update_goal UND get_goal zur Verfügung. Pausieren bleibt nutzerkontrolliert.

Schicht 2: API
JSON-RPC (set/get/clear) für die Anbindung externer IDEs.

Schicht 1: Persistenz
SQLite (thread_goals) – Status überlebt Reboots. Strikte State-Machine (active, paused, complete).

Unter der Haube – Der Claude-Lifecycle



Codex vs. Claude Code
(Der System-Vergleich)

	Codex	Claude Code
Zustand/Persistenz	SQLite-basiert, überlebt System-Neustart.	Session-scoped (über Stop-Hooks).
Ziel-Validierung	Inline Runtime Checks (Hauptmodell bewertet sich selbst).	Separates Evaluator-Modell (Haiku) als neutraler Richter.
Budget-Management	Natives Token-Budget-Limit in der Datenbank.	Kein natives Cap, Risiko von Endlosschleifen.

Der Sweet Spot (Wann nutzen?)

Perfekt für:

OLD_FRAMEWORK → NEW_FRAMEWORK

Framework-Migrationen (Klares Vorher/Nachher).

Prototyping aus einer Spezifikation (PLAN.md mit Milestones).

[✓] Milestone 1 > [✓] Milestone 2 > [_] Milestone 3

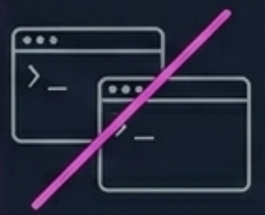
Iterative Optimierung (Gegen externe Metriken, z.B. Tests reparieren bis alles grün ist).

[PASS] 98/98 tests

Schlecht für:

Explorative Architekturfindung (Dialog ist hier wertvoller).

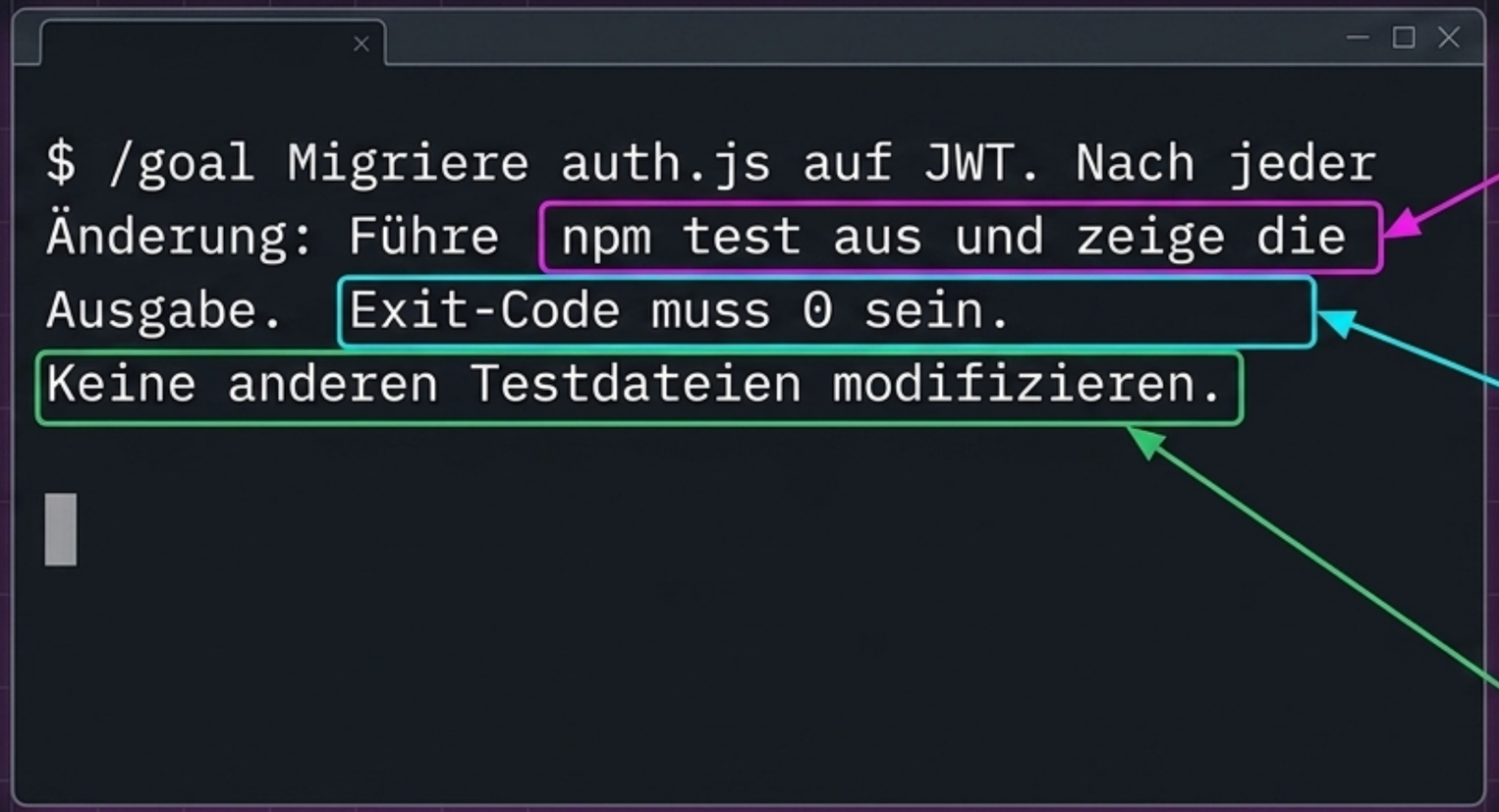
Schwammige Stopp-Bedingungen (Mach den Code besser).



Parallelisierte Workflows (Aktuell nur 1 Ziel pro Session/Worktree möglich).

[ERROR] Session lock conflict.

Die Anatomie des perfekten Ziels



Binär & messbar: Statt 'mach es besser' -> 'Exit-Code 0'.

Verifikation explizit machen: Der Evaluator braucht den Beweis im Transcript!

Constraints setzen: Verhindert kreative Uminterpretation durch den Agenten.

Playbook & Risiko-Management

Die Gefahr: Der 200-Dollar-Vorfall – Schwammige Ziele verursachten **14-Stunden-Loops**, in denen sich der Agent in einer Nacht verhedderte.

Mitigation 1 (Turn-Limits erzwingen)

Hard-Caps direkt in die Bedingung schreiben (z.B. 'Stop after 30 turns or if tests pass').

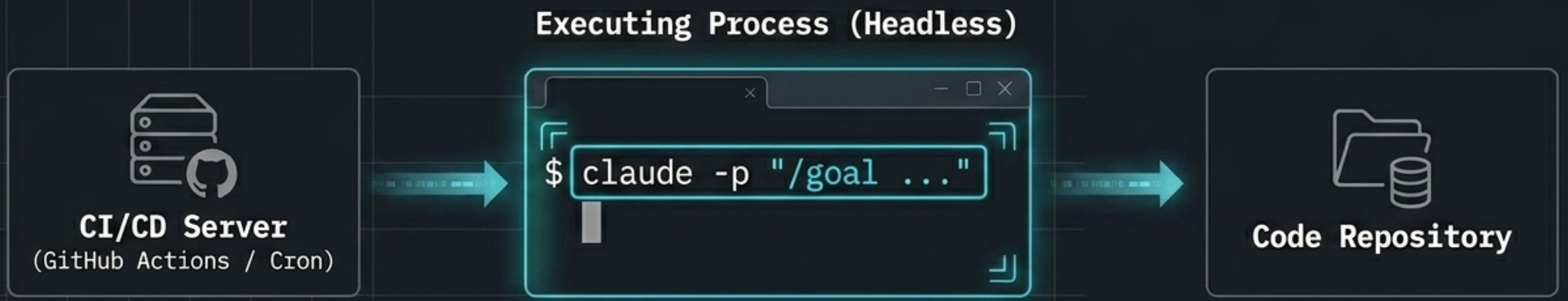
Mitigation 2 (Evaluator Kosten senken)

Das kleine Evaluator-Modell ist konfigurierbar. Der Wechsel auf ein noch günstigeres Modell senkt die Prüfkosten pro Turn erheblich.

Mitigation 3 (Sandboxing)

Zuerst kurzer interaktiver Probelauf für 2-3 Turns. Grundsätzlich immer in einem dedizierten Git-Feature-Branch arbeiten.

CI/CD & Headless Integration



</> /goal ist nicht auf die interaktive Terminal-UI beschränkt.

>_ Ideal für nächtliche Cron-Jobs und automatisierte Refactorings in CI/CD-Pipelines.

>_ Sicherheit: Der Headless-Aufruf reagiert auf Standard-Signale (Ctrl+C) und kann von der Pipeline sauber beendet werden.

Essenzielle Skills für Autonomie

Spec-Writing

Weg vom klassischen Prompting. Näher an Test-Driven Development (TDD). Ziele müssen für Maschinen binär greifbar sein.

Verifkationsdenken

Das Mindset eines QA-Engineers übernehmen. Die

Kernfrage lautet stets: Wie kann die Maschine beweisen, dass sie fertig ist?

Cost-Awareness $

Ein intuitives Gespür für Turn-Zyklen und API-Kosten-Metriken entwickeln.

Wissen, wann ein Loop zu teuer wird.

Fazit & Ausblick

Die Konvergenz

Das Zeitfenster für proprietäre CLI-Unterschiede schließt sich – das Pattern der persistierten Ziele mit Validatoren ist nun der De-facto-Standard.

Die Zukunft

Weg vom Chat. Hin zu verschachtelten Goals (Goal-Trees) und deklarativen Pipelines (YAML-basierte Agenten-Steuerung).

Dein nächster Schritt

Klein anfangen, Bedingungen schmerzhaft präzise formulieren, und den Agenten die Nacht durcharbeiten lassen.