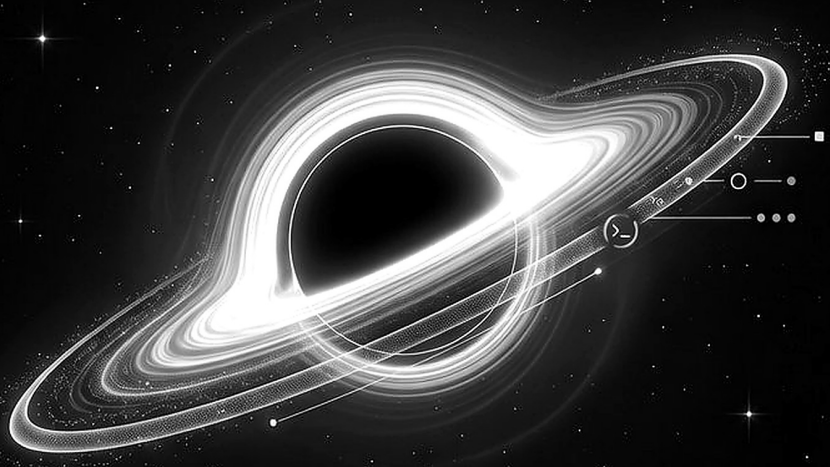
PRAXISBUCH

Grok Build

Der umfassende Praxisleitfaden

Der Coding-Agent von xAI richtig einsetzen

Plan Mode · AGENTS.md · Headless



Martin Richardt

Impressum

Grok Build – Der umfassende Praxisleitfaden
Der Coding-Agent von xAI richtig einsetzen

© 2026 Martin Richardt
Alle Rechte vorbehalten.

Stand: Juli 2026 (Grok Build / xAI / SpaceXAI).
Autor: Martin Richardt

Dieses Werk ist urheberrechtlich geschützt. Kurze Zitate und interne Weitergabe im Team sind gestattet; kommerzielle Vervielfältigung nur mit Zustimmung des Autors.

Produktnamen (xAI, Grok, Grok Build, SpaceXAI, …) sind Eigentum der jeweiligen Rechteinhaber. Dies ist keine offizielle xAI-Dokumentation.

Inhaltsverzeichnis

Vorwort 1
Warum dieses Buch – und wie du es liest 1
Kapitel 1 8
Einführung und Philosophie – was Grok Build wirklich ist 8
Kapitel 2 19
Installation und erster Start 19
Kapitel 3 31
Die interaktive TUI im Detail 31
Kapitel 4 44
Plan Mode – Review und Freigabe 44
Kapitel 5 54
Subagents und Parallelität 54
Kapitel 6 65
Worktrees und isolierte Arbeit 65
Kapitel 7 75
AGENTS.md und Projektregeln 75
Kapitel 8 87
Skills, Plugins, Hooks und Marketplaces 87
Kapitel 9 98
MCP-Server und externe Tools 98
Kapitel 10 108
Headless Mode, Scripting und CI 108
Kapitel 11 118
ACP, IDE-Integration und Bots 118
Kapitel 12 125
Memory, Sessions und Kontextmanagement 125
Kapitel 13 134
Permissions, Sandbox und Sicherheit 134
Kapitel 14 144
Konfiguration und Custom Models 144
Kapitel 15 153
Reale Workflows und Use-Cases 153
Kapitel 16 163
Best Practices, Tipps und Troubleshooting 163

Kapitel 17 171
Vergleich mit Claude Code, Codex und Cursor 171
Kapitel 18 178
Anhang – Schnellreferenz 178

Vorwort

Warum dieses Buch – und wie du es liest

> ⊙ **KERN**
>
> Grok Build ist kein Chatfenster mit Terminal-Ambiente, sondern ein **agentischer Harness** um ein Sprachmodell: Es liest Code, plant, ruft Werkzeuge auf, wartet auf deine Freigaben und schreibt Änderungen – und genau diese Schleife musst du beherrschen, um produktiv und sicher zu arbeiten.

Für wen dieses Buch ist

Du arbeitest mit Code im Terminal (oder lernst das parallel mit dem Buch-Übungsprojekt). Coding-Agenten können neu für dich sein – das ist der Normalfall für den Einstiegspfad, nicht die Ausnahme. Zielgruppe:

- **Professionelle Entwickler und Juniors**, die Features, Fixes und Refactors im Repo erledigen – nicht nur „eine Funktion erklären lassen“
- **Tech-Leads**, die Standards setzen wollen (`AGENTS.md`, Permissions, CI) und dem Team einen gemeinsamen Workflow geben
- **Power-User**, die Plan Mode, Subagents, Worktrees, Headless und MCP nicht nur kennen, sondern *steuern*

Du brauchst keinen Informatik-Abschluss und musst kein Tech-Lead sein. Terminal und ein kleines Repo reichen als Startbasis; Kapitel 2 holt die ersten Git-/Diff-Muskeln nach.

Was du hier nicht findest: Marketing-Versprechen, „10x productivity“-Rhetorik und Kapitel, die nur die offizielle Flag-Liste abtippen. Die Doku listet Optionen. Dieses Buch erklärt *wann* du welchen Hebel ziehst und *welche Fehlerklasse* du damit vermeidest.

Was du in diesem Buch lernst

Grok Build (CLI: `grok`) ist der Coding-Agent von **xAI** (Auth-Oberfläche und OAuth: oft als SpaceXAI sichtbar). Er läuft vor allem als **interaktive TUI** im Terminal, als **Headless**-Lauf für Scripts und CI, und als **ACP-Server** für IDEs und eigene Clients. Überall sitzt dieselbe Engine: ein Modell plus eine Steuerungsschicht aus Tools, Permissions, Memory und Workflows.

Wie das Lernen hier aufgebaut ist: Zuerst baust du ein mentales Modell (Kapitel 1), dann installierst und übst du risikolos (Kapitel 2–3), dann kommen die Produktivitäts-Hebel (Plan Mode, Subagents, Worktrees, `AGENTS.md`). Erst danach Automation, Erweiterungen und der große Use-Case-Block – weil sie ohne Grundlagen schwer zu debuggen sind.

Dieses Buch führt dich stufenweise:

1. **Einstieg** – Lesevertrag, Harness-Mentalmodell, Surfaces
2. **Setup & TUI** – Installation, Auth, Alltag in der Oberfläche
3. **Kern-Workflows** – Plan Mode, Subagents, Worktrees, Projektregeln
4. **Erweiterbarkeit** – Skills, Plugins, Hooks, MCP
5. **Automation** – Headless, CI, ACP/IDE
6. **Steuerung** – Memory, Permissions, Sandbox, Config
7. **Praxis & Nachschlagen** – Use-Cases, Best Practices, Vergleich, Cheatsheets

Du darfst springen. Querverweise greifen, ohne dass du alles linear gelesen haben musst.

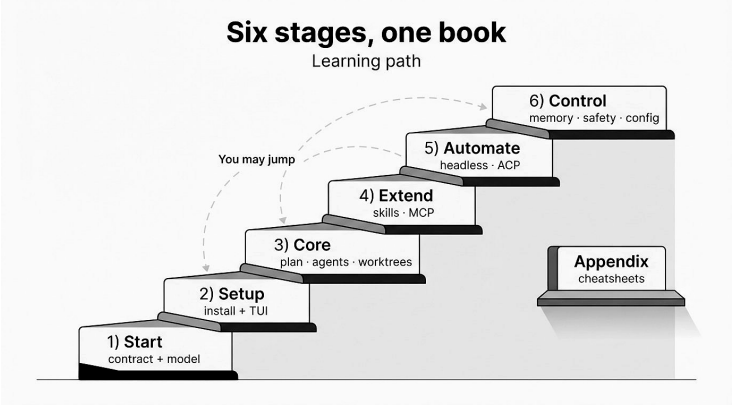


Abbildung 1: Sechs Stufen, ein Buch — springen erlaubt, Querverweise greifen.

Anfänger-Lesepfad (erste Session ~90–120 Min., dann Woche 1)

Wenn Grok Build und agentisches Arbeiten neu für dich sind, nimm diesen **Minimum-Pfad** – nicht das ganze Inhaltsverzeichnis am Tag 1. **Day-1** meint hier die erste Lernsession (Install bis TUI-Grundlagen), nicht „alles an einem Kalendertag“. Die Zeiten sind grob; Auth-Probleme und Neugier fressen Puffer – das ist normal.

Zeit	Was	Erfolg
~45–60 Min.	Vorwort skimmen → Kapitel 1 **skimmen** (Kern + Mini-Glossar 1.11 merken, Rest bei Bedarf) → Kapitel 2 (Install + 15-Min-Parcours mit `beispiele/todo-cli`)	`grok` läuft; ein Read-only-Turn; ein gelesener Diff
~30–45 Min.	Kapitel 3 nur **Day-1-Cockpit** (Fokus, Abbruch, Modes, Sessions 1–2) + Parcours-Pflichtteil	Mode Normal erkannt; einmal bewusst abgebrochen
Diese Woche	Kapitel 4 (Plan Mode) + Kapitel 7 minimal (`AGENTS.md`); optional Use-Case-Variante in 15.2 (todo-cli)	Ein Plan kommentiert oder freigegeben
Später	05–06, 08–14 nach Bedarf; weitere Use-Cases in 15; Symptom→Fix in 16	Wenn der Alltag es verlangt

Tag 1 / Day-1 brauchst du nicht: Subagents, Worktrees, MCP, Headless, Always-approve/YOLO, Custom Models. Die multiplizieren Fehler, bevor Scope und Review sitzen.

Standard-Übungsrepo: `beispiele/todo-cli/` – Mini-Python-CLI im Buch-Repo (Kapitel 2–4). Eigenes Side-Projekt geht auch.

Steckst du fest? Zuerst die Fehlertabelle in Kapitel 2.12, danach Symptom→Fix in Kapitel 16.6.

Warum dieses Buch und nicht nur die Doku? Die offizielle Hilfe und der User-Guide listen Commands und Pfade. Dieses Buch erklärt die *Arbeitsarchitektur*: welche Schicht was steuert, welche Fehlerklasse eine Config vermeidet, und in welcher Reihenfolge du die Knöpfe drehst, ohne dich zu überfordern. **70–75 % der Seiten** sind Praxis: Sessions, Prompts, Checklisten, Fallstricke.

Wie dieses Buch aufgebaut ist

Jedes größere Kapitel endet mit:

- **Was du jetzt kannst** – prüfbare Fähigkeiten
- **Nächste Schritte** – wohin blättern

Größere Abschnitte ergänzen:

- **Checkliste**
- **Häufige Fehler**
- **So baust du das in deinem Projekt nach**

Im PDF erkennst du drei **Callout-Kästen** am linken Farbbalken und einem kleinen Icon:

Kasten	Bedeutung	Wann du handelst
Kern (navy)	Die eine Aussage, die der Abschnitt tragen muss	Merken / als Kompass nutzen
Jetzt selbst machen (teal)	Übung – Terminal auf, nachtippen	Aktiv ausführen, nicht nur lesen
Achtung (burgundy)	Echtes Risiko (Secrets, Prod, Always-approve, destruktive Shell)	Vor dem nächsten Schritt anhalten

Passives Lesen reicht bei **Jetzt selbst machen** nicht – agentische Tools lernt man nur mit Feedbackschleife (Prompt → Tool → Diff → dein Urteil). **Warum-/Wie-Absätze** und Checklisten bleiben normaler Fließtext: absichtlich ohne Icon-Flut.

Weitere Gerüste und Vorlagen: `beispiele/`, `vorlagen/` – von Anfang an kopierfähig. Das Day-1-Übungsrepo ist oben im Anfänger-Lesepfad verlinkt.

Lesevertrag: Dieses Buch erklärt **stabile Konzepte** (Harness, Plan Mode, Subagents, Memory vs. Config, Explore–Plan–Code–Verify). Einzelne Slash-Command-Labels und TOML-Felder können mit Releases driften. Im Zweifel gelten `grok --version`, die eingebaute Hilfe und der User-Guide unter `~/.grok/docs/user-guide/`.

Haltung

Der Ton ist der eines erfahrenen Kollegen: präzise, geduldig, ohne Hype. Wenn etwas riskant ist, steht das da. Wenn eine Config eine Fehlerklasse vermeidet, erklären wir *welche* und *wie*.

Ein Nachmittag, den du vermeiden willst: Du klebst einen Stacktrace in die TUI, akzeptierst drei Diffs ohne Lesen, committest „fix stuff“ – und merkst erst im Review, dass `.env.example` mit echten Keys und ein `rm` im falschen Ordner im Diff lagen. Das Buch trainiert die Gegenbewegung: Scope formulieren, Freigaben lesen, Diffs prüfen, Wissen in Dateien legen (`AGENTS.md`, Plan-Datei, Skills) statt nur im Chat. Nicht weil KI böse ist – weil der Harness ausführt, was das Modell vorschlägt.

Versionshinweis: Features und Slash-Commands entwickeln sich. Konzepte (Permissions, Memory, Tools, Sessions) bleiben stabiler als einzelne UI-Labels. Stand der Fakten im Buch: **Juli–Aug 2026** (CLI **0.2.x**).

Was dieses Buch nicht ist

- **Keine offizielle xAI-Dokumentation** – Produktnamen gehören den Rechteinhabern; bei Abweichungen gilt die aktuelle Hersteller-Doku.
- **Kein Ersatz für Review und Ownership** – du bleibst verantwortlich für Merge und Produktion.
- **Kein Modell-Trainingshandbuch** – wir steuern den Harness; Modellwahl ist ein Hebel unter vielen (Kapitel 14).
- **Kein reines Feature-Inventar** – wenn ein Feature die Nutzung nicht verbessert, bekommt es keinen eigenen Exkurs.

Voraussetzungen in einem Blick

Für den vollen Praxisnutzen brauchst du:

- macOS, Linux oder Windows (nativ / Git Bash / WSL – Details in Kapitel 2)
- Internetzugang für Auth und Modellaufrufe
- Ein Konto bzw. API-Zugang, der Grok Build freischaltet (Browser-Login über grok.com oder `XAI_API_KEY` – Kapitel 2)

- Ein **bekanntes** Übungsrepo – Standard im Buch: `beispiele/todo-cli/` (Kapitel 2); alternativ ein eigenes Side-Projekt (nicht Produktion, nicht der Downloads-Ordner mit dem fremden ZIP)

Hardware-Richtwerte hängen vom Host und vom Modell ab; praxisnah reichen moderne Laptops mit genug RAM für Editor, Terminal und Build deines Projekts.

Wie du prüfst, ob du startklar bist: Nach der Installation `grok --version` (Kapitel 2). Danach trennt du „Binary/PATH“ von „Login/Key“ – spart 20 Minuten Rätselraten.

Harness, nicht Zauberei

Ab hier geht es um die Werkbank: Plan Mode, Permissions, `AGENTS.md`, Subagents und Grenzen. Das Modell liefert Vorschläge – der Harness macht daraus steuerbare Arbeit.



Abbildung 2: Grok Build: Werkbank, nicht Zauberei.

Was du jetzt kannst

- Einordnen, für wen dieses Buch geschrieben ist und welchen Praxis-Anteil es verspricht
- Die drei Callout-Typen (Kern, Jetzt selbst machen, Achtung) und den Kapitel-Rhythmus erklären
- Den **Anfänger-Lesepfad** (Day-1 + Woche 1) und das Standard-Übungsrepo `beispiele/todo-cli` benennen

Nächste Schritte

→ Kapitel 01 – Einführung und Philosophie: Modell vs. Harness, Surfaces und das mentale Modell für alles Weitere. Anschließend Kapitel 2 mit todo-cli.

Kapitel 1

Einführung und Philosophie – was Grok Build wirklich ist

> ⊙ **KERN**
>
> Grok Build ist ein **agentischer Harness** um ein Sprachmodell: Das Modell schlägt vor, der Harness prüft Regeln, führt Tools aus und speist Ergebnisse zurück – und genau diese Schleife steuerst du mit Scope, Modes, Dateien und Review.

1.1 Der Montagmorgen, den du kennst

Du öffnest das Ticket: „Login-Flow bricht ab, wenn der Refresh-Token abgelaufen ist.“ Drei Dateien sind verdächtig, die CI ist rot, der Kollege ist im Urlaub. Früher hättest du gesucht, gelesen, spekuliert, einen Fix skizziert, Tests geschrieben.

Mit Grok Build tippst du im Projektordner `grok` und gibst einen **engen** Auftrag:

```
Untersuche den Refresh-Token-Fehler im Login-Flow.
Lies nur auth- und session-bezogene Dateien.
Ändere noch nichts. Fasse Ursache und 2 Fix-Optionen zusammen.
```

Was passiert dann – und was *nicht*? Das Modell plant Schritte. Der Harness lässt es Dateien lesen und suchen. Du siehst Tool-Aufrufe im Scrollback. Am Ende hast du eine Analyse – **ohne** dass schon Code geschrieben wurde. Das ist der philosophische Kern dieses Buchs: **Zuerst verstehen und steuern, dann handeln lassen.**

Warum der enge Prompt zählt: „Fix the login“ ohne Scope lädt den Agenten ein, halb das Repo umzubauen. Die Junior-Metapher hilft: Du gibst Auftrag, Scope und Definition of Done – nicht „mach mal irgendwas Sinnvolles“.

1.2 Die wichtigste Unterscheidung: Modell vs. Harness

Wenn Marketing „KI schreibt deinen Code“ sagt, verschmelzen zwei Schichten zu einer Phrase. Trenne sie bewusst – sonst optimierst du am falschen Knopf

(lauter prompten statt Plan Mode, `AGENTS.md` und Permissions).

Schicht	Was sie ist	Was sie tut
Modell	Next-Token-Predictor (Grok u. a.)	Erzeugt Text und strukturierte Tool-Aufrufe
Harness / System	Grok Build (TUI, Tools, UI, Config)	Führt Tools aus, prüft Safety/Permissions, speichert Sessions, lädt `AGENTS.md`

Was dieses Buch steuert: vor allem den Harness. Das Modell wählst du je Aufgabe (Kapitel 14); die Steuerknöpfe lernst du einmal.

Wie die Schleife läuft: Das Modell erzeugt die nächste sinnvolle Fortsetzung – oft in Form von *„jetzt lies Datei X“* oder *„jetzt führe pytest aus“*. Der Harness erkennt den Tool-Aufruf, prüft deine Regeln und den aktuellen Mode (Normal, Plan, Always-approve), fragt dich bei riskanten Schritten, führt aus und speist das Ergebnis zurück in den Kontext. Daraus entsteht der Eindruck von Autonomie – obwohl jede „Handlung“ ein freigegebener (oder im Always-approve-Modus vorab freigegebener) Tool-Call ist.

Das Modell „will“ nichts. Es maximiert keine Ziele und hat keine Absicht, dein Repo zu retten. Es erzeugt die fortgesetzte Antwort, die zu Prompt, Code und bisherigen Tool-Ergebnissen passt. **Deshalb** wirken schlechte Anweisungen und fehlende Grenzen so stark: Der Harness führt aus, was das Modell vorschlägt – im Rahmen deiner Regeln.

Wer nur das Modell sieht, überschätzt Magie. Wer nur die Tools sieht, unterschätzt, wie stark schlechte Anweisungen und fehlende Grenzen das Ergebnis ruinieren.

Ein Bild für den Alltag: Das Modell ist der Junior, der vorschlägt. Der Harness ist der Arbeitsplatz mit Tastatur, Repo-Zugriff und Sicherheitsregeln. Ohne Arbeitsplatz bleibt der Junior ein Chat. Ohne Junior bleibt der Arbeitsplatz leer. Produktivität entsteht erst, wenn du *beide* steuerst – und genau das ist der rote Faden der folgenden Kapitel.

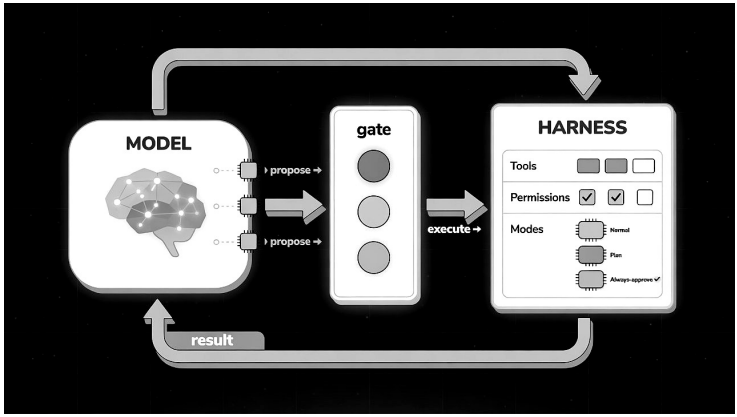


Abbildung 1: Das Modell schlägt vor; der Harness prüft, führt aus und speist Ergebnisse zurück.

1.3 Surfaces: eine Engine, drei Arbeitsmodi

Sobald `grok` läuft (Installation und First-Start: Kapitel 2), stehst du vor der Wahl: interaktive TUI? Einmal-Prompt im Script? IDE über ACP? Die gute Nachricht: Es ist **dieselbe Engine**. Die schlechte: Wer nur klickt oder nur „yolo“ in CI drückt, sieht seltener, *warum* eine Freigabe greift oder eine Regel greift.

Surface	Einstieg (typisch)	Wann sinnvoll
Interaktive TUI	`grok` im Projektordner	Alltag, Review, Lernen, maximale Sichtbarkeit
Headless	`grok -p „…“`	Scripts, CI, Einmal-Jobs ohne TUI
ACP / Agent-Server	`grok agent …`	IDEs, SDKs, langlebige Clients über Agent Client Protocol

Was du heute mitnimmst: Starte zum Lernen in der **TUI**. Headless und ACP später – Config, `AGENTS.md` und viele MCP-Setups reisen mit. Surface wechseln ändert die UI und die Interaktivität, nicht die Grundarchitektur.

TUI: Hier siehst du Tool-Streams, Diffs, Modes und Slash-Commands am klarsten. **Warum starten viele hier?** Weil Plan Mode, Freigaben und Abbrüche weniger „magisch“ wirken und du die Ursache von Fehlern leichter findest.

Headless: Ausgabe, Exit-Code, Automation. **Warum mächtig und riskant?** Ohne interaktive Rückfragen brauchst du ein klares Permission-/Always-approve-Konzept – sonst blockiert der Job oder er tut zu viel (Kapitel 10 und 13).

ACP: JSON-RPC-Clients (IDE, eigene Bots). **Warum nicht als einziger Einstieg?** Du lernst dieselben Konzepte, aber die Feedbackschleife ist eine Schicht weiter weg. Erst TUI-Muskeln, dann Server-Modus (Kapitel 11).

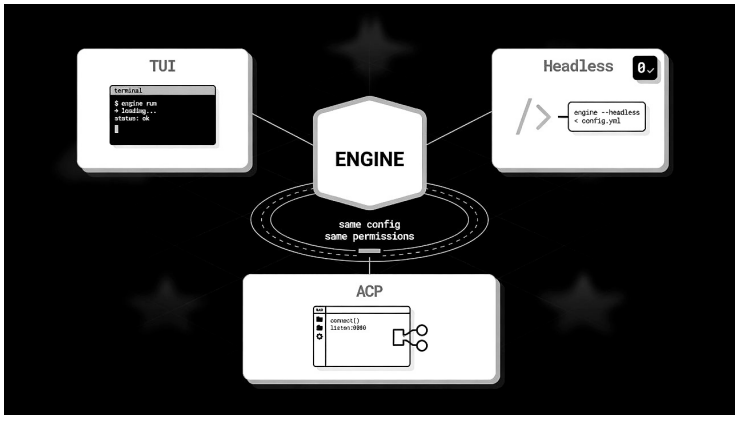


Abbildung 2: Dieselbe Engine — TUI, Headless, ACP. Config und Permissions reisen mit.

1.4 Was Grok Build kann – und was nicht

Grenzen sind keine Marketing-Absätze. Jede Zeile unten schützt vor einer typischen Fehleinschätzung.

Kann (realistisch):

- Codebasen erkunden und erklären – *wie:* Read/Grep/Glob-Tools, nicht „telepathisches Verstehen“
- Features und Fixes über mehrere Dateien **planen** und umsetzen – *warum Plan Mode zählt:* Fehler akkumulieren in langen Ketten
- Tests schreiben und ausführen, bis sie grün sind – *wie:* Shell-Tools + dein Verify-Command in `AGENTS.md`
- Git-Workflows (Diffs, Commits, PRs) mit deiner Freigabe – *warum Freigabe:* Commits und Push sind irreversibel genug für menschliche Ownership

- Arbeit **delegieren** an Subagents und optional in Worktrees isolieren – *wie:* eigener Kontext, parallele Tracks (Kapitel 5–6)
- Externe Systeme über MCP anbinden – *warum riskant und mächtig:* jedes Tool ist Code + oft Credentials
- Wiederholbare Workflows als Skills und Hooks automatisieren – *wie:* Skills on-demand, Hooks erzwungen
- Headless in CI und ACP in der IDE – *wie:* dieselbe Engine, andere Einstiegsfläche

Kann nicht (oder soll nicht allein):

- **Verantwortlichkeit ersetzen** – du bleibst Owner von Merge und Produktion. Ein grüner Agent-Diff ist kein freigegebener Release.
- **Secrets „sicher raten"** – Secrets gehören in Env/Vault, nicht in den Prompt. Was im Kontext landet, kann in Logs, Zusammenfassungen oder Tool-Outputs weiterwandern.
- **Blindes Trust in unbekannte Repos** – unbekannter Code + Agent-Shell ist dasselbe Risikoprofil wie „Skript vom fremden USB-Stick", nur höflicher formuliert.
- **Unendlichen Kontext** – das Kontextfenster ist endlich. Lange Sessions brauchen Compact, neue Sessions und gute `AGENTS.md`, sonst steigen Kosten und sinkt Qualität (Kapitel 12).
- **Produktpolitik ersetzen** – Auth, Kontingente und verfügbare Modelle hängen von deinem Zugang ab (Kapitel 2 und 14).

1.5 Philosophie: Explore → Plan → Code → Verify (danach Commit/PR)

Grok Build liefert bessere Diffs, wenn du nicht „alles auf einmal" forderst, sondern den Tag in **Phasen** schneidest – dieselben Phasen, die gute Teams ohnehin nutzen:

Phase	Ziel	Typischer Grok-Hebel
Explore	Verstehen, nicht ändern	Enger Prompt („ändere nichts"), explore-Subagent

Phase	Ziel	Typischer Grok-Hebel
Plan	Ansatz freigeben	Plan Mode (`/plan`, Shift+Tab), Plan-Datei reviewen
Code	Kleine, reviewbare Diffs	Normal-Mode, klare DoD, `AGENTS.md`
Verify	Beweisen statt hoffen	Tests/Lint im Prompt oder als Skill/Hook
Commit / PR	Ownership	Diff lesen, Message, menschliche Freigabe

Warum diese Reihenfolge? In der Explore-Phase ist ein falscher Edit teuer, weil er Annahmen zementiert. Im Plan Mode ist der Agent **read-only** außer der Plan-Datei – du korrigierst den Ansatz, bevor hundert Zeilen Code entstehen (Kapitel 4). Verify trennt „fühlt sich fertig an“ von „ist grün“.

Mikrogeschichte: Ein Tech-Lead lässt den Agenten „Auth refactorn“. Ohne Plan: 40 Dateien, gemischte Patterns, rote CI. Mit Plan Mode: drei Optionen, eine freigegebene Strategie, dann Implementierung in zwei PRs. Dieselbe Modell-Intelligenz – anderer Harness-Einsatz.

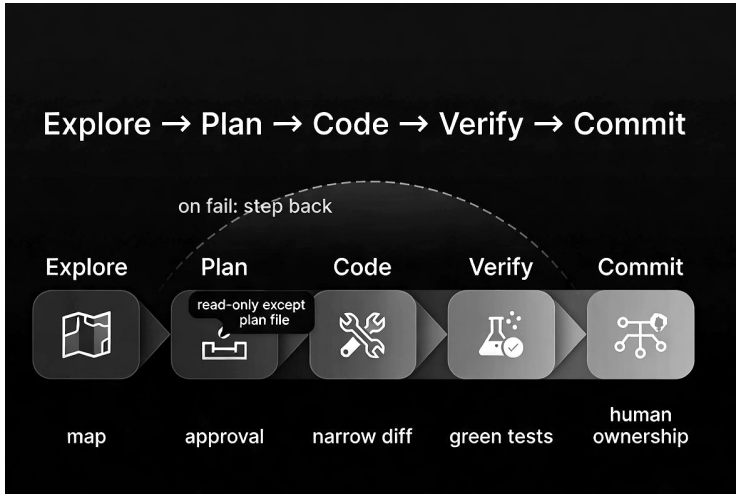


Abbildung 3: Phasengrenzen sind billige Sicherungspunkte — Plan Mode stoppt vor dem Diff.

1.6 Drei Steuerknöpfe, die du früh kennen solltest

Du musst noch nicht jedes Feature auswendig können. Diese drei Knöpfe tragen 80 % des Alltags:

1. **Scope im Prompt** – Dateien, Nicht-Ziele, „ändere noch nichts“ / „nur Tests“ / „kein Git-Push“.
2. **Mode** – Normal für Umsetzung, Plan für Architektur-Ambiguität, Always-approve nur wenn du die Konsequenzen verstehst (CI, enge Sandbox).
3. **Dateien statt Wiederholungen** – stabiles Wissen in `AGENTS.md` und Rules, wiederkehrende Abläufe in Skills, erzwungene Gates in Hooks.

Wie sie zusammenspielen: Der Prompt steuert *diese* Session. `AGENTS.md` steuert *jede* Session im Repo. Config und Permissions steuern, was der Harness **überhaupt** darf. Skills und MCP erweitern, was er *kann*. Wer nur promptet, trainiert Ausdauer – wer Dateien pflegt, trainiert das Team.

```
# Schwacher Einstieg
Mach die App besser und fix bugs.

# Brauchbarer Einstieg (Deutsch)
Im Modul payments: flaky Test test_retry_timeout.
Repro finden, Ursache erklären, minimalen Fix vorschlagen.
Keine API-Änderungen. Nach dem Fix: nur tests/payments ausführen.

# Gleicher Anspruch, oft im Team-Englisch
In payments: flaky test_retry_timeout.
Find repro, explain root cause, propose minimal fix.
No API changes. After fix: run only tests/payments.
```

Beide starken Varianten sind gleich gut – wähle die Sprache, in der dein Team Tickets und Diffs reviewt. **Warum sie gewinnen:** Sie begrenzen den Suchraum, definieren Erfolg und verbieten Scope-Creep. Das Modell halluziniert weniger „Hilfs-Refactors“, weil der Harness-Kontext klar ist.

Schablone zum Abschreiben (ausführlicher in Kapitel 16):

```
Ziel: <was am Ende wahr sein soll>
Nicht-Ziel: <explizit out of scope>
Dateien: <pfade wenn bekannt>
Vorgehen: <explore | plan | implement>
```

```
Verify: <exakter Command, z. B. pytest -q>
Stop: <kein push / warte nach Plan>
```

1.7 Chat, Agent, IDE-Autocomplete – kurze Abgrenzung

Werkzeug-Klasse	Typische Stärke	Typische Grenze
Chat im Browser	Erklären, skizzieren, brainstormen	Kein direkter Repo-Zugriff; du kopierst hin und her
Inline-Autocomplete / Chat in der IDE	Schnelle lokale Edits, Selection-Kontext	Oft schwächer bei langen Multi-File-Ketten und Policy
Grok Build (Agent)	Tool-Loop im Repo, Plan, Subagents, Headless/ACP	Braucht Scope, Review und klare Regeln

Detailvergleich mit Claude Code, Codex und Cursor: Kapitel 17. Hier reicht die Haltung: **Autocomplete beschleunigt Tippen. Ein Agent beschleunigt Arbeit – und multipliziert Fehler, wenn du nicht steuerst.**

1.8 Sicherheit als Feature, nicht als Anhang

> ⚠ **ACHTUNG**
>
> Starte Grok Build nur in Ordnern, die du kennst. Nicht in Downloads mit unbekannten ZIPs. Nicht „mal eben“ auf Produktionsservern. Always-approve und Headless multiplizieren jede zu weite Regel.

Warum das hier schon steht: Ohne dieses Mentalmodell liest du Kapitel 2 als „Installationsanleitung“ und überspringst Trust. Mit diesem Mentalmodell installierst du, übst im sicheren Repo und baust Deny-/Sandbox-Gewohnheiten auf, *bevor* du CI-Keys einbindest.

Drei **Muskeln** für den ersten Tag (Tag-1-Routine – nicht dieselben wie die drei **Steuerknöpfe** in 1.6: Knöpfe = Hebel-Set Scope/Mode/Dateien; Muskeln = Gewohnheit Scope/Freigaben/Diffs):

1. **Scope** formulieren

2. **Freigaben** lesen statt wegklicken
3. **Diffs** prüfen, bevor du committest

Wer diese drei kann, steigt später in Subagents und MCP ein, ohne die Kontrolle zu verlieren.

1.9 Was „gut steuern“ im Alltag heißt

Szene: Feature „Export als CSV“ in einer bestehenden API.

- **Explore:** „Finde bestehende Export- und Streaming-Patterns. Ändere nichts.“
- **Plan:** `/plan` – Endpoint, Auth, Tests, Migrations? Eine Seite, die du kommentierst.
- **Code:** Freigabe, dann Implementierung in kleinen Diffs.
- **Verify:** `pytest tests/export -q` (oder dein Stack) – im Prompt oder als Skill.
- **Persistenz:** Was das Team immer wissen muss → `AGENTS.md`; was nur diese Story betrifft → Ticket/PR.

Häufiger Fehler: In der Code-Phase neue Architektur erfinden, die im Plan nicht stand. **Gegenmittel:** Plan Mode ernst nehmen; Abweichungen zurück in den Plan oder explizit im Prompt freigeben.

Häufiger Fehler: Eine Session für „alles bis Release“. **Gegenmittel:** Neue Session pro klarer Teilaufgabe; Kontext ist endlich (Kapitel 12).

1.10 Checkliste: Mentalmodell sitzt

- ☐ Ich kann Modell und Harness in einem Satz unterscheiden.
- ☐ Ich kenne die drei Surfaces (TUI, Headless, ACP) und starte bewusst in der TUI.
- ☐ Ich formuliere Scope und Nicht-Ziele, bevor der Agent schreibt.
- ☐ Ich weiß, warum Plan Mode vor großen Features kommt.
- ☐ Ich behandle Diffs und Freigaben als Teil der Arbeit, nicht als nervige UI.

1.11 Mini-Glossar für den Einstieg

Spickzettel vor Kapitel 2. Ausführlicher: Kapitel 18.14.

Begriff	In einem Satz
Harness	Schicht ums Modell: Tools, Permissions, Sessions, Modes – was dieses Buch steuert
Tool-Call	Konkreter Handlungsschritt (lesen, shell, edit), den der Harness freigibt und ausführt
Session	Ein Arbeitsgespräch mit Kontext; nicht endlos führen
CWD	Working Directory – Ordner, in dem `grok` die Dateien „sieht“
Diff	Zeilen-Ansicht der Änderung (+/–); vor Allow und vor Commit lesen
Plan Mode	Read-only planen, Plan-Datei freigeben, erst dann Code
Always-approve / YOLO	Fast keine Klick-Freigaben – Deny/Hooks bleiben; Tag 1 aus
Compact	Kontext zusammendrücken (verlustbehaftet) – mit Hinweis, was behalten werden soll
Deny	Harte Regel: dieser Tool-Call wird blockiert, auch unter Always-approve
Folder-Trust	Vertrauens-Gate für project-lokale Hooks/MCP – nur bekannte Repos
MCP / ACP	Externe Tools bzw. IDE/Server-Protokoll – **später** (Kapitel 9 und 11)

Was du jetzt kannst

- Modell und den Harness von Grok Build unterscheiden und die Tool-Schleife skizzieren
- TUI, Headless und ACP einordnen und erklären, was über Surfaces hinweg gilt

- Den Zyklus Explore → Plan → Code → Verify als Steuerungsphilosophie anwenden
- Erklären, warum Scope, Modes und `AGENTS.md` keine Nice-to-haves sind, sondern die Steuerknöpfe des Harness
- Die Day-1-Begriffe aus dem Mini-Glossar in einem Satz erklären

Nächste Schritte

→ Kapitel 02 – Installation und erster Start: Binary installieren, anmelden und die ersten sicheren Aufgaben – Standard: `beispiele/todo-cli`.

Kapitel 2

Installation und erster Start

> ⊙ **KERN**
>
> Typisch in unter einer Viertelstunde (Auth- und PATH-Probleme nicht mitgezählt) installierst du `grok`, meldest dich an und trainierst die drei Muskeln, die den Rest des Buchs tragen: Scope formulieren, Freigaben lesen, Diffs prüfen.

2.1 Was du am Ende dieses Kapitels kannst

Du hast ein lauffähiges Binary, eine funktionierende Auth und eine Session in einem **bekannten** Übungsordner hinter dir. Du weißt, wie du Version und Terminal-Gesundheit prüfst (`grok --version`, `grok doctor` / `/doctor`), und du hast drei sichere Aufgaben durchgespielt, ohne Always-approve und ohne Produktion.

Warum zuerst installieren und üben, bevor Config und MCP kommen: Jede spätere Schicht (Permissions, Headless, Subagents) multipliziert Fehler, die du am Tag 1 noch gar nicht siehst. Ein sauberer First-Start spart Debug-Zeit in Kapitel 10 und 13.

2.2 Installation

Offizielle Empfehlung: der Installer von xAI. Danach startest du mit dem Befehl `grok`.

macOS / Linux / WSL / Git Bash

```
curl -fsSL https://x.ai/cli/install.sh | bash
```

Feste Version pinnen (Platzhalter – nimm die Version, die dein Team pinnen will; aktuell z. B. `grok --version` / Release-Notes, Buch-Stand **0.2.x**):

```
curl -fsSL https://x.ai/cli/install.sh | bash -s 0.2.0
```

Warum pinnen? Teams und CI brauchen reproduzierbare Agent-Versionen. „Immer latest“ ist bequem lokal und riskant in Pipelines.

Windows PowerShell

```
irm https://x.ai/cli/install.ps1 | iex
```

Feste Version:

```
$env:GROK_VERSION="0.2.0"; irm https://x.ai/cli/install.ps1 | iex
```

Der PowerShell-Installer legt typischerweise `%USERPROFILE%\.grok\bin` in den User-PATH. Alternativen: Git for Windows (Git Bash) oder MSYS2 mit dem Bash-Installer oben. **WSL** bekommt das Linux-Binary.
PowerShell vs. CMD vs. Git Bash: Die Install-Befehle sind **nicht** austauschbar. Falsche Shell → kryptische Fehler, obwohl der Rechner „Windows“ heißt.

Zu `curl … | bash` und `irm … | iex`

> ⚠ **ACHTUNG**
>
> Genau dieses Muster ist bei **fremden** URLs riskant. Hier vertraust du dem **offiziellen** xAI-Installer einmal. Im Projekt ordnest du fremde `curl | bash`-Vorschläge des Agenten unter Deny / Ablehnen ein – nicht unter „hilfreich, Allow“.

2.3 Verifizieren und aktualisieren

```
grok --version
```

Update auf die neueste stabile Version:

```
grok update
```

Wie du weißt, dass PATH greift: `which grok` (Unix) bzw. `Get-Command grok` (PowerShell) zeigt auf ein Binary unter deinem Home (häufig `~/.grok/bin/grok` bzw. `%USERPROFILE%\.grok\bin\grok`). Fehlt der Befehl: neue Shell öffnen oder PATH prüfen – nicht sofort neu installieren.
Terminal-Gesundheit (Farben, Clipboard, tmux, Sandbox-Hinweise):

```
grok doctor
# optional maschinenlesbar:
grok doctor --json
```

In einer laufenden Session: `/doctor` – und bei angebotenen Auto-Fixes `/doctor fix` (bzw. gezielte Fixes laut Ausgabe). **Warum doctor zuerst bei „TUI sieht komisch aus“?** Du trennst „Binary fehlt / PATH falsch“ von „Terminal-Capabilities / Clipboard / tmux“ und von „Login/Key fehlt“.

2.4 Anmelden

Browser-Login (Standard)

```
cd /pfad/zu/einem/bekannten/projekt
grok
```

Beim ersten Start öffnet Grok den Browser (grok.com / SpaceXAI-OAuth). Nach dem Login liegen Credentials in `~/.grok/auth.json` und werden über Sessions hinweg wiederverwendet. Tokens werden im Hintergrund refreshed; wenn das nicht mehr geht, wirst du erneut zum Login aufgefordert.

Erneut anmelden / Account wechseln:

```
grok login
```

Flag	Wann
(default / `--oauth`)	Browser-Login über SpaceXAI OAuth (`auth.x.ai`)
`--device-auth` / `--device-code`	Device-Code-Flow für headless/remote Umgebungen

Abmelden:

```
grok logout
```

API-Key (CI, headless, kein Browser)

Key aus der Console (z. B. console.x.ai), dann:

```
export XAI_API_KEY="xai-..."
grok
```

Alltag (Default-Grok): Session-Token aus Browser-Login schlägt `XAI_API_KEY` – deshalb `grok logout`, wenn du **nur** den Key willst.
Custom Models (Kapitel 14): davor können per-model `api_key` / `env_key` in `config.toml` greifen (Reihenfolge grob: `api_key` → `env_key` → Session → `XAI_API_KEY`).

> ⚠ **ACHTUNG**
>
> `~/.grok/auth.json` und MCP-Credentials (`~/.grok/mcp_credentials.json`) sind Dateien mit Secrets (Unix typisch `0600`). Nicht in Tickets, Chats oder Shared-Dirs kopieren. Full-Disk-Encryption und privates `$HOME` / `$GROK_HOME` auf Multi-User-Hosts sind Pflicht, nicht Nice-to-have.

Enterprise / SSO

Firmen-Login (OIDC/SSO) konfigurierst du erst, wenn Browser- oder Key-Auth nicht greift – Details: User-Guide *Authentication* und Kapitel 14. **Am Tag 1 reicht Browser-Login oder `XAI_API_KEY`.**

2.5 Im richtigen Ordner starten

Warum im Projektordner starten? Grok Build arbeitet relativ zum Working Directory (CWD). Im falschen Ordner „sieht“ der Agent die falschen Dateien – oder du reviewst Diffs im falschen Kontext.
Hilfreiche Launch-Varianten:

```
# interaktiv im aktuellen Ordner
grok

# explizites Projektverzeichnis
grok --cwd ~/projects/your-project
```

```
# erste Nachricht direkt mitschicken (wird zum ersten Turn)
grok "Erkläre die Projektstruktur in 10 Bullet Points. Ändere
↪  nichts."

# letzte Session fortsetzen
grok -c

# bestimmte Session fortsetzen
grok --resume <session-id>
```

Was du an Tag 1 nicht brauchst: `--yolo` / `--always-approve`, wilde Worktrees und MCP-Server. Die kommen, wenn Scope und Review sitzen (Kapitel 4–10, 13).

2.6 Erste Sitzung: die TUI in 60 Sekunden

Nach Auth siehst du die Fullscreen-TUI mit zwei Hauptbereichen:

Bereich	Rolle
Scrollback	Verlauf: deine Prompts, Antworten, Tool-Calls, Diffs, Tasks
Prompt	Eingabe unten – hier tippst du Aufträge und `/`-Commands

- `Enter` sendet die Nachricht.
- `Tab` wechselt den Fokus zwischen Prompt und Scrollback.
- Während ein Turn läuft: `Esc` bricht ab (Draft bleibt erhalten; Ausnahme: Fullscreen-Vim-Modus – dort `Ctrl+C`).
- Idle mit leerem Prompt: doppeltes `Esc` (innerhalb ~800 ms) öffnet Rewind; mit Text im Prompt löscht doppeltes `Esc` den Draft.

Details und Edge-Cases: Kapitel 3. Hier reicht: **Prompt fokussieren, kurzen Auftrag senden, Tool-Ausgabe im Scrollback lesen.**

2.7 Permissions am Tag 1: lesen, nicht wegklicken

Standardmäßig fragt Grok nach, bevor Shell-Befehle oder Datei-Edits laufen (Mode **ask** / default). Read-only-Tools und eingebaute harmlose Lese-Shell laufen oft ohne Prompt – der Rest will deine Zustimmung.

Was du im UI erwarten solltest: Bei riskanten Schritten erscheint ein **Freigabe-Dialog** (Wortlaut je Release: erlauben / einmal erlauben / ablehnen o. Ä.). Du siehst den geplanten Befehl oder die Datei-Änderung **bevor** etwas passiert. Bei Edits erscheint im Scrollback ein **Diff-Block** (+/–-Zeilen) – den expandierst du und liest, bevor du „passt“ denkst. Unsicher? Ablehnen ist der sichere Default, kein Scheitern.

Reaktion	Bedeutung
Einmal erlauben	Diese eine Aktion jetzt
Ablehnen	Nein – der Agent muss einen anderen Weg finden
Abbrechen (`Esc` / `Ctrl+C` je nach Zustand)	Stopp, du übernimmst

Anfänger-Regel: Bei Shell-Befehlen, die du nicht in einem Satz erklären kannst → **ablehnen** und nachfragen: „Erkläre den Befehl in einem Satz, ohne ihn auszuführen.“

Always-approve / YOLO (`Ctrl+O`, `/always-approve`, `grok --yolo`) brauchst du an Tag 1 **nicht**. Es spart Klicks und multipliziert Fehler. Für Automation später: Always-approve **plus** Deny-Regeln und Hooks (Kapitel 10 und 13).

Gefährliche Muster (ablehnen oder zumindest nachdenken):

- `rm -rf` und ähnliche Destruktiv-Shell
- `curl … | bash` / unbekannte Remote-Installer
- Lesen/Schreiben von `.env` und Secret-Pfaden
- `git push --force` / History umschreiben
- Unerwartete Netzwerk-Calls mit internen URLs

2.8 Drei sichere Aufgaben (ca. 15 Minuten)

Standard-Übungsrepo des Buchs: `beispiele/todo-cli/` – klein, absichtlich unperfekt, mit fertiger `AGENTS.md`. Eigenes Side-Projekt geht auch; nicht

Produktion, nicht der Downloads-Ordner mit dem fremden ZIP.

Nur PDF, kein Buch-Repo? Entweder ein kleines eigenes Side-Projekt nehmen **oder** das Buch-Repo clonen und die Übungen relativ zum Repo-Root fahren. Die Pfade `beispiele/todo-cli` gelten vom Root dieses Repos aus – nicht vom Home-Verzeichnis.

Setup todo-cli (einmal)

```
cd beispiele/todo-cli   # vom Root des Buch-Repos; sonst absoluten
↪  Pfad nutzen
python3 -m venv .venv
source .venv/bin/activate   # Windows: .venv\Scripts\activate
pip install -e ".[dev]"
pytest -q
grok
```

Beim ersten Start ggf. Folder-Trust für Project-Hooks bestätigen (`/hooks-trust` bzw. Prompt) – Details später; am Tag 1 blockiert fehlendes Trust dich nicht am Erklären und kleinen Edits.

90 Sekunden: Terminal und Diff

Bevor Aufgabe B startet, reicht dieses Minimum:

```
pwd          # wo bin ich? (CWD = Arbeitsverzeichnis von grok)
ls           # was liegt hier?
git status   # optional: geänderte/untracked Dateien
git diff     # optional: Zeilen-Diff – + hinzugefügt, – entfernt
```

Kein Git im Übungsordner? Datei nach dem Edit mit Editor oder `cat NOTES.md` öffnen – der Review-Muskel zählt, nicht das Tool. **CWD** ist das Verzeichnis, in dem `grok` startet; im falschen Ordner sieht der Agent die falschen Dateien.

Aufgabe A – Code erklären lassen (ohne Edits)

Mit todo-cli:

```
Erkläre todo/store.py in einfachen Worten: was speichert die
↪  Datei, wie speichert sie?
Ändere noch nichts. Keine Shell außer lesenden Befehlen.
```

Mit eigenem Repo:

```
Erkläre die wichtigsten Einstiegspunkte dieses Repos in einfachen
↪  Worten.
Ändere noch nichts. Keine Shell außer lesenden Befehlen.
```

Was du beobachten solltest: Lesende Tools (`read_file`, `grep`, `list_dir`), eine Erklärung, **keinen** Edit-Diff. Wenn trotzdem geschrieben wird: abbrechen und den Prompt enger wiederholen.
Warum so formulieren? „Ändere noch nichts“ setzt den Erwartungshorizont. Grok *kann* schreiben – du willst erst verstehen.

Aufgabe B – kleine Datei erzeugen und selbst prüfen

```
Erstelle eine Datei NOTES.md mit genau drei Bullet-Points:
was das Projekt tut, wie man Tests startet, und eine offene Frage.
Keine anderen Dateien anfassen.
```

Danach **selbst** im Terminal (außerhalb oder mit `!` in der TUI – Kapitel 3):

```
cat NOTES.md
# oder:
git diff
```

Warum manuell prüfen? Der Agent kann „fertig“ sagen und trotzdem den Scope verfehlen. Zehn Sekunden Review trainieren das Muskelgedächtnis für große Diffs.

Aufgabe C – Fehler erklären, noch nicht fixen

Wähle **eine** Variante:
Variante todo-cli / Python:

```
python3 -c "import missing_module_xyz"
```

Variante eigener Stack: Nimm einen echten Fehler aus CI oder lokal (komplette Meldung kopieren).

Variante ohne Traceback: Im Prompt beschreiben, was du erwartest vs. was passiert – und den Agenten *nur erklären* lassen, z. B. „Was würde fehlen, damit `todo rm 1` funktioniert? Noch nichts implementieren."

In der Session (bei Traceback):

```
Ich bekomme diesen Fehler:

[kompletten Traceback / Log hier einfügen]

Erkläre die Ursache und schlage einen minimalen Fix vor.
Den Fix noch nicht anwenden.
```

Warum erst erklären, dann fixen? Sonst springt der Agent in Edit-Loops, bevor die Diagnose sitzt. **Wie du den Fehler übergibst:** komplette Meldung – „geht nicht" zwingt zum Raten und verbrennt Kontext.

2.9 Mini-Tour: Slash-Commands und `@` am Tag 1

Slash-Commands steuern den **Harness**, nicht „das Modell umstimmen". Tippe `/` im Prompt – es erscheint ein fuzzy Menü.

Command	Wofür an Tag 1
`/session-info` (Aliases `/status`, `/info`)	Modell, Auth, Kontext grob sehen
`/context`	Kontextfenster-Aufschlüsselung
`/new` / `/clear`	Frische Session
`/resume`	Alte Session laden
`/compact`	Kontext zusammendrücken (mit optionalem Hinweis, was behalten werden soll)
`/doctor`	Terminal/Session-Diagnose
`/always-approve`	**Noch nicht** für den Lern-Parcours
`/quit` / `/exit`	Beenden
`/plan`	ab Kapitel 4 – hier nur merken

Datei anhängen mit `@`:

```
# mit todo-cli
@todo/store.py erkläre Speichern und Laden. Ändere nichts.
@todo/store.py:1-40 was macht dieser Block?

# mit eigenem Repo
@src/main.ts erkläre diese Datei. Ändere nichts.
```

`@` öffnet einen Fuzzy-Picker (respektiert `.gitignore`, versteckt Dotfiles). Mit `!` auch versteckte Pfade: `@!.env` – **vorsichtig** (Secrets im Kontext).

2.10 Project Rules in einem Satz

Lege bei Bedarf eine kurze `AGENTS.md` ins Repo (oder global `~/.grok/AGENTS.md`). Grok injiziert sie als Projekt-Anweisungen. Deeper paths gewinnen; `CLAUDE.md` wird kompatibilitätsweise mitgelesen. Ausführlich: Kapitel 7. Am Tag 1 reicht oft noch gar keine Datei – Scope im Prompt genügt. Minimal-Vorlage: `vorlagen/AGENTS.md.minimal.md`.

2.11 Jetzt selbst machen: 15-Minuten-Parcours

> ☑ **JETZT SELBST MACHEN**
>
> 1. Installer ausführen und `grok --version` verifizieren.
> 2. `grok doctor` einmal laufen lassen; offensichtliche PATH-/Terminal-Findings notieren.
> 3. `beispiele/todo-cli` setuppen (oder eigenes bekanntes Repo) und `grok` starten; Auth abschließen.
> 4. Aufgabe A (nur erklären) – kein Diff.
> 5. Aufgabe B (`NOTES.md`) – danach `git diff` oder Datei lesen.
> 6. Aufgabe C (Fehler erklären) – kein Fix anwenden.
> 7. `/session-info` und `/context` einmal ansehen.
> 8. Session mit `/quit` beenden; optional `grok -c` testen.
>
> **Erfolg:** Du hast mindestens einen erlaubten und einen abgelehnten Tool-Call bewusst erlebt – und einen Diff gelesen, bevor du „passt“ gesagt hast.

2.12 Häufige Fehler

Symptom	Typische Ursache	Erster Check
`grok: command not found`	PATH / Shell nicht neu gestartet	`which grok`, neue Terminal-Tab, Installer-Log
Login-Schleife / sofort wieder raus	Token abgelaufen, kaputte `auth.json`	`grok logout && grok login`
Key greift nicht	Session-Token hat Vorrang	`grok logout`, dann nur `XAI_API_KEY`
TUI ohne Farbe / kaputte Keys in tmux	Terminal-Capabilities	`grok doctor`, `/doctor`, Guide *Terminal support*
Agent schreibt, obwohl „nur erklären“	Prompt zu weich	Scope wiederholen, Turn abbrechen, engere Formulierung
Zu viele Permission-Prompts	Erwartetes ask-Verhalten	Nicht sofort YOLO – gezielte Allows und später Config (Kap. 13)
„Falsches Repo“	CWD falsch	`pwd`, `grok --cwd …`

Steckst du länger fest? Tabelle oben zuerst; später Symptom→Fix in Kapitel 16 (Abschnitt 16.6).

2.13 Checkliste First-Start

- ☐ `grok --version` zeigt eine Version
- ☐ `grok doctor` / `/doctor` ohne blockierende PATH-Fehler
- ☐ Auth (Browser oder Key) funktioniert
- ☐ Session im bekannten Übungsrepo, nicht in Produktion
- ☐ Mindestens ein Read-only-Auftrag ohne Edits
- ☐ Mindestens ein Edit mit manuellem Diff-Check
- ☐ Always-approve noch aus
- ☐ Keine Secrets in Prompts oder committeden Dateien

Was du jetzt kannst

- Grok Build installieren, updaten und mit `grok --version` / `grok doctor` verifizieren
- Browser-Login und API-Key-Auth unterscheiden und Credentials sicher behandeln
- Eine TUI-Session im richtigen CWD starten und drei sichere Aufgaben durchführen
- Permission-Prompts und die wichtigsten Day-1-Slash-Commands einordnen

Nächste Schritte

→ Kapitel 03 – Die interaktive TUI im Detail: Scrollback, Shortcuts, Modes, Queue und der tägliche Bedienrhythmus.

Kapitel 3

Die interaktive TUI im Detail

> ⊙ **KERN**
>
> Die TUI ist dein Cockpit: Scrollback zeigt die Wahrheit (Tools, Diffs, Denken), der Prompt steuert den nächsten Schritt – und wer Fokus, Abbruch und Modes beherrscht, spart Stunden „der Agent macht komische Dinge“.

3.1 Warum die TUI zuerst meistern

Headless und ACP (Kapitel 10–11) nutzen dieselbe Engine. In der TUI siehst du aber **live**, was der Harness tut: welcher Tool-Call, welcher Diff, welche Freigabe, welcher Mode. Wer nur Headless lernt, debuggt Permissions später blind.

Alltagsschmerz: Du tippst einen langen Auftrag, der Agent startet, du merkst den Fehler im Prompt – und findest `Esc` / `Ctrl+C` nicht. Oder du scrollerst im Diff und landest versehentlich im Always-approve-Mode. Dieses Kapitel macht die Gesten bewusst.

Zwei Lesegeschwindigkeiten (Day-1 = erste Lernsession, siehe Vorwort):

Pfad	Kurz	Wann
Day-1-Cockpit (~20 Min.)	Fokus, Abbruch, Modes, `@` / `!`, Kurz-Slash, Sessions 1–2	Grok Build neu; nach Kapitel 2
Referenz / Vertiefung	Vim, Navigation, Queue, volle Shortcuts, Sessions 3–4	Stecken geblieben oder Alltag ausreizen

Day-1 im Detail (Reihenfolge):

1. 3.2–3.3 Anatomie und Fokus
2. 3.4 nur Merksatz Esc vs. Ctrl+C (Tabelle skimmen)
3. 3.5 Modes
4. 3.7 nur `@` und `!`
5. 3.8 Slash-Kurzreferenz

6. 3.11–3.13 Sessions 1–2 und Parcours-Pflichtteil

Referenz bei Bedarf: 3.4 Edge-Cases (Vim-Esc, Steal-Esc); 3.6; History/Multiline/Queue in 3.7; 3.9–3.10; Sessions 3–4.
Du musst die Shortcut-Tabellen nicht auswendig können, bevor Plan Mode (Kapitel 4) kommt. **Muskeln zuerst:** Fokus, Abbruch, Mode erkennen, Diff lesen. Was du heute änderst: Day-1-Pfad fahren, Rest bookmarken.
Übungsrepo: weiter `beispiele/todo-cli/` aus Kapitel 2 – oder dein Side-Projekt.

3.2 Anatomie der Oberfläche

Nach dem Login siehst du im Wesentlichen **zwei Zonen** plus Overlays: den Scrollback oben und den Prompt unten. Wer die Rollen kennt, sucht Shortcuts nicht im falschen Fokus und liest Diffs dort, wo sie erscheinen.

Teil	Was du siehst	Wofür
Scrollback	Prompts, Antworten, Thinking, Tool-Calls, Diffs, Tasks	Lesen, navigieren, collapsen, kopieren
Prompt	Eingabezeile unten	Aufträge, `/`-Commands, `@`-Anhänge, `!`-Shell
Overlays	Slash-Menü, File-Picker, Settings, Session-Picker, …	Harness steuern, ohne den Chat zu „überreden“

Was du mitnimmst: Scrollback = Review-Oberfläche; Prompt = Steuerung; Overlays = Harness-Menüs. Sticky User-Prompts im Scrollback helfen, Turns zu scannen: „Wo habe ich den Scope gesetzt?“ ist oft die bessere Frage als „Warum ist das Modell dumm?“.
Thinking-Blöcke (wenn das Modell Reasoning liefert) sind collapsbar – nützlich zum Nachvollziehen, teuer im Scroll-Rauschen. `Ctrl+E` expandiert/collapst Thinking global (Kapitel-Shortcuts unten).
Tool-Calls streamen in Echtzeit: Shell-Output, Suchtreffer, Inline-Diffs bei Edits. Das ist deine Review-Oberfläche, nicht nur Dekoration.

3.3 Fokus: Prompt vs. Scrollback

Geste	Wirkung
`Tab`	Fokus wechseln Prompt ↔ Scrollback
`Space` (Scrollback fokussiert, Simple Mode)	Prompt fokussieren
`Enter` (Prompt fokussiert)	Nachricht senden

Warum Fokus trainieren? Viele Shortcuts gelten nur im Scrollback (Navigation, Fold, Copy). Tippe „blind“ im Scrollback-Modus, landet Text im Prompt – oder gar nicht, je nach Mode. Faustregel: **Schreiben → Prompt fokussiert. Lesen/Review → Scrollback fokussiert.**

3.4 Abbruch, Clear, Rewind – Esc und Ctrl+C

Das ist der Abschnitt, den du zweimal liest, wenn der erste „hängende“ Turn kommt.

Zustand	Geste	Wirkung
Turn läuft (Default / kein Fullscreen-Vim)	`Esc`	Turn **sofort abbrechen**, Draft **bleibt**
Turn läuft, Fullscreen + Vim-Mode	`Esc`	**kein** Cancel – nutze `Ctrl+C` o. Ä.
Turn läuft, Draft nicht leer	`Ctrl+C`	zuerst Draft leeren; zweites `Ctrl+C` auf leerem Prompt cancelt
Idle, Prompt nicht leer	2× `Esc` (≲800 ms)	Prompt leeren (Text landet in History)
Idle, Prompt leer, es gibt Messages	2× `Esc`	**Rewind**-Picker (wie `/rewind`)
Nach Esc-Cancel ~1 s	doppeltes Esc	Rewind-Arm unterdrückt – kein versehentliches Rewind

Esc vs. Ctrl+C (Merksatz): Esc bricht den laufenden Turn ab und schützt den Draft. Ctrl+C ist „erst aufräumen, dann stoppen“ und eskaliert Richtung Quit, wenn du weiterdrückst.

Steal-Esc: Offene Menüs, Dropdowns, Selection und Modi (`!` Shell, `#` Remember, Feedback) fangen Esc zuerst ab – erwartbar, sobald du Overlays nutzt.

3.5 Session-Modes: Normal, Plan, Always-approve

`Shift+Tab` im fokussierten Prompt **cyclt** die Session-Modes: **Normal → Plan → Always-approve → Normal**. **Auto** ist **nicht** in diesem Zyklus – dafür `/auto` (Feature-abhängig).

Hebel	Mode / Wirkung
`Shift+Tab`	Mode cyclen (Normal / Plan / Always-approve)
`Ctrl+O`	Always-approve (YOLO) toggeln
`/always-approve`	Always-approve toggeln
`/auto`	Auto-Mode (Classifier; gefährliche Calls können weiter fragen) – Feature-abhängig
`/plan`	Plan Mode betreten (Kapitel 4)
`/settings` / `F2`	Settings-Modal

Terminal-Hinweis: Auf **Apple Terminal** kann `Ctrl+O` primär **Interject/Send-now** belegen (kein modified Enter) – dann YOLO über `/always-approve` oder `/settings`. Details: User-Guide *Terminal Support*.

Was die Modes bedeuten (Arbeitsdefinition):

Mode	Praxis
Normal (Permission oft *ask*)	Alltag: lesen oft frei, Schreiben/Shell oft mit Freigabe
Plan	Read-only Planung (außer Plan-Datei) – Ansatz freigeben, bevor Code fließt
Always-approve	Fast keine Permission-Prompts; Deny-Regeln und Hooks greifen weiter
Auto	Weniger Prompts durch Safety-Checks; blockierte Calls scheitern/eskalieren

> ⚠ **ACHTUNG**
>
> Always-approve ist kein „sicherer Turbo“. Es ist ein Turbo **ohne Klick-Bremse**. In der Lern- und Review-Phase aus; in CI nur mit Deny/Hooks und engem Scope (Kapitel 10, 13).

Plan + Always-approve: Plan Mode blockiert Repo-Edits; riskante Shell und andere Non-Edit-Tools können unter Always-approve trotzdem laufen – Details und Subagent-Gate in Kapitel 4.

Mikrogeschichte: Du cyclst versehentlich auf Always-approve, der Agent installiert eine Dependency und pusht einen Branch. Diff war „grün“ im Scrollback – Ownership war weg. Gegenmittel: Mode-Anzeige im UI bewusst lesen; nach Experimenten `Shift+Tab` zurück auf Normal.

Day-1-Cockpit bis hier: Anatomie, Fokus (`Tab`), Abbruch (`Esc`), Modes (Normal / Plan / Always-approve aus).

Day-1 – so weiter (nicht linear alles lesen): springe zu **3.7** (`@` und `!`) → **3.8** (Slash-Kurzreferenz) → **3.11–3.12** (Sessions 1–2) → **3.13** (Parcours Pflichtteil). Abschnitte **3.6**, **3.9–3.10** und die Vertiefungs-Unterabschnitte in 3.7 (History, Multiline, Queue) sowie Sessions 3–4 **überspringen** – bei Bedarf später nachschlagen.

3.6 Navigation und Fold im Scrollback

Referenz / Vertiefung – am Tag 1 optional (Day-1-Leser: weiter bei 3.7).

Simple Mode (Default): Pfeiltasten navigieren; Buchstaben fokussieren oft den Prompt und tippen.

Vim Mode (opt-in): `/vim-mode` oder `[ui] vim_mode = true` in `config.toml`. Dann `j`/`k`, `h`/`l`, `g`/`G`, `y`/`Y` usw. im Scrollback.

Aktion	Simple	Vim (Scrollback)
Nächster / vorheriger Eintrag	`↓` / `↑`	`j` / `k`
Nächster / vorheriger User-Turn	`Shift+→` / `Shift+←`	`L` / `H` (Shift)
Eintrag einklappen / ausklappen	`←` / `→`	`h` / `l`
Alle Thinking toggeln	`Ctrl+E`	`Ctrl+E`
Block-Inhalt kopieren	(Vim: `y`)	`y`

Aktion	Simple	Vim (Scrollback)
Metadaten kopieren (z. B. Shell-Cmd)	(Vim: Y)	Y
Fullscreen-Viewer	Enter	Enter
Seite scrollen	PageUp / PageDown	gleich

PageUp/PageDown scrollen die Conversation oft auch bei Prompt-Fokus, ohne den Draft zu stehlen – praktisch beim Nachlesen während du den nächsten Prompt baust.

Warum Fold? Lange Shell-Logs und riesige Diffs füllen den Viewport. Collapsen hält den Faden; Expand nur für Review. In Vim-Mode ⇧E expandiert/collapst alles.

3.7 Prompt-Power: @, !, History, Multiline, Queue

Day-1: die beiden Unterabschnitte @ und !. History, Multiline und Queue sind Vertiefung.

Dateien mit @

```
# Day-1 mit todo-cli
@todo/store.py erkläre Speichern und Laden. Noch nichts ändern.
@todo/cli.py:1-40 was macht der Einstieg?

# später / eigenes Repo
@src/api/auth.ts prüfe die Fehlerbehandlung. Noch nichts ändern.
@!.github/workflows/ci.yml erkläre den Job test.
```

Fuzzy-Picker, .gitignore-aware. ! **ohne Leerzeichen** direkt nach @ für Dotfiles/versteckte Pfade (@!.env, @!.github) – **Secrets nicht achtlos anhängen**.

Shell-Mode mit !

Auf leerem Prompt ! tippen → Shell-Mode. Du führst Befehle **selbst** aus (nicht der Agent-Turn). Nützlich für git status, pytest, schnelle Checks, ohne den Agenten zu bitten.

Prompt-History *(Vertiefung)*

- Leerer Prompt: ↑ öffnet History mit dem letzten Prompt.
- `/history` für Fuzzy-Suche.
- Recalled `!`-Befehle landen wieder im Shell-Mode.

Multiline *(Vertiefung)*

Hebel	Wirkung
`/multiline` oder `Ctrl+M` (Prompt fokussiert)	Multiline toggeln
Multiline an: `Enter`	Neue Zeile
Multiline an: `Shift+Enter` / `Alt+Enter`	Senden

Wann Multiline? Lange Specs, Stacktraces, mehrere Constraints – ohne dass Enter den Turn zu früh startet.

Prompt-Queue *(Vertiefung)*

Während ein Turn läuft, kannst du Folge-Prompts **queuen**. Pane toggeln: `Ctrl+;` (Alternativen je Plattform: `Ctrl+'`, unter lokalem macOS-VS-Code-Family oft `Ctrl+4`). Mid-turn leeres `Enter` kann den obersten Queue-Eintrag force-senden – bewusst einsetzen.

3.8 Slash-Commands: die tägliche Kurzreferenz

Vollständige Liste: User-Guide und Kapitel 18. Hier die Befehle, die den Alltag tragen:

Command	Zweck
`/new` (`/clear`)	Frische Session
`/resume`	Session-Picker
`/compact [Hinweis]`	Kontext komprimieren
`/context`	Token-/Kategorie-Breakdown

Command	Zweck
`/session-info` (`/status`, `/info`)	Session-Metadaten
`/rewind`	Zu früherem Turn zurück
`/fork`	Session ab hier verzweigen
`/model` (`/m`), `/effort`	Modell / Reasoning-Effort
`/plan`, `/view-plan`	Plan Mode (Kapitel 4)
`/always-approve`, `/auto`	Permission-Modes
`/vim-mode`, `/minimal`, `/fullscreen`	Bedien-/Render-Modi
`/copy`, `/export`	Antwort sichern
`/doctor`	Diagnose
`/hooks`, `/plugins`, `/skills`, `/marketplace`	Extensions-Modal
`/quit` (`/exit`)	Beenden

Skills mit `user-invocable: true` erscheinen ebenfalls im `/`-Menü – dann ist `/mein-skill` ein Workflow, kein Builtin (Kapitel 8).
Command Palette: `Ctrl+P` oder `?` – wenn du den Slash-Namen vergessen hast.

3.9 Wichtige Agent-Shortcuts (eine Tabelle zum Üben)

Referenz – ausdrucken oder bookmarken; nicht auswendig vor dem ersten Plan Mode.

Key	Aktion
`Ctrl+P` / `?`	Command Palette
`Ctrl+M`	**kontextabhängig:** Model-Picker (Prompt **nicht** fokussiert) **oder** Multiline (Prompt fokussiert)
`Ctrl+O`	Always-approve toggeln (Apple Terminal: oft Interject – dann `/always-approve`)
`Ctrl+S`	Session-Picker (Resume)
`Ctrl+N`	Neue Session (wo gebunden)
`Shift+Tab`	Mode cyclen Normal/Plan/Always-approve (Prompt fokussiert)

Key	Aktion
`Ctrl+T`	Todos-Pane
`Ctrl+G`	Tasks-Pane (Fullscreen-TUI); in Minimal-Mode oft External-Editor
`Ctrl+B`	Laufenden Foreground-Command in den Background
`Ctrl+.` / `Ctrl+X`	Keyboard-Shortcuts-Hilfe (`Ctrl+X` ist der robuste Fallback)
`F2` / `Ctrl+,`	Settings

Terminal-Fallen: In tmux, VS-Code-integriertem Terminal oder ohne Kitty-Keyboard-Protocol greifen manche Chords nicht. Dann: `/doctor`, Shortcuts-Hilfe, und `Ctrl+X` statt `Ctrl+.`.

3.10 Render-Modi: Fullscreen vs. Minimal

Referenz / optional. Am Tag 1 bleibst du im **Fullscreen**-Default.

Mode	Charakter
Fullscreen (Standard-TUI)	Alt-Screen, klassisches Agent-UI
Minimal	Experimenteller scrollback-nativer Modus

Umschalten: `/minimal`, `/fullscreen` (Alias `/full`), CLI `--minimal` / `--fullscreen`. Sticky-Verhalten über Default screen mode in Settings / `[ui] screen_mode` – CLI-Flags und Slash-Switch können session-scoped sein; Details im User-Guide *Getting Started* / *Slash Commands*.

Wann Minimal? Wenn du Scrollback-natives Lesen bevorzugst. **Wann Fullscreen?** Default-Lernpfad und die meisten Shortcuts-Beschreibungen in diesem Buch.

3.11 Session-Lifecycle im Alltag

```
grok                  # neue interaktive Session im CWD
grok -c               # continue last
grok --resume ID      # konkrete Session
```

In der TUI:

Ziel	Mittel
Neu	`/new`, oft `Ctrl+N`
Fortsetzen	`/resume`, `Ctrl+S`
Umbenennen	`/rename` / `/title`
Verzweigen	`/fork`
Zurückspulen	`/rewind` oder 2× Esc (leer)
Kontext retten	`/compact keep die API-Entscheidungen`
Kontext prüfen	`/context`, `/session-info`

Sessions liegen unter `~/.grok/sessions/` (Kapitel 12). **Warum nicht alles in einer Session?** Kontextfenster und Widersprüche. Frischer Kontext ist oft billiger als „bitte vergiss den alten Plan“.

3.12 Praxis-Sessions (2–4)

Day-1: Sessions 1–2. Sessions 3–4, sobald Abbruch und Modes sitzen.

Session 1 – Nur lesen, Tools beobachten

Mit todo-cli:

```
@todo/store.py erkläre Speichern und Laden in einfachen Worten.
↪  Ändere nichts.
```

Oder generisch:

```
Liste die Top-Level-Struktur und nenne die drei Dateien,
die ein neuer Entwickler zuerst lesen sollte. Ändere nichts.
```

Trainingsziel: Scrollback lesen, Tool-Calls collapsen, keine Freigabe für Writes. Optional `@README.md` anhängen.

Session 2 – Edit mit bewusster Freigabe

```
Ergänze in NOTES.md einen vierten Bullet „Nächster Lernschritt:
↪  Plan Mode“.
(Falls NOTES.md fehlt: mit genau diesem einen Bullet anlegen.)
Nur diese Datei. Danach git-diff-fähig halten.
```

Trainingsziel: Permission-Prompt lesen, Diff im Scrollback expandieren, manuell `git diff` (Prompt oder `!`).

Session 3 – Abbruch und Nachschärfen *(Vertiefung)*

Mit todo-cli absichtlich zu breit:

```
Refactore die gesamte todo-cli auf ein neues Persistenz-Framework.
```

Oder generisch:

```
Refactore das gesamte auth-Modul auf das neueste Pattern.
```

Sobald der Agent zu breit sucht oder den ersten riskanten Befehl vorschlägt: **`Esc`**. Dann:

```
Stop. Scope enger: nur store.py – erkläre atomare Writes in einem
↪  Absatz.
Kein Refactor, kein neues Framework.
```

Trainingsziel: Abbruch ohne Scham; Prompt heilen statt „weiterwursteln“.

Session 4 – Mode-Bewusstsein *(Vertiefung)*

1. `Shift+Tab` bis Plan Mode (oder `/plan`).
2. Auftrag (todo-cli): „Plane atomare Writes in `store.save` – noch kein Code.“
3. Plan lesen; **nicht** implementieren lassen, bis Kapitel 4.
4. Zurück auf Normal.
5. `Ctrl+O` **kurz** testen und **sofort** wieder aus – spüre den Unterschied in den Prompts.

Trainingsziel: Mode ist ein Harness-Hebel, kein Dekor.

3.13 Jetzt selbst machen: TUI-Parcours (20 Minuten)

☑ **JETZT SELBST MACHEN**

Day-1 (Pflicht, ~15 Min.):

1. `grok` in `beispiele/todo-cli` (oder eigenem Übungsrepo); `/session-info` und `/context` öffnen.
2. Session 1 (nur lesen) – einen Tool-Block collapsen und wieder expandieren.
3. Mit `@` eine Datei anhängen und eine gezielte Frage stellen.
4. Session 2 (kleiner Edit) – Diff lesen, Freigabe bewusst geben.
5. Einen absichtlich schlechten Prompt senden und mit `Esc` abbrechen; nachschärfen.

Optional (Vertiefung):

6. `/history` oder ↑ nutzen, um einen alten Prompt zu recyclen.
7. Multiline an, Stacktrace einfügen, senden.
8. `/doctor` einmal ausführen und ein Finding notieren (auch wenn alles grün ist).
9. Optional: `/vim-mode` 5 Minuten, dann zurück.

3.14 Häufige Fehler

Symptom	Ursache	Fix
Tasten landen „nirgendwo“	Fokus im Scrollback / Overlay	`Tab`, Esc schließt Overlay
Esc bricht nicht ab	Fullscreen-Vim	`Ctrl+C` oder Vim aus
Versehentlich YOLO	`Ctrl+0` / Shift+Tab	Mode zurück; Diffs prüfen
Context „voll“ / Drift	Lange Session	`/compact` mit Hinweis oder `/new`
Shortcuts tot in tmux/VS Code	Terminal-Protokoll	`/doctor`, `Ctrl+X`, Guide Terminal
Agent ändert zu viel	Scope fehlt	Abbrechen, enger Prompt, Plan Mode

---

3.15 Checkliste TUI-Alltag

- ☐ Ich wechsle bewusst Prompt ↔ Scrollback (`Tab`)
- ☐ Ich kann einen Turn mit Esc/Ctrl+C zuverlässig stoppen
- ☐ Ich lese Diffs und Shell-Befehle vor dem Allow
- ☐ Ich kenne meinen aktuellen Mode (Normal / Plan / Always-approve)
- ☐ Ich nutze `@` statt „lies irgendwas im Repo“
- ☐ Ich weiß, wann `/compact` vs. `/new`
- ☐ `/doctor` ist mein erster Schritt bei kaputter TUI

Was du jetzt kannst

- Scrollback und Prompt gezielt bedienen, abbrechen und rewinden
- Modes (Normal, Plan, Always-approve, Auto) und ihre Risiken einordnen
- Die Alltags-Slash-Commands und – bei Bedarf – die Shortcut-Referenz nutzen
- Mindestens zwei kurze Sessions fahren, die Review- und Scope-Muskeln trainieren

Nächste Schritte

→ Kapitel 04 – Plan Mode, Review und Freigabe: vor dem Code den Ansatz freigeben – und Diffs danach kontrolliert abnehmen. Stecken geblieben bei PATH/Login/TUI? Kapitel 2.12 und später 16.6.

Kapitel 4

Plan Mode – Review und Freigabe

> ⊙ **KERN**
>
> Plan Mode trennt **Ansatz** und **Code**: Der Agent erforscht read-only (außer der Plan-Datei), du kommentierst und freigibst – erst danach darf Implementierung den Diff-Strom füllen.

4.1 Warum Plan Mode existiert

Alltagsschmerz: Du sagst „Bau Auth ein“. Der Agent wählt JWT, legt Middleware an, rührt die Session-DB an – und erst im Review merkst du: Das Team nutzt Cookie-Sessions und einen bestehenden Guard. Drei Stunden Rework, obwohl die Anforderung „Auth“ hieß.

Plan Mode erzwingt die Gegenbewegung: **Zuerst Ansatz auf eine Seite bringen, dann bauen.** Das spart nicht „Token“, sondern falsche Architektur.

Wann es sich lohnt: echte Ambiguität (mehrere vernünftige Wege), unklare Requirements, hoher Blast Radius. **Wann nicht:** offensichtlicher Bugfix, Rename, Feature das 1:1 bestehende Patterns kopiert, reines Explore (dafür `explore`-Subagent, Kapitel 5).

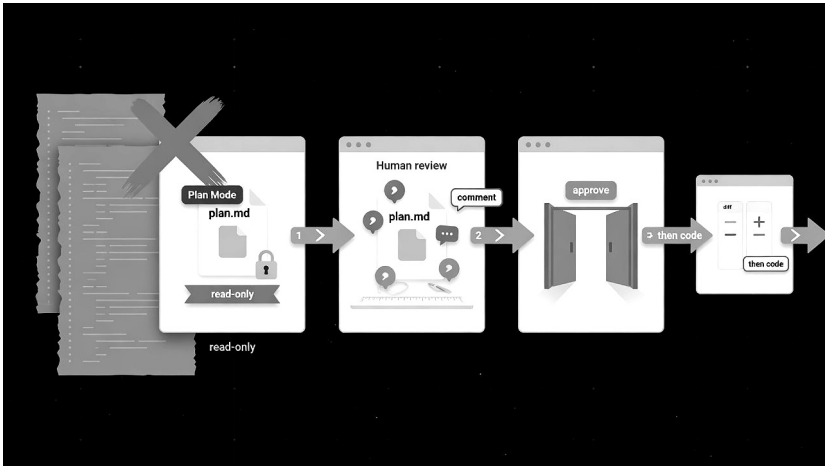


Abbildung 1: Read-only planen, Plan-Datei kommentieren, erst dann implementieren.

4.2 Was Plan Mode technisch tut

Wenn Plan Mode **aktiv** ist:

1. Der Agent liest und sucht im Code (Patterns, bestehende Utils).
2. Er schreibt den Ansatz in die **Plan-Datei**.
3. Optional stellt er Klärungsfragen (`ask_user_question`).
4. Er beendet die Planungsphase mit `exit_plan_mode` und präsentiert den Plan zur Freigabe.

Die harte Regel: Read-only für fast alles. Edits an der Plan-Datei (`plan.md` im Session-Verzeichnis) sind auto-approved. Edits an **jeder anderen Datei** werden **abgelehnt** – der Tool-Call scheitert mit dem Hinweis, dass nur die Plan-Datei editierbar ist. Das gilt in **jedem** Permission-Mode, **inklusive Always-approve**.

Erlaubt in Plan Mode	Blockiert (Edit-Tools)
Lesen, Suchen, Listen	`search_replace` / Write auf Repo-Dateien

Erlaubt in Plan Mode	Blockiert (Edit-Tools)
Plan-Datei schreiben	Implementierungs-Diffs im Working Tree
Shell lesen/erkunden (siehe Fallstrick unten)	–

> ***Achtung — zwei getrennte Risiken:***
>
> 1. ***Edit-Tools ≠ Shell.*** *Plan Mode blockiert Repo-Edits über Edit-Tools, nicht magisch jede Shell-Redirection. Bash wird nicht als Dateischreiber inspiziert.*
>
> 2. ***Parent-Plan ≠ Kind-Gate.*** *Subagents erben die Plan-Mode-Edit-Sperre des Parents* ***nicht****. Ein* `general-purpose`*-Kind kann schreiben, während der Parent noch plant – und erbt oft den Permission-Mode (inkl. Always-approve). Für reine Recherche:* `explore` */* `plan` */* `capability_mode: read-only` *(Kapitel 5–6).*

Plan-Datei-Pfad (typisch):

```
~/.grok/sessions/<encoded-cwd>/<session-id>/plan.md
```

`<encoded-cwd>` ist ein kodierter Verzeichnisname, nicht der Literalpfad deines Repos.

Session-`plan.md` vs. Team-Plan-Doku: Die Datei oben ist **session-lokal** (Harness-Artefakt). Team-Vorlagen wie `vorlagen/PLAN.md.skeleton.md` oder `beispiele/todo-cli/PLAN.md.example` sind **Repo-Doku** – nicht dieselbe Datei und kein Ersatz für den Plan-Mode-Lifecycle.

Inhalt der Plan-Datei (erwartbar):

- **Context** – warum die Änderung
- Empfohlener Ansatz (nicht jede Alternative)
- Kritische Dateipfade
- Bestehende Funktionen/Utils zum Wiederverwenden (mit Pfaden)
- **Verification** – wie du end-to-end testest

4.3 Plan Mode betreten

Du willst den Mode setzen, **bevor** der nächste Prompt fliegt – sonst startet der Agent schon im falschen Gate.

Du startest (user-initiated)

Weg	Wirkung
`/plan`	Plan Mode **Pending** – aktiv mit dem **nächsten** Prompt
`/plan <Beschreibung>`	Plan Mode + sofortiger Turn mit dieser Beschreibung
`Shift+Tab`	Mode cyclen: Normal → **Plan** → Always-approve → Normal (**Auto** separat: `/auto`)

Lifecycle (vereinfacht):

```
Inactive  --(/plan oder Shift+Tab)-->  Pending
Pending   --(erster Prompt)--------->  Active
Active    --(Approve exit_plan_mode)--> Inactive (+
↪  Implementierung)
Active    --(Mode aus, idle)--------->  Inactive
Active    --(Mode aus, Turn läuft)-->  ExitPending → dann Inactive
```

Agent-initiiert: Der Agent ruft `enter_plan_mode` auf – **du musst zustimmen**. Ablehnen = bleibt Normal. Bei Zustimmung springt der Flow oft **Inactive → Active** (ohne Pending).

Gute Trigger (für dich und den Agenten):

- „User-Auth einbauen“ / „Add user authentication“ (Session vs. JWT vs. Middleware)
- „Datenpipeline neu schneiden“ / „Redesign the data pipeline“
- „Caching ergänzen“ (Redis vs. in-memory vs. file)
- „Echtzeit-Updates“ (WebSocket vs. SSE vs. Polling)

Schlechte Trigger:

- „Delete-Button am Profil“

- Tippfehler in der README
- Error-Handling an einer bekannten Stelle nachziehen
- „Können wir an Search arbeiten?“ (Startimpuls, kein Plan-Auftrag)

4.4 Der Approval-Flow: Preview, Kommentar, Freigabe

Wenn der Agent `exit_plan_mode` aufruft, liest das Tool `plan.md` und die TUI öffnet eine **scrollbare Preview** mit Action-Bar.

Shortcut	Aktion
`a`	**Approve** – Plan freigeben und bauen. Mit offenen Kommentaren: *approve w/ comments*
`s`	**Request changes** – Fokus auf Prompt, Revisionsnotizen tippen, `Enter` senden
`c`	**Comment** auf die selektierte Zeile/Range
`y`	**Copy** – gesamten Plan in die Zwischenablage
`q`	**Quit plan** – Plan verwerfen, Plan Mode aus
`Tab`	Fokus Preview ↔ Prompt
`Esc`	Vom Prompt zurück zur Preview (nicht automatisch quit)

Drei Fokus-Zustände:

1. **Preview** – scrollen, Zeilen für Kommentare wählen
2. **Commenting** – Inline-Kommentar zur Range (`c` oder `Enter` auf Zeile)
3. **Prompt** – freie Revisionsnotizen

Feedback (Inline, Freeform oder beides) hält Plan Mode **aktiv**: Der Agent überarbeitet den Plan, du iterierst, bis der Ansatz stimmt.

Leerer Plan: Fehlt `plan.md` oder ist sie leer, öffnet sich trotzdem die Approval-UI (Empty-State). Du kannst freigeben und bauen, Änderungen fordern oder quitten – in Minimal Mode landet der Empty-Hinweis im Scrollback.

Plan später wieder öffnen: `/view-plan` (Aliases `/show-plan`, `/plan-view`).

4.5 Always-approve und Compact unter Plan Mode

- **Always-approve bleibt „darunter“ scharf:** Nicht-Edit-Tools (Bash, Reads, MCP) können weiter auto-laufen. **Datei-Edits** am Repo bleiben gesperrt, bis du den Exit freigibst. Nach Approve gilt Always-approve wieder für die Implementierung.
- **Status-Flag:** während aktiv `plan`; Always-approve-Flag kommt nach Exit zurück, falls es an war.
- **`/compact`:** Plan-Mode-State bleibt erhalten; im kompaktierten Kontext steckt ein Reminder, dass Plan Mode aktiv ist.

Praxisregel: Plan Mode ist kein Ersatz für enge Shell-Deny-Regeln. Unter Always-approve in Plan Mode können riskante Commands weiterlaufen – nur Repo-Edit-Tools bleiben gesperrt.

4.6 Der Review-Vertrag (was du im Plan prüfen musst)

Bevor du `a` drückst, scanne bewusst:

Check	Frage
Scope	Welche Dateien – und was ist explizit out of scope?
Ansatz	Ein empfohlener Weg, nicht fünf unverbindliche Optionen?
Reuse	Nutzt er bestehende Utils/Patterns (mit Pfaden)?
Risiken	Migrationen, Breaking Changes, Feature-Flags?
Verify	Steht ein konkreter Test-/Manual-Check drin?
Größe	Passt das in **einen** PR – oder muss geschnitten werden?

Gute Kommentar-Beispiele (Inline oder Prompt):

```
Keine neue Redis-Abhängigkeit – nutze den bestehenden Cache-Client
↪  in pkg/cache.
Endpoint unter /v2/… wie die anderen Payments-Routen.
Verification: pytest tests/payments -q und ein manueller 401-Fall.
```

Schlechtes Approve: „Sieht smart aus“ ohne Dateiliste und Verify – das ist dasselbe wie blindes Diff-Accept in Normal Mode.

4.7 Praxis-Sessions

Session A – Feature mit Ambiguität (durchgängig)

```
cd ~/projects/your-app
grok
```

```
/plan
```

Nächster Prompt (oder direkt `/plan …`):

```
Wir brauchen rate limiting für die public API.
Constraints: kein neues Infra-Produkt, bestehende
↪  Middleware-Pipeline nutzen,
Konfiguration über Env, Tests grün.
Schreibe einen Implementierungsplan. Noch kein Code im Repo.
```

Ablauf erwarten: Explore → `plan.md` → Preview.
Deine Review-Aktion:

1. Scrollen, Dateipfade prüfen.
2. `c` auf eine Zeile: „In-memory reicht für single-instance; multi-instance später.“
3. Optional Freeform mit `s` / Prompt.
4. Wenn der revidierte Plan passt: `a`.
5. Implementierung beobachten; Diffs lesen; Verify-Abschnitt abarbeiten.

Session B – Plan ablehnen und neu schneiden

```
/plan Baue eine Event-Sourcing-Schicht für alle Domain-Events.
```

Im Preview merkst du: zu groß.

```
# s → Request changes, sinngemäß:
Zu groß für einen PR. Plane nur den Outbox-Pattern-Schritt
für den Order-Service. Kein Rewrite der History.
```

Oder `q` und neuer, engerer `/plan`-Prompt. **Trainingsziel:** Quit/Revise ist Erfolg, nicht Scheitern.

Session C – Wann du Plan Mode nicht brauchst

```
# Normal Mode, kein /plan
Im File src/utils/formatDate.ts: Timezone-Bug bei UTC+0.
Repro im Test ergänzen, minimal fixen, tests für formatDate grün.
```

Trainingsziel: Plan Mode ist ein Skalpell, kein Default für jeden Turn.

Session D – Agent will Plan Mode, du entscheidest

Manchmal schlägt der Agent `enter_plan_mode` vor.
Approve, wenn die Aufgabe architektonisch ambig ist.
Decline, wenn du schon weißt: „Nur den einen Guard anpassen“ – und den Scope im Prompt festnageln.

4.8 So baust du den Workflow im Team nach

1. **Definition of Ready:** Tickets mit Ambiguität markieren („Plan first“).
2. **Session-Start:** `/plan` + Constraints (Stack, Out-of-scope, Verify-Command).
3. **Review-SLA:** Tech-Lead oder Autor kommentiert Dateiliste und Verify, nicht Stil-Nitpicks.
4. **Approve nur mit Verify:** ohne Testplan kein `a`.
5. **Implementierung:** gleicher Session-Kontext oder frische Session mit „Implementiere den freigegebenen Plan: …“ + Link/Paste der Kernpunkte.
6. **PR-Beschreibung:** Plan-Ansatz in 5 Zeilen – Reviewer sieht die Entscheidung, nicht nur den Diff.

Optional: Plan-Inhalte, die das Team immer braucht, später in `AGENTS.md` oder Skills heben (Kapitel 7–8) – nicht den gesamten Plan committen, wenn er session-lokal bleiben soll.

4.9 Jetzt selbst machen: Plan-Parcours (25 Minuten)

☑ **JETZT SELBST MACHEN**

Standard: `beispiele/todo-cli` (Setup wie Kapitel 2). Eigenes ambiges Feature geht auch.

1. In todo-cli (oder bekanntem Repo) `grok` starten.
2. `/plan` und einen **ambigen** Auftrag mit 3 Constraints – todo-cli-Vorschlag:

```
/plan Atomare Writes in store.save: zuerst temp-Datei, dann
↪  replace.
Constraints: bestehende JSON-Form behalten; Tests grün (pytest
↪  -q);
kein neues Framework. Noch kein Repo-Code außer dem Plan.
```

3. Preview öffnen lassen; mindestens **einen** Inline-Kommentar (`c`) setzen.
4. Mit `s` eine Freeform-Revision schicken; warten auf Plan v2.
5. Plan mit `a` freigeben **oder** bewusst mit `q` verwerfen und enger neu starten.
6. Nach Approve: ersten Implementierungs-Diff lesen, bevor du weitermachst.
7. `/view-plan` einmal testen.
8. Zweite Mini-Session: klarer Bugfix **ohne** Plan Mode (z. B. Tippfehler in der README oder ein fehlender Test-Name).

4.10 Häufige Fehler

Symptom	Ursache	Fix
Agent „plant“ und editiert trotzdem Dateien	Subagent mit Write-Rechten / Shell-Write	Nur explore/plan; Parent-Plan ≠ Kind-Gate
Plan zu vage	Prompt ohne Constraints	Scope, Stack, Out-of-scope, Verify fordern
Endlos-Iteration	Kommentare ohne Priorität	Max. 3 Änderungswünsche pro Runde
Approve aus Höflichkeit	Review-Vertrag fehlt	Checkliste 4.6
Plan Mode für Tippfehler	Falsches Werkzeug	Normal Mode

Symptom	Ursache	Fix
Nach Compact „vergisst“ er den Mode	selten; State sollte bleiben	`/session-info`, Mode-Flag prüfen; ggf. `/plan` erneut

---

4.11 Checkliste Plan Mode

- ☐ Aufgabe hat echte Ambiguität oder hohen Blast Radius
- ☐ Constraints und Verify stehen im Prompt
- ☐ Preview gelesen (Dateien, Reuse, Risiken)
- ☐ Mindestens eine gezielte Korrektur, wenn etwas hakt
- ☐ Approve erst nach nachvollziehbarem Verify-Plan
- ☐ Subagents während Plan Mode nicht unkontrolliert schreiben lassen
- ☐ Always-approve-Risiken (Shell) bewusst

Was du jetzt kannst

- Plan Mode betreten, den Lifecycle skizzieren und die Edit-Sperre erklären
- Einen Plan in der Preview kommentieren, revidieren, freigeben oder verwerfen
- Entscheidungskriterien anwenden, wann Plan Mode hilft und wann er stört
- Einen durchgängigen Feature-Flow Plan → Approve → Implementierung fahren

Nächste Schritte

→ Kapitel 05 – Subagents und Parallelität: Kontext-Ventil, Agent-Typen, Personas und parallele Arbeit ohne den Parent zu überfüllen.

Kapitel 5

Subagents und Parallelität

> ⊙ **KERN**
>
> Subagents sind **Kind-Sessions mit eigenem Kontext**: Du delegierst Research, Implementierung oder Review parallel – der Parent bleibt schlank und bekommt Zusammenfassungen statt den ganzen Trace.

5.1 Das Problem, das Subagents lösen

Alltagsschmerz: Eine Session erforscht Auth, liest 40 Dateien, startet dann den Fix – und am Ende ist der Kontext voller toter Enden. `/compact` rettet Tokens, verliert aber Nuancen. Neue Session vergisst den Faden.

Subagents als Kontext-Ventil: Der Parent spawnt ein Kind, das *selbst* liest und sucht. Zurück kommt ein **Summary** (plus Artefakte/Pfade), nicht der komplette Scrollback des Kindes. Der Parent orchestriert; die Kinder schuften.

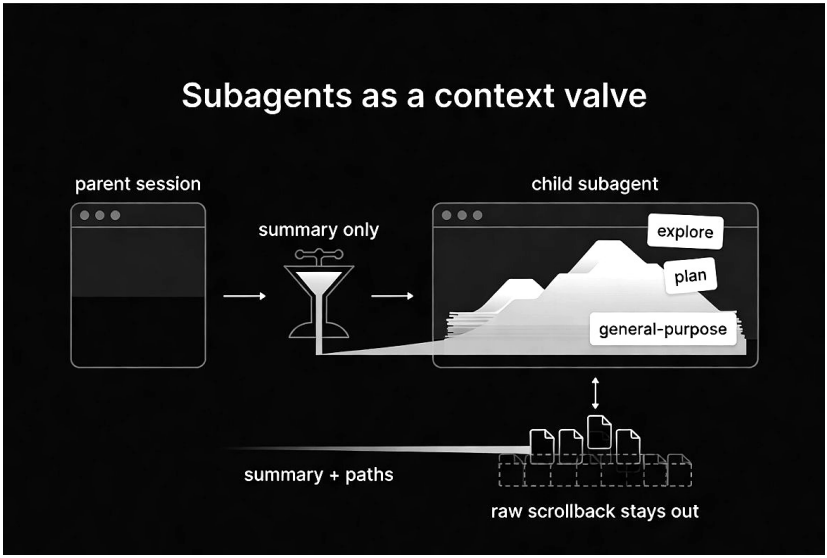


Abbildung 1: Der Roh-Leseberg bleibt draußen — nur die Zusammenfassung kehrt zurück.

Subagents sind **default an**. Abschalten:

```
export GROK_SUBAGENTS=0
```

```
# ~/.grok/config.toml
[subagents]
enabled = false
```

5.2 Agents vs. Personas (nicht verwechseln)

	Agents	Personas
Was	Ganze Session: Modell, Tools, Prompt, Skills	Verhaltens-Overlay auf Subagent-Prompt
Scope	Primary oder Subagent	**nur** Subagents
Wo	`.grok/agents/`, `~/.grok/agents/`, Modal	`config.toml [subagents.personas]`, `.grok/personas/*.toml`

	Agents	Personas
Beispiele	`grok-build`, `explore`, `plan`, Custom	`researcher`, `concise`

Merksatz: Ein Agent definiert *wer* die Session ist. Eine Persona formt *wie* ein Subagent innerhalb einer Session auftritt. Ein Kind läuft immer als Agent-Typ (z. B. `general-purpose`); die Resolution kann eine Persona **darüberlegen**. Verwalten: `/config-agents` (Alias `/agents`) – Tabs **Agents** und **Personas**. Personas-Tab direkt: `/personas`.

5.3 Built-in-Typen

`subagent_type`	Rolle	Datei-Edits
`general-purpose`	Default, volle Fähigkeiten	ja (voller Toolset)
`explore`	Research: lesen, greppen, Shell	**nein**
`plan`	Strukturierter Implementierungsplan	**nein**

Custom-Agents können Typen ergänzen oder Built-ins per Name überschatten.

Wann welcher Typ:

Aufgabe	Typ
„Wo wird Rate-Limit gesetzt? Nur finden.“	`explore`
„Skizziere einen Ansatz für Feature X.“	`plan` (oder Parent-Plan-Mode, Kapitel 4)
„Implementiere den freigegebenen Fix + Tests.“	`general-purpose`
Unklar / gemischt	`general-purpose` + enger Prompt, oder erst explore

Default: `general-purpose` nur, wenn Writes wirklich nötig sind – sonst `explore`/`plan` und Kontext sparen.

5.4 Wie Spawning funktioniert

Der Parent ruft intern `spawn_subagent` auf. Für dich als Nutzer zählt: Du **formulierst Delegation** klar – und reviewst das Summary.
Wichtige Parameter (Konzept – der Agent setzt sie; du steuerst über Prompt und Config):

Parameter	Bedeutung
`prompt`	Vollständiger Auftrag an das Kind
`description`	Kurzes Label (3–5 Wörter) für UI/Tasks
`subagent_type`	s. oben
`background`	`true`: sofort zurück mit ID; Ergebnis später holen
`capability_mode`	Grober Tool-Filter (s. 5.5)
`isolation`	`none` (Default, shared Workspace) oder `worktree` (Kapitel 6)
`cwd`	Anderes Working Directory (nicht mit `worktree` kombinieren)
`resume_from`	Abgeschlossenes Kind fortsetzen (gleicher Typ, gleiche Session)

Background-Kinder: Output/Status über `get_command_or_subagent_output`; abbrechen mit `kill_command_or_subagent`. Mehrere Tasks: warten mit dem Tool **`wait_commands_or_subagents`** und Parameter `mode` = `wait_any` oder `wait_all` (max. 20 Tasks). In der TUI: **Tasks-Pane** `Ctrl+G` – laufende/fertige Subagents und Background-Commands.
Nesting-Tiefe 1: Nur die Top-Level-Session spawnt Subagents. Ein Kind kann **keine** eigenen Subagents spawnen (`spawn_subagent` → Depth-Limit-Fehler). Der Agent-Baum bleibt flach – Orchestrierung bleibt beim Parent.

5.5 Capability Modes

Optionaler Filter **zusätzlich** zum Agent-Typ:

Mode	Read	Write	Shell/Execute
`read-only`	ja	nein	nein (auch kein Shell; Web/LSP-Lesen je nach Toolset)
`read-write`	ja	ja	nein
`execute`	ja	nein	ja
`all`	ja	ja	ja

Ohne `capability_mode` gilt das Toolset des Typs: `explore`/`plan` lesen+suchen+Shell, aber **keine** File-Edits; `general-purpose` = voll. **Praxis:** Für „nur recherchieren, Parent plant danach“ → `explore` oder `read-only`. Für „Tests laufen lassen, Code nicht anfassen“ → `execute`. Für parallele Feature-Arbeit → `general-purpose` + idealerweise `isolation: worktree` (Kapitel 6).

5.6 Personas in der Praxis

Persona = named instructions als `<system-reminder>`-Schicht: Ton, Output-Format, Fokus – **ohne** Agent-Typ/Tools zwingend zu ändern (Model/Effort können über Persona greifen).

```
# ~/.grok/config.toml oder Projekt-Config
[subagents.personas.researcher]
instructions = "Du bist gründlich. Nenne immer konkrete Dateipfade
↪  und Zitate (Funktion/Zeile)."
description = "Deep investigator."
```

Oder Datei `.grok/personas/researcher.toml` (Dateiname = Name). Priorität: Inline-Config > Projekt-Dateien > User-Dateien > Bundled (read-only). Nützliche Felder:

Feld	Nutzen
`instructions` / `instructions_file`	Verhalten
`description`	Katalog

Feld	Nutzen
`inputs` / `outputs`	IO-Contracts für Ketten (Datei rein/raus)
`model`, `reasoning_effort`	Overrides
`default_isolation`	`none` oder `worktree`

Resolution (Modell/Effort), grob höchste Priorität zuerst: Spawn-Override → Role-Default → Persona → Parent.

Isolation: ähnliche Kette, Default aber **`none`** (erbt nicht automatisch Worktree vom Parent).

Personas greifen über Resolution/Roles – der Parent übergibt typischerweise **keinen** Persona-Namen als freien `spawn_subagent`-Parameter. Model/Effort wechseln nur, wenn Persona oder Role sie setzen; sonst erbt das Kind den Parent. Du gestaltest Personas und Roles so, dass die Resolution den richtigen Overlay wählt; Details im Agents-Modal und in der Config (Kapitel 14).

5.7 resume_from und MCP-Vererbung

Multi-Stage:

1. Explore-Kind: Ursachen finden.
2. Neues Kind mit `resume_from=<id>`: implementiert auf Basis des **vollen** Kind-Transcripts (nicht nur Summary an Parent).

Voraussetzungen: Quelle **completed**, gleiche Session, **gleicher** `subagent_type`.

MCP: Kinder erben default die **bereits verbundenen** MCP-Server des Parents (`search_tool` / `use_tool` wie der Parent). Steuerung per Agent-Frontmatter `mcpInheritance`: `all` | `none` | `named: […]` | `except: […]`. Plugin-Agents: keine eigenen `mcpServers`/Hooks im Frontmatter; kein `permissionMode: bypassPermissions` im Plugin-Agent.

5.8 Parallelität ohne Chaos – Spielregeln

Regel	Warum
Klare **Nicht-Überlappung** der Dateien	Sonst Merge-Konflikt im Kopf und auf Disk
Research parallel, Writes serial oder in Worktrees	Shared Workspace + zwei Writer = Reibung
Jedes Kind bekommt **Definition of Done**	Sonst Summary ohne Beleg
Parent orchestriert Verify	„Beide sagen grün“ ≠ Integration grün
Tasks-Pane beobachten (`Ctrl+G`)	Du siehst hängende Kinder

Gute Parallel-Splits:

- Explore A: Call-Sites von `legacyPay()`
- Explore B: bestehende Retry-Utils
- Danach **ein** general-purpose: Fix

Schlechte Parallel-Splits:

- Zwei general-purpose schreiben beide an `auth/middleware.ts` im selben Tree

---

5.9 Config-Hebel (kurz)

Toggle, Models und Roles greifen in der **Resolution**, bevor der Parent spawnt – das spart Prompt-Yoga und erzwingt Team-Defaults (z. B. Research nur read-only).

```
[subagents.toggle]
explore = true
plan = false                    # plan-Subagent aus, wenn ihr nur
↪  Parent-Plan-Mode wollt

[subagents.models]
explore = "grok-build"        # Modell-Override pro Typ

[subagents.roles.researcher]
description = "Deep research agent"
```

```
default_capability_mode = "read-only"
model = "grok-build"
# prompt_file = ".grok/prompts/researcher.md"
```

Roles/Personas auch als Dateien unter `.grok/roles/*.toml` und `.grok/personas/*.toml`.

5.10 Praxis-Sessions

Session 1 – Explore als Ventil

```
Untersuche, wo Refresh-Tokens validiert werden.
Spawne dafür einen explore-Subagenten (read-only, keine Edits).
Liefere mir: Dateipfade, Sequenz, und 2 Risiken.
Parent: noch nichts implementieren.
```

Erfolg: Parent-Kontext bleibt klein; du hast Pfade und Risiken; kein Diff.

Session 2 – Plan-Subagent vs. Plan Mode

```
Spawne einen plan-Subagenten:
Implementierungsplan für CSV-Export der Invoices.
Constraints: bestehender Streaming-Helper, kein neues Framework.
Parent bleibt in Normal Mode und wartet auf den Plan-Text.
```

Vergleich zu Kapitel 4: Parent-**Plan Mode** hat Edit-Gate + Approval-UI. Der **plan-Subagent** liefert einen Plan ins Kind/Summary – ohne dass der Parent zwingend in Plan Mode ist. Für teamfähiges Approve-UI: Parent-Plan-Mode. Für schnelle Skizze parallel zu anderer Arbeit: plan-Subagent.

Session 3 – Implement + Test parallel (mit Vorsicht)

```
Zwei Tracks:
1) general-purpose: minimaler Fix für Bug B-123 in
↪  payments/retry.ts + Unit-Test.
2) explore (parallel): alle Call-Sites von retryPayment
↪  dokumentieren.
Keine weiteren Module anfassen. Am Ende Parent: Summary mergen und
↪  pytest tests/payments.
```

Wenn beide schreiben müssten: Track 1 im **Worktree** (Kapitel 6).

Session 4 – Background und Tasks-Pane

```
Starte die volle Testsuite im Hintergrund und
parallel einen explore-Subagenten zu flaky test_timeout.
Zeig Status in der Tasks-Pane; warte auf beide Results.
```

In der TUI: `Ctrl+G`, bei hängenden Jobs killen statt ewig warten.
Foreground-Shell → Background: `Ctrl+B`.

Session 5 – resume_from (Zwei-Stufen)

```
1) explore: root cause für Memory-Leak im Worker.
2) Danach general-purpose mit resume_from auf dieses Kind:
   minimaler Fix + Regressionstest. Kein Refactor außerhalb.
```

5.11 Prompt-Muster für Delegation

Schwach:

```
Mach das mit Subagents irgendwie parallel.
```

Stark:

```
Delegiere so:
- explore „find cache writers“: alle Writes auf Redis-Keys mit
↪  Prefix order:
  Liste path:line, kurze Notiz. Keine Edits.
- explore „find readers“: analog für Reads.
- Parent: nach beiden Summaries einen gemeinsamen Fix-Vorschlag in
↪  5 Bullets.
  Noch nicht implementieren.
```

Warum stark: typisiert, scoped, Output-Contract, Parent-Rolle klar.

5.12 Jetzt selbst machen: Subagent-Parcours (30 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. Subagents enabled lassen (`GROK_SUBAGENTS` nicht 0).
> 2. Session 1 (explore only) – Tasks-Pane einmal öffnen.
> 3. Session 2 (plan-Subagent) – Plan-Text reviewen, **nicht** blind bauen.
> 4. Session 3 – paralleles Explore + ein Writer; Dateiüberlappung vermeiden.
> 5. Optional: `/agents` öffnen und Built-ins ansehen.
> 6. Notiere: Wann würdest du Parent-Plan-Mode statt plan-Subagent wählen?

5.13 Häufige Fehler

Symptom	Ursache	Fix
Parent-Kontext trotzdem voll	Zu viel im Parent selbst gelesen	Explizit „nur Subagent exploriert“
Kind schreibt im Plan-Parent	general-purpose ohne Gate (Parent-Plan ≠ Kind-Gate, Kap. 4)	explore/plan oder read-only
Zwei Writer, Chaos-Diff	Shared isolation	Worktrees / serialisieren
Resume schlägt fehl	Anderer Typ / noch running	completed + same type
Persona „wirkt nicht“	Resolution/Name	Modal, Config-Priorität, instructions prüfen
MCP im Kind fehlt	nicht connected / mcpInheritance	Parent-Server zuerst; Inheritance

---

5.14 Checkliste Subagents

- ☐ Auftrag hat klaren Typ (explore / plan / general-purpose)
- ☐ DoD und Out-of-scope im Kind-Prompt
- ☐ Nesting-Tiefe 1: Orchestrierung nur im Parent
- ☐ Keine unnötige Dateiüberlappung bei Parallel-Writes

- ☐ Parent macht Integration-Verify
- ☐ Tasks-Pane im Blick bei Background
- ☐ Plan Mode Parent ≠ automatische Kind-Sperre verstanden

Was du jetzt kannst

- Subagents als Kontext-Ventil erklären und Agents von Personas trennen
- Built-in-Typen und Capability Modes gezielt wählen
- 2–4 parallele bzw. gestufte Sessions orchestrieren und im Tasks-Pane verfolgen
- Typische Parallelitäts-Fallen (Writes, Plan-Parent, Resume) vermeiden

Nächste Schritte

→ Kapitel 06 – Worktrees und isolierte Arbeit: git-isolierte Copies, damit parallele Agenten sich nicht gegenseitig die Working Tree zerschießen.

Kapitel 6

Worktrees und isolierte Arbeit

> ⊙ **KERN**
>
> Ein Git-Worktree ist eine **zweite Arbeitskopie desselben Repos**: Agenten (oder du) editieren isoliert – Merge/Apply erst, wenn der Diff reviewbar und grün ist.

6.1 Warum Isolation, wenn Subagents schon „eigenen Kontext“ haben?

Kapitel 5 trennt **Kontextfenster**. Worktrees trennen **Dateien auf Disk**.

Ohne Worktree (`isolation: none`)	Mit Worktree (`isolation: worktree`)
Alle schreiben in denselben Working Tree	Kind schreibt in eigene Worktree-Kopie
Parallel-Fixes reiben sich an denselben Dateien	Konflikte entstehen beim Merge, nicht mittendrin
`git status` mischt alle Experimente	Pro Experiment eigener Status

Alltagsschmerz: Session A refactort Auth, Session B fixt CI – beide rühren `package.json` an. Am Abend: weder Feature noch CI, dafür ein Diff, den niemand mergen will.

Philosophie: Isolation ist billiger als Heldentum. Lieber zwei saubere Worktrees und ein bewusstes Apply als einen gemeinsamen Working Tree als Chaos-Zone.



Abbildung 1: Kontextfenster trennen Subagents; Worktrees trennen Dateien auf Disk.

6.2 Drei Einstiege in Worktrees bei Grok Build

A) Neue interaktive Session im Worktree

```
# Wichtig: --worktree=name mit Gleichheitszeichen,
# sonst frisst -w den Prompt als Worktree-Label.
grok --worktree=feat "refactor module X"

# Kurzform -w, Basis-Ref setzen (z. B. main statt aktuellem HEAD):
grok -w --ref main "implement feature from main"
# bzw. --worktree-ref
```

Warum = bei `--worktree=feat`?

`grok -w „refactor module X“` interpretiert den String leicht als **Worktree-Name**, nicht als Prompt. Mit `--worktree=feat` und Prompt danach (oder expliziter Trennung) bleibt die Absicht klar.

B) Subagent mit `isolation: worktree`

Der Parent spawnt ein Kind; Grok legt einen isolierten Worktree an. Das Kind arbeitet dort; das Result enthält typischerweise den **Worktree-Pfad**. Änderungen bleiben isoliert, bis du sie zurückführst (Apply/Merge – intern u. a. über `x.ai/git/worktree/*`, inkl. Apply in den Haupt-Working-Tree).

```
# Prompt-Steuerung (sinngemäß – der Agent setzt isolation)
Implementiere Feature Flag "export_csv" in einem
↪  worktree-isolierten
general-purpose-Subagenten. Parent-Tree nicht anfassen.
Am Ende: Summary, Worktree-Pfad, Testbefehl, Diff-Statistik.
```

C) Session forken mit Worktree

```
/fork --worktree untersuche parallel den CI-Fix auf main-Stand
/fork --no-worktree bleib im selben Tree, nur Gespräch verzweigen
/fork   # ohne Flag: Grok fragt nach
```

Resume in frischem Worktree:

```
grok -w -r <session-id>
```

Neue Session-Taste: `Ctrl+N` (ggf. Doppel-Geste) kann optional einen Worktree anbieten – UI-abhängig; CLI-Flags sind die reproduzierbare Variante.

6.3 Was Grok intern orchestriert

Worktree-Sessions nutzt Extension-Methoden unter `x.ai/git/worktree/*`, sinngemäß:

Operation	Bedeutung
Create	Neuer Worktree für isolierte Session/Kind
Apply	Änderungen zurück in den Haupt-Working-Tree mergen/anwenden
Remove / gc	Aufräumen, wenn fertig
list / status	Übersicht und Fortschritt

Du musst die JSON-RPC-Namen nicht auswendig können – aber du musst **Apply bewusst** behandeln: Isolation ohne Merge-Plan ist nur aufgeschobenes Chaos.

`grok sessions list` gruppiert Sessions oft nach **Worktree-Label** – hilfreich, wenn du den Überblick verlierst.

6.4 Git-Grundlagen, die du brauchst (90 Sekunden)

```
# klassisches manuelles Worktree (ohne Grok) – Konzept
git worktree list
git worktree add ../your-project-feat-x -b feat/x main
cd ../your-project-feat-x
# … arbeiten …
cd ../your-project
git worktree remove ../your-project-feat-x
```

Grok automatisiert Create/Apply/Remove in seinem Workflow. **Dein Job:**

1. Wissen, **in welchem Tree** du gerade bist (`pwd`, `git rev-parse --show-toplevel`).
2. Diffs **pro Tree** reviewen.
3. Apply/Merge nur mit grünen Checks.
4. Keine Secrets und keine `.env` mit echten Keys in Worktrees committen.

> ⚠ **ACHTUNG**
>
> Untracked Secrets in einem Tree sind nicht „sicherer“, nur weil ein zweiter Tree existiert. Worktrees teilen oft das Objekt-Repo; Credentials gehören in Env/Vault, nicht in Dateien.

6.5 Wann Worktree – wann nicht?

Situation	Worktree?
Zwei Features parallel mit Datei-Overlap-Risiko	**ja**
Riskanter Refactor, Baseline soll clean bleiben	**ja**

Situation	Worktree?
Agent-Experiment „was-if“, das du verwerfen willst	**ja**
Einzeiler-Bugfix, eine Datei	nein – Overhead
Reines Lesen / Explore	nein – `explore` reicht
Schon schmutziger Working Tree voller WIP	**ja** (oder erst stash/commit)

Persona-Default: `default_isolation = „worktree“` an Personas/Roles, die immer schreiben – dann startet Resolution eher isoliert (Kapitel 5). Default ohne Override bleibt meist `none`.

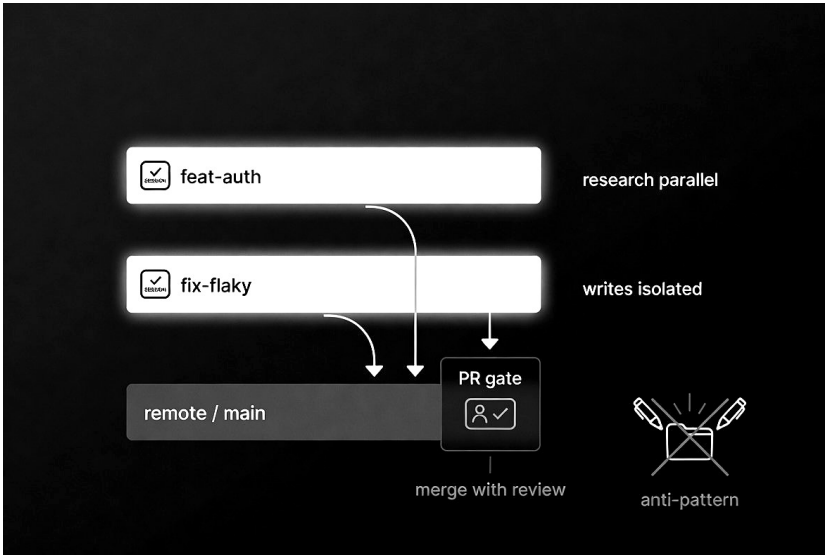


Abbildung 2: Parallel arbeiten ohne sich ins Gehege zu kommen.

6.6 Der Apply-/Merge-Vertrag

Nach isolierter Arbeit:

1. **Summary lesen** – was wurde geändert, welche Tests?
2. **Worktree-Pfad öffnen** – `git status`, `git diff`, kritische Files.
3. **Verify im Worktree** – dieselben Commands wie im Plan.

4. **Apply/Merge** in den Haupt-Tree (Agent-Tooling oder manuell per Branch/PR).
5. **Integrationstest im Haupt-Tree** – „im Worktree grün“ ≠ „mit deinem WIP grün“.
6. **Worktree entfernen**, wenn der Branch im Remote/PR lebt.

PR-first Variante (oft teamtauglicher): Worktree pusht Branch → normaler PR → kein Apply in den schmutzigen lokalen main. Isolation bleibt; Review läuft über GitHub/GitLab.

```
Arbeite im Worktree-Branch feat/export-csv.
Am Ende: Branch pushen, PR-Body skizzieren, Haupt-Working-Tree
↪  unangetastet.
```

6.7 Zusammenspiel mit Plan Mode und Subagents

```
Parent: Plan Mode (Kapitel 4) → Ansatz freigeben
   └─ general-purpose Subagent, isolation=worktree →
↪  implementieren
Parent: Summary + Verify-Plan prüfen → Apply oder PR
```

Fallen:

- Parent in Plan Mode, Kind `general-purpose` **ohne** Worktree im shared Tree → Kind kann trotz Parent-Plan **schreiben** (Parent-Plan ≠ Kind-Gate, Kapitel 4).
- Zwei Worktree-Kinder am selben logischen Feature ohne Absprachen → doppelte Branches, doppelte Wahrheit.
- Apply während Parent selbst dirty ist → Konflikte; vorher Parent clean oder PR-only.

6.8 Praxis-Sessions

Session A – Feature im Launch-Worktree

```
cd ~/projects/your-app
git status   # ideal: clean oder bewusstes WIP
```

```
grok --worktree=export-csv "Add CSV export for invoices using
↪  existing streaming helpers. Tests required. Do not touch
↪  auth."
```

Trainingsziel: `pwd` / Worktree-Label in Session-Info; Diff nur in diesem Tree; Haupt-Tree unverändert (`git status` im Original-Pfad).

Session B – Parent orchestriert isoliertes Kind

```
grok

Bleib im aktuellen Tree und ändere nichts selbst.
Spawne einen general-purpose-Subagenten mit isolation=worktree:
Fix flaky test_retry_timeout, minimale Änderung, pytest
↪  tests/payments -q.
Report: Worktree-Pfad, Diff-Stat, Testergebnis.
Apply noch nicht — ich review zuerst.
```

Trainingsziel: Review vor Apply; du entscheidest Merge.

Session C – Zwei parallele Tracks

```
Track 1 (worktree): Feature-Flag docs-only in README + config
↪  sample.
Track 2 (worktree): Unit-Test für formatDate Timezone-Bug.
Kein gemeinsamer Dateipfad. Am Ende zwei Summaries, kein
↪  Auto-Apply.
```

Trainingsziel: Parallelität mit **disjunkten** Scopes.

Session D – Fork mit Worktree

```
/fork --worktree alternative: implement caching with in-memory map
↪  only
```

Original-Session bleibt auf dem anderen Ansatz. Später einen Weg verwerfen.

Session E – Resume isoliert

```
grok -w -r <session-id>
```

Trainingsziel: Alte Session, frische Disk-Isolation – nützlich nach Laptop-Wechsel oder wenn der alte Worktree weg ist.

6.9 Manuelles Sicherheitsnetz

Wenn Grok-Worktrees und dein Git durcheinanderwirken:

```
git worktree list
git status
git branch -vv
# vorsichtig aufräumen – nur Worktrees, die du verstehst:
# git worktree remove <path>
```

Nicht blind `rm -rf` auf Worktree-Pfade mit uncommitteter Arbeit. **Nicht** force-pushen, „um den Tree sauber zu machen“.

6.10 Jetzt selbst machen: Isolation-Parcours (30 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. Repo mit sauberem `git status` wählen (oder WIP committen/stashen).
> 2. Session A: `grok --worktree=…` mit engem Feature-Prompt.
> 3. Im **Original**-Pfad `git status` – sollte das Experiment nicht zeigen.
> 4. Im Worktree Diff + Test laufen lassen.
> 5. Session B: Parent spawnt worktree-Subagent, **kein** sofortiges Apply.
> 6. Bewusst entscheiden: PR-Branch **oder** Apply – und den anderen Weg einmal notieren.
> 7. `git worktree list` / `grok sessions list` ansehen.
> 8. Aufräumen: Worktree remove oder Branch behalten, aber dokumentieren.

6.11 Häufige Fehler

Symptom	Ursache	Fix
Prompt wurde Worktree-Name	`-w „langer prompt“`	`--worktree=label` + Prompt getrennt
„Isolation“, aber Dateien im Haupt-Tree	`isolation: none` / Agent ignoriert	Prompt erzwingen; Diff-Pfade prüfen
Apply-Konflikt-Hölle	Dirty Parent + überlappende Edits	Clean Parent oder PR-only
Verwaiste Worktrees	Session tot, Tree bleibt	`git worktree list` + remove
Doppelte Features	Zwei Isolationen ohne Owner	Ein Track pro Feature-Owner
Falscher Ref	Worktree von dirty HEAD	`--ref main` / stabile Basis

6.12 Checkliste Worktrees

- ☐ Parent-Tree clean **oder** bewusst PR-only (kein Apply in dirty Parent)
- ☐ Basis-Ref bewusst (main vs. feature HEAD)
- ☐ Label/Name verständlich (`feat-…`, nicht `tmp1`)
- ☐ Scope disjunkt bei Parallel-Tracks
- ☐ Verify **im** Worktree vor Apply/PR
- ☐ Integration nochmal im Ziel-Tree
- ☐ Aufräumen terminiert (remove oder Branch-Lifecycle)
- ☐ Keine Secrets in Trees

Was du jetzt kannst

- Worktrees als Disk-Isolation von Kontext-Isolation (Subagents) unterscheiden
- Sessions und Subagents isoliert starten (`--worktree=`, `isolation`, `/fork`)
- Einen Apply/PR-Vertrag fahren und parallele Tracks ohne Tree-Kollision orchestrieren
- Häufige CLI- und Git-Fallen vermeiden

Nächste Schritte

→ Kapitel 07 – AGENTS.md und Projektregeln: was in **jeder** Session (und jedem Worktree) stabil gelten soll – ohne es in jeden Prompt zu kopieren.

Kapitel 7

AGENTS.md und Projektregeln

> ⊙ **KERN**
>
> `AGENTS.md` und Rules-Dateien sind der **stabile Kompass** deines Repos: einmal geschrieben, in jeder Session geladen – präzise genug zum Handeln, kurz genug zum Folgen.

7.1 Warum Dateien statt Prompt-Rituale

Alltagsschmerz: Jede Session beginnt mit „Bitte TypeScript, bitte `npm test`, bitte keine Secrets committen…“. Drei Leute, drei leicht abweichende Versionen. Der Agent hält sich an den lautesten Prompt – nicht an den Team-Standard.

Lösung: Handlungsanweisungen **versionieren**. Was in *jeder* Session gelten soll, gehört in Projektregeln. Was nur *diese* Story betrifft, bleibt im Prompt oder Ticket.

Schicht	Lebensdauer	Beispiel
Prompt	ein Turn / eine Story	„Nur `payments/retry.ts` anfassen“
`AGENTS.md` / Rules	Repo-Lebenszeit	„Tests: `npm test`. Conventional Commits.“
Skill	wiederholbarer Workflow	`/commit`, `/verify` (Kapitel 8)
Hook / Permission	erzwungen	`rm -rf` blocken (Kapitel 8, 13)

Merksatz: AGENTS.md ist **weich** (Anweisung). Permissions/Hooks sind **hart** (Durchsetzung). Beides brauchen gute Teams – ersetzt einander nicht.

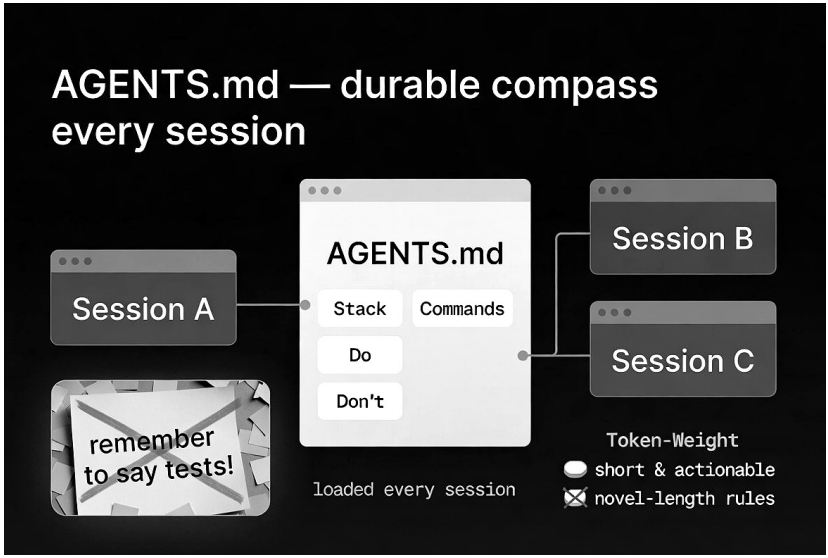


Abbildung 1: Was du jedes Mal sagen müsstest, gehört in AGENTS.md — kurz und konkret.

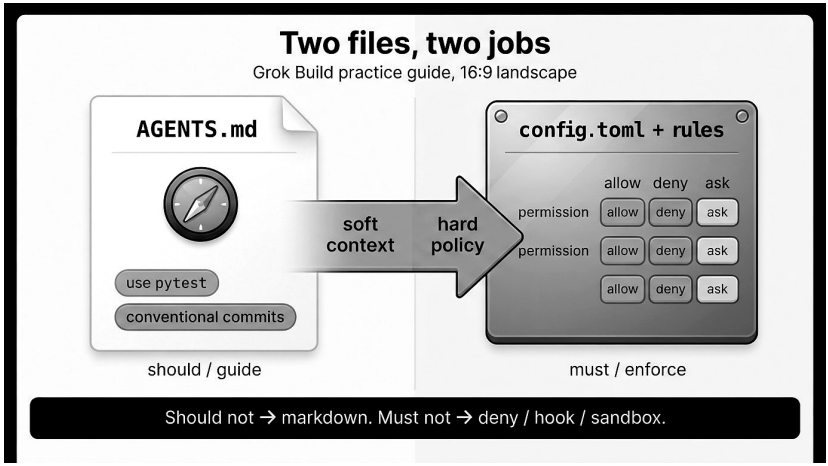


Abbildung 2: Sollte nicht → Markdown. Darf nicht → deny, Hook oder Sandbox.

7.2 Welche Dateien Grok lädt

Pro Verzeichnis prüft Grok u. a. (Reihenfolge der Namensliste laut User-Guide):

- `Agents.md`, `Claude.md`, `CLAUDE.md`, `CLAUDE.local.md`, `AGENT.md`, `AGENTS.md`

Mehrere Treffer im **selben** Ordner werden alle geladen. Auf case-insensitiven Dateisystemen dedupliziert Grok denselben Dateiinhalt (z. B. `Agents.md` ≡ `AGENTS.md`).

Claude-Kompatibilität (default an): zusätzlich Home `~/.claude/` und pro Level `.claude/CLAUDE.md` / `.claude/CLAUDE.local.md`.

Cursor-Kompatibilität: analog `~/.cursor/` und projektbezogene Cursor-Rules, wenn enabled.

Rules-Verzeichnisse

Zusätzlich zu den Named Files: alle `*.md` in:

Ort	Scope
`<dir>/.grok/rules/`	immer
`<dir>/.claude/rules/`	Claude-compat (konfigurierbar)
`<dir>/.cursor/rules/`	Cursor-compat (konfigurierbar)
`~/.grok/rules/`	global (alle Projekte)
`~/.claude/rules/`, `~/.cursor/rules/`	global, vendor-gated

Empfehlung für Grok-first Teams: Root-`AGENTS.md` + bei Bedarf `.grok/rules/*.md` für thematische Scheiben (`testing.md`, `security.md`).

7.3 Discovery und Precedence

Reihenfolge:

1. **Home-Rules** (`$GROK_HOME` / `~/.grok/rules/`, dann optionale Vendor-Homes)
2. **Repo-Pfad** vom Git-Root bis CWD (inklusive) – jedes Level Named Files + rules/

3. **Kein Git-Repo:** nur CWD

```
my-app/
  AGENTS.md                 # global im Repo
  src/
    AGENTS.md               # enger
    components/
      AGENTS.md             # am engsten
```

In `src/components/` lädt Grok **alle drei**. **Deeper wins:** bei Konflikt gilt die Anweisung, die **später** im Kontext erscheint (tieferes Verzeichnis).

Auto-Loading: Root→CWD beim Session-Start. Arbeitet der Agent in Verzeichnissen außerhalb dieses Pfads, erkennt er dortige Instruction-Files und zieht sie bei Bedarf nach.

Gitignore: Von `.gitignore` ignorierte Instruction-Files werden bei der Discovery **übersprungen**. Persönliche Overrides z. B. als `CLAUDE.local.md` gitignoren – Custom-Namen wie `AGENTS.local.md` werden als Top-Level-Named-File **nicht** erkannt (in `.grok/rules/` gilt jedes `*.md`).

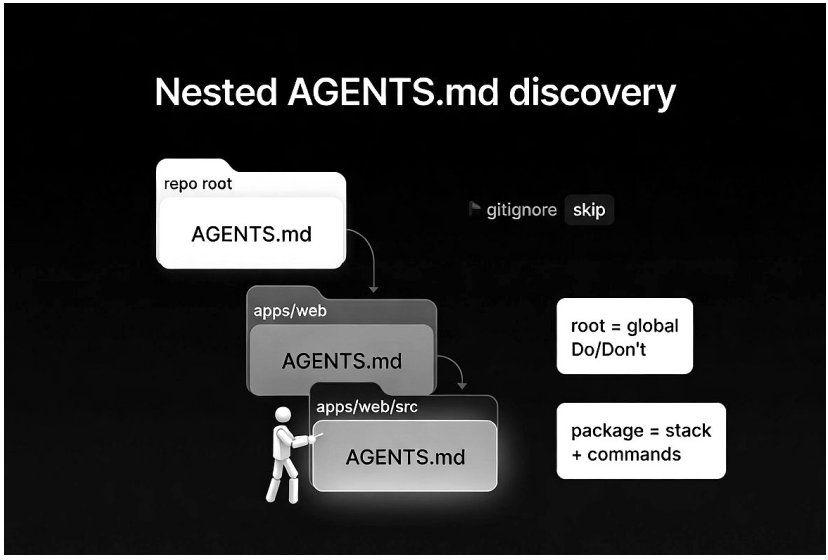


Abbildung 3: Vom Repo-Root zum Package — engere Regeln gewinnen, wo der Agent arbeitet.

7.4 Was rein darf – und was nicht

Rein (actionable)

```
# AGENTS.md

## Stack
- TypeScript strict; keine neuen `any` ohne Begründung im PR

## Commands
- Test: `npm test`
- Lint: `npm run lint`
- Nie `npm install` ohne Nachfrage

## Do
- Vor Merge: Tests + Lint grün
- Diffs klein und reviewbar halten

## Don't
- Keine Secrets in Repo oder Prompt
- Kein Force-Push auf `main`
- Keine Edits an `dist/` generierten Artefakten
```

Commands an euren Stack anpassen (`pytest`/`cargo`/…); siehe auch `beispiele/todo-cli/AGENTS.md` und die Vorlagen unter `vorlagen/`.

Raus (oder nur verlinken)

- README-Kopie, Architektur-Roman, Marketing
- Secrets, Tokens, interne Hostnamen mit Credentials
- 20 Seiten Style-Guide – lieber Link + 5 harte Regeln

Dateigröße: Grok lädt Instruction-Files **vollständig** (kein Cap/Truncate). Trotzdem: kurz und spezifisch – Kontext ist endlich, und lange Wände werden schlechter befolgt.

7.5 Nested Rules und Monorepos

```
my-monorepo/
  AGENTS.md                    # Workspace: pnpm,
↪  Package-Grenzen
  packages/
    frontend/
      AGENTS.md                # React, CSS modules
      .grok/rules/a11y.md
    backend/
      AGENTS.md                # Express, REST, OpenAPI
    shared/
      AGENTS.md                # kein Framework-Code
```

Pattern: Root = Build-Orchestrierung und Do/Don't global. Package = Stack und Commands. `.grok/rules/` = thematisch (Testing, Security), damit Root-AGENTS.md nicht explodiert.

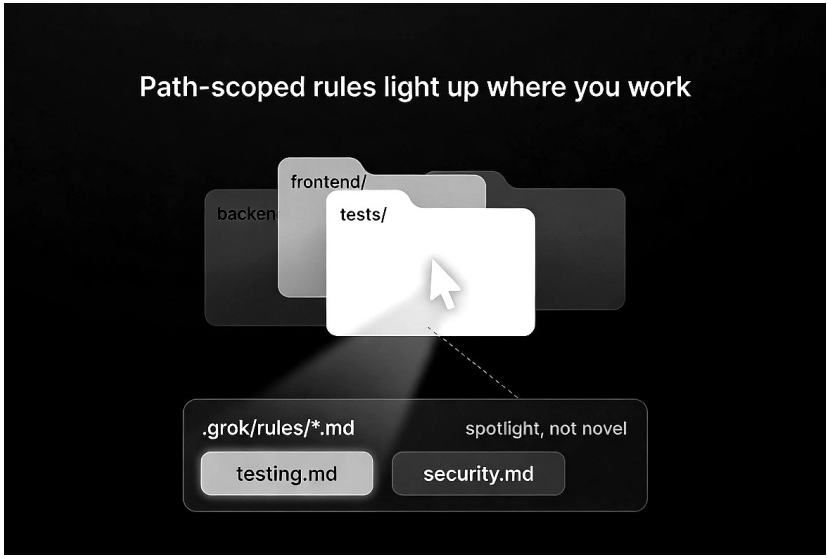


Abbildung 4: Regeln leuchten dort, wo der Agent arbeitet.

CWD-Trick: Starte `grok` im Package, das du bearbeitest (`cd packages/backend && grok`), damit die relevanten Nested Rules ohne Umweg greifen – und der Agent weniger Frontend-Kontext mitschleppt.

7.6 Session-Flags und Overrides

```
# Zusätzliche Regeln nur für diese Session (an System-Prompt
↪ angehängt)
grok --rules "Always use TypeScript. Prefer functional
↪ components."
# Alias: --append-system-prompt

# Kompletter System-Prompt-Ersatz (Default-Prompt + --rules
↪ entfallen)
grok --system-prompt-override "Du bist ein strenger
↪ Security-Reviewer. …"
# Alias: --system-prompt
```

Wann `--rules`: Experiment, Pairing, einmaliger Policy-Spike.
Wann Datei: Team-Standard.
Wann Override: Spezial-Agent (selten) – du verlierst den Default-Harness-Prompt, also bewusst einsetzen.

7.7 Inspizieren: was wurde wirklich geladen?

```
grok inspect
grok inspect --json
```

Zeigt Instruction-Files mit Pfad und grober Token-Last – plus Skills, MCP, etc.
Erster Debug-Schritt, wenn „er hält sich nicht an unsere Regeln“: Liegt die Datei im Discovery-Pfad? Gitignored? Falsches CWD?
In der Session helfen `/context` und `/session-info` bei der Kontext-Bilanz (Kapitel 3, 12).

7.8 Kompatibilität mit Claude Code und Cursor

Viele Repos haben schon `CLAUDE.md` und `.claude/rules/`. Grok liest sie standardmäßig mit – du musst nicht sofort umbenennen.

Ziel	Praxis
Grok-first, grünes Feld	`AGENTS.md` + `.grok/rules/`
Claude-Team, Grok-Einstieg	`CLAUDE.md` behalten; optional `AGENTS.md` mit denselben Kernregeln
Dual-Agent	Beide Dateien **kurz und konsistent** – nicht zwei Wahrheiten
Vendor aus	`[compat.claude]` / `[compat.cursor]` Rules/Agents/Skills in Config (Kapitel 14)

Anti-Pattern: `AGENTS.md` sagt „pnpm“, `CLAUDE.md` sagt „npm“ – die **später geladene** Anweisung gewinnt (Precedence ist bestimmt). Im Alltag wirkt das wie Willkür, wenn beide Dateien divergieren. Eine Source of Truth, der Rest verlinkt oder ist identisch.

7.9 Best Practices (knapp und verbindlich)

1. **Root zuerst** – das Wichtigste project-weit.
2. **Spezifisch** – `` `cargo fmt` vor Commit `` schlägt „formatiere bitte“.
3. **Kurz** – eine Seite schlägt ein Essay.
4. **Nested für Monorepos** – nicht alles in Root pressen.
5. **Versionieren** – committe Team-Regeln; Persönliches nach `~/.grok/` oder `*.local.md` + gitignore.
6. **Nicht duplizieren** – README verlinken, nicht kopieren.
7. **Periodisch reviewen** – tote Commands sind schlimmer als keine.
8. **Prüfbar** – jede Do-Regel idealerweise mit Command oder „Definition of Done“.

Minimal-Vorlage: `vorlagen/AGENTS.md.minimal.md`.

7.10 Praxis-Sessions

Session 1 – Minimal-AGENTS.md anlegen

```
cd ~/projects/your-app
# Datei anlegen (oder vom Agenten schreiben lassen – dann Diff
↪  lesen)

Erstelle eine knappe Root-AGENTS.md mit: Stack in 3 Bullets,
Test- und Lint-Commands exakt, 3 Don'ts (Secrets, force-push,
↪  dist/).
Keine Romane. Zeig mir den Diff.
```

Danach:

```
grok inspect | head -80
```

Erfolg: Datei gelistet, Token-Count sichtbar.

Session 2 – Nested Rule greift

```
# packages/backend/AGENTS.md (oder lass den Agenten anlegen)
Nur im Backend: REST, keine React-Imports, Tests mit pytest.

cd packages/backend && grok

Welche Projektregeln gelten hier? Liste die geladenen
↪  Instruction-Pfade
und fasse die Backend-spezifischen Don'ts zusammen. Ändere nichts.
```

Erfolg: Nested Inhalt taucht auf; Root bleibt sichtbar.

Session 3 – Konflikt Deep vs. Root

Root: „Formatter: prettier“.
`src/legacy/AGENTS.md`: „Dieser Ordner: kein Auto-Format, manuell.“

```
Arbeite nur unter src/legacy/. Welcher Formatter-Hinweis gewinnt
↪  und warum?
```

Trainingsziel: Deeper precedence erklären können.

Session 4 – Session-Rules ohne Datei

```
grok --rules "Antworte auf Deutsch. Keine neuen Dependencies."

Schlage einen Fix für den flaky Test vor – halte dich an die
↪  Session-Rules.
```

Trainingsziel: `--rules` vs. committed AGENTS.md unterscheiden.

7.11 Das `.grok/`-Projektverzeichnis (Landkarte)

Pfad	Zweck
`.grok/config.toml`	u. a. MCP, Plugins, Permissions (projektskopiert)
`.grok/skills/`	Team-Skills (Kapitel 8)
`.grok/plugins/`	Projekt-Plugins
`.grok/agents/`	Custom Agents
`.grok/hooks/`	Lifecycle-Hooks (Kapitel 8)
`.grok/rules/`	Nested Markdown-Rules
`.grok/lsp.json`	LSP

Alles optional. **AGENTS.md bleibt der Einstieg** – der Rest skaliert das Harness.

7.12 Jetzt selbst machen: Rules-Parcours (20 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. Root-`AGENTS.md` mit Commands + Don'ts anlegen oder schärfen.
> 2. `grok inspect` – Datei und Tokens notieren.
> 3. Eine `.grok/rules/testing.md` mit einem Satz „Neue Features: failing-first Test".
> 4. Session starten und den Agenten die geladenen Rules nennen lassen.
> 5. Optional: absichtlich gitignorierte `CLAUDE.local.md` mit privatem Hinweis – prüfen, ob sie (nicht) erscheint.

6. Monorepo: CWD in ein Package legen und Nested-Verhalten spüren.

7.13 Häufige Fehler

Symptom	Ursache	Fix
Regeln greifen nicht	Falsches CWD / gitignore / Name	`grok inspect`, Pfade prüfen
Zwei widersprüchliche Dateien	AGENTS + CLAUDE divergieren	Eine Wahrheit
Agent ignoriert lange Bibel	Zu viel Text	Kürzen, Skills für Workflows
Secrets in AGENTS.md	Copy-Paste	Sofort rotieren; nur Token-Dummies in Beispielen
„Lokal nur bei mir“ committed	Personal in Root	`~/.grok/` oder local+gitignore

---

7.14 Checkliste AGENTS.md

- ☐ Root-Datei mit Stack, Commands, Do/Don’t
- ☐ Commands exakt und lauffähig
- ☐ Nested nur wo Konventionen abweichen
- ☐ `grok inspect` einmal verifiziert
- ☐ Team-Datei committed; Locals gitignored
- ☐ Keine Secrets
- ☐ Review-Rhythmus (Quartal / nach großen Stack-Wechseln)

Was du jetzt kannst

- Discovery, Precedence und Named-File-Kompatibilität erklären
- Eine knappe, prüfbare Root-`AGENTS.md` und Nested Rules bauen
- `grok inspect` und Session-Flags gezielt einsetzen
- Dual-Agent-Repos ohne widersprüchliche Kompasse fahren

Nächste Schritte

→ Kapitel 08 – Skills, Plugins, Hooks und Marketplaces: wiederholbare Workflows on-demand, erzwungene Gates und installierbare Bundles.

Kapitel 8

Skills, Plugins, Hooks und Marketplaces

> ⊙ **KERN**
>
> Skills sind **wiederholbare Verfahren on-demand**, Hooks **erzwingen** Lebenszyklus-Gates, Plugins/Marketplaces **packen und verteilen** beides – und du wählst bewusst weich vs. hart.

8.1 Steuerflächen im Vergleich

Alltagsschmerz: Der Agent „vergisst" das Verify-Ritual, der Hook blockt still einen `rm`, und niemand weiß, ob das Team-Plugin überhaupt trusted ist. Die Tabelle sortiert **weich → hart → verpackt**.

Steuerfläche	Charakter	Typisches Beispiel
AGENTS.md	immer im Kontext, weich	„Tests: `npm test`"
Skill	bei Bedarf / per `/name`, Verfahren	`/commit`, `/verify`
Hook	Event-getrieben, kann **blocken**	PreToolUse: `rm -rf` deny
Plugin	Bundle aus Skills, Hooks, Agents, MCP	Team-„deploy-tools"
Marketplace	Katalog, aus dem du Plugins installierst	org/team-plugins

Wann Skill statt AGENTS.md? Zu speziell oder zu lang für jede Session, aber zu häufig zum Tippen.
Wann Hook statt Skill? „Darf gar nicht passieren" oder „nach jedem Edit formatieren" – ohne dass der Agent es „vergessen" darf.
Wann Plugin? Mehrere Artefakte + Team-Distribution.

Weich vs. hart (K-05) und Skills vs. Hooks (M-07) hängen zusammen: erst die Härte-Skala, dann die Verpackung.

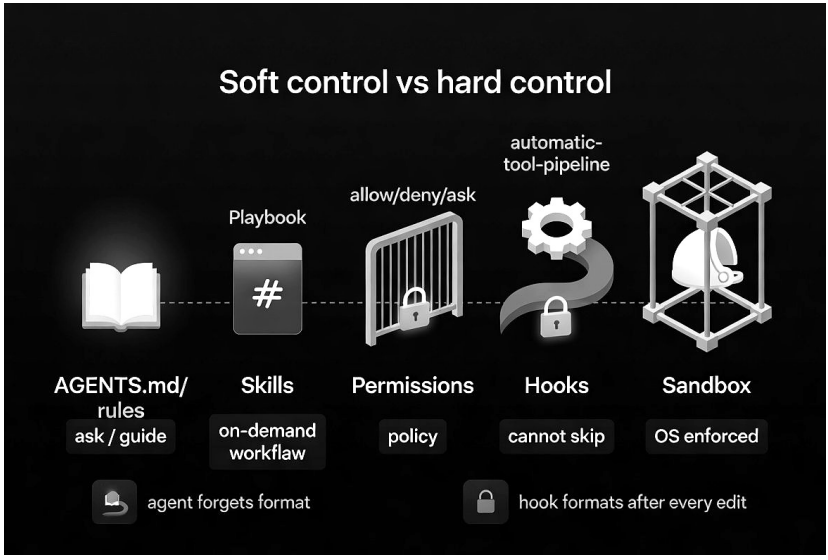


Abbildung 1: Je härter die Anforderung, desto weiter rechts: AGENTS.md bittet, Sandbox erzwingt.

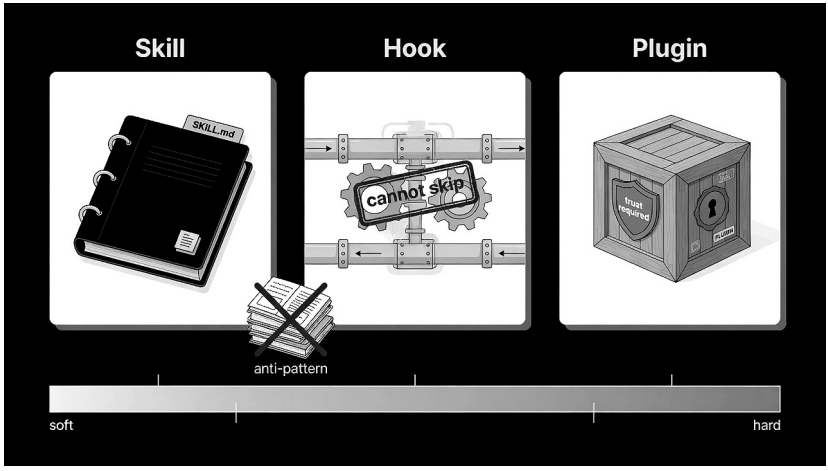


Abbildung 2: Prozeduren → Skills. Unverhandelbares → Hooks. Pakete → Plugins mit Trust.

8.2 Skills: Aufbau und Orte

Ein Skill = Verzeichnis mit `SKILL.md` (YAML-Frontmatter + Markdown-Body).

Discovery (Priorität, höher gewinnt bei gleichem Namen):

Ort	Scope
`./.grok/skills/`, `./.grok/commands/`	CWD/local (höchste)
`<repo>/.grok/skills/`, `…/commands/`	Repo
`~/.grok/skills/`, `~/.grok/commands/`	User
Claude/Cursor-Pfade	Compat (konfigurierbar)
Bundled / Plugins	Plattform bzw. installiert

Zusätzlich: `.agents/skills/` pro Tier; Walk von CWD bis Repo-Root. Flat `commands/*.md` → Slash-Commands (Stem = Name), Claude-Legacy-Layout.

Wichtig: Skill-Discovery **ignoriert `.gitignore` nicht im Sinne von „ausblenden“** – Pfade unter bekannten Roots (`.grok/`, `.agents/`, `.claude/`, `.cursor/`) laden, wenn sie auf Disk liegen. Verstecken über Config:

```
[skills]
paths = ["~/my-team-skills"]
ignore = ["~/my-team-skills/wip"]
disabled = ["wip-skill"]
```

Minimal-SKILL.md

```
---
name: verify
description: >
  Run project tests and lint after code changes. Use when the user
↪  asks
  to verify, /verify, or before committing.
---

# Verify

1. Run the project's test command from AGENTS.md (fallback: `npm
↪  test`).
2. Run lint if defined.
```

```
3. Report pass/fail with the first failure excerpt.
4. Do not push or open PRs.
```

Frontmatter	Rolle
`name`	Identifier (lowercase, digits, hyphens; max 64)
`description`	**Trigger** für Auto-Invocation – spezifisch schreiben
`when-to-use`	extra Trigger-Phrasen
`user-invocable`	default `true` → erscheint als `/name`
`disable-model-invocation`	`true` = nur Slash, kein Auto
`allowed-tools`, `model`, `effort`, …	optional eng führen

Wie Trigger greifen: `description` und `when-to-use` steuern, wann der Agent den Skill **von allein** lädt. Vage Descriptions („hilft bei Code“) nerven: zu oft oder nie. Schärfe sie so, dass der Use-Case und die Slash-Form (`/verify`) erkennbar sind (Vertiefung: 8.3).

8.3 Skills nutzen und erzeugen

Aufruf

```
/verify
/commit fix the build
/local:commit          # CWD: ./.grok/skills/
/repo:commit           # Repo-Root: <repo>/.grok/skills/
/user:commit           # User: ~/.grok/skills/
/deploy-tools:release  # Plugin-Skill: /<plugin-name>:<skill>
```

`/` öffnet das Menü (Builtins + Skills). Builtins gewinnen unqualifiziert über gleichnamige Skills – dann den Scope-Prefix nutzen. CLI-Übersicht: `grok inspect`.

Erzeugen mit `/create-skill`

Interaktiver Flow: Name, Scope (Project vs. User), Description-Draft, Verzeichnis, `SKILL.md`, Verify.

- **Project:** `<repo>/.grok/skills/<name>/` – teamfähig, committen. - **User:** `~/.grok/skills/<name>/` – persönlich.

Team-weit: lieber **Plugin + Marketplace** (unten) als 30 manuelle Kopien.

Reload: Dateiänderung auf Disk; Skills erscheinen meist in Sekunden im Slash-Menü. Neu installierte Plugin-Skills: `r` im Plugins-Tab oder neue Session.

Auto-Invocation: Grok matcht Prompt gegen `description` / `when-to-use`. Vage Descriptions = ungewollte Triggers. Für manuell-only: `disable-model-invocation: true`.

8.4 Plugins und Marketplaces

Marketplace = Katalog (noch nichts aktiv). **Plugin** = installierte Einheit (Skills, Commands, Agents, Hooks, MCP).

```
# Marketplace hinzufügen
grok plugin marketplace add my-org/team-plugins
grok plugin marketplace add https://gitlab.com/acme/plugins.git
grok plugin marketplace add ./my-marketplace
grok plugin marketplace list
grok plugin marketplace update
grok plugin marketplace remove <url>

# Plugin installieren
grok plugin install deploy-tools --trust
# ohne --trust: Warnung und Stop – Install bricht ab
```

Quellen: `owner/repo`, `@ref`/`@sha`, git URL, lokaler Pfad.

```
grok plugin list
grok plugin update
grok plugin enable|disable <name>
grok plugin uninstall <name>
grok plugin details <name>
```

TUI: `/plugins` (überall; auf VS-Code-Family der sichere Weg). Modal-Tabs: Hooks, Plugins, Marketplace, Skills, MCP. Keys u. a. `r` reload, `Space` enable/disable, `i` install (Marketplace).

Enable vs. Trust: Enable lädt Skills, Commands und Agents. **Trust** schaltet Hooks, MCP und LSP des Plugins scharf. Beides ist getrennt – enable allein startet keine Hook-/MCP-Ausführung.

Config-Quellen dauerhaft:

```
[[marketplace.sources]]
name = "My Team Plugins"
git = "https://github.com/my-org/plugins.git"
```

> ⚠ **ACHTUNG**
>
> Install + Trust aktiviert **Hooks und MCP** des Plugins. Nur Quellen, denen du vertraust. Untrusted Plugin-MCP erscheint als `blocked`, bis Trust greift.

Qualified Skill-Namen bei Kollision: `/deploy-tools:release`.

8.5 Hooks: erzwingen statt hoffen

Hook = Command oder HTTP an Lifecycle-Events.

Event	Blockierend?	Typischer Nutzen
`SessionStart`	nein	Setup, Env
`UserPromptSubmit`	nein	Logging
`PreToolUse`	**ja – deny**	Safety-Guard
`PostToolUse`	nein	Format, Log
`PostToolUseFailure`	nein	Alert
`Stop` / `SubagentStop` (Alias `SubagentEnd`)	**ja – Stop blocken**	„Tests müssen grün sein“
`SubagentStart`	nein	Audit
`PreCompact` / `PostCompact`	nein	Memory-Hygiene
`SessionEnd`	nein	Cleanup
`PermissionDenied` / `Notification` / `StopFailure`	nein	Audit / UI-Hinweise

Vollständige Event-Liste: User-Guide *Hooks*.

Orte (gemerged): `~/.grok/hooks/*.json`, Projekt `.grok/hooks/*.json` (**Folder-Trust nötig**), Config-TOML, managed/requirements, Plugins, optional Claude/Cursor-Compat-Dateien.

Trust: Projekt-Hooks laufen erst nach Trust (`/hooks-trust` oder `--trust`). Decision in `~/.grok/trusted_folders.toml` – **derselbe Gate** wie für repo-lokale MCP/LSP. Global `~/.grok/hooks/` ist immer trusted.

```
# Trust nur bewusst – multipliziert mit MCP
grok --trust
# in Session: /hooks-trust
```

Quick-Start PreToolUse-Guard (Skizze)

`~/.grok/hooks/guard-rm.json` (Pfad-Skizze; im Übungsrepo oft `bin/block-rm.sh` wie `beispiele/todo-cli/`, kopierfertig auch unter `vorlagen/hooks/` mit `scripts/block-rm.sh`):

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash|run_terminal_command",
        "hooks": [
          {
            "type": "command",
            "command": "bin/block-rm.sh",
            "timeout": 10
          }
        ]
      }
    ]
  }
}
```

Script liest JSON von stdin; bei gefährlichem Command stdout:

```
{"decision": "deny", "reason": "rm -rf blocked by policy"}
```

Sonst `{„decision“: „allow“}`. Matcher akzeptiert Claude-Aliase (`Bash` ↔ `run_terminal_command`). MCP-Tools matchen als `server__tool`.

Fail-open: Timeouts/Crashes des Hooks blocken **nicht** – nur explizites `deny`. Für Compliance: Monitoring der Hook-Failures im Scrollback.

Hooks in `config.toml`

```
[[hooks.PreToolUse]]
matcher = "Bash|Write|Edit"
hooks = [
  { type = "command", command = "/opt/guard/pretooluse.sh",
  ↪  timeout = 10 },
]
```

Layer sind **additiv** (user + managed + requirements).
TUI: `/hooks` – geladene Hooks sehen, toggeln (Trust-Modell beachten).

8.6 Zusammenspiel: ein Tag im Leben

1. **SessionStart-Hook** setzt `ENV=dev`.
2. **AGENTS.md** sagt: Testcommand + Don'ts.
3. Du: Feature bauen.
4. **Skill `/verify`** – reproduzierbarer Testlauf.
5. **PostToolUse-Hook** formatiert nach Edits.
6. **PreToolUse-Hook** blockt `git push --force`.
7. **Skill `/commit`** – Conventional Message.
8. MCP aus Plugin (Kapitel 9) öffnet PR – nur wenn trusted.

Anti-Pattern: Alles in Hooks pressen (langsam, fragil) oder alles nur in AGENTS.md (wird ignoriert, wenn lang und weich).

8.7 Praxis-Sessions

Session 1 – Ersten Skill bauen

```
/create-skill
```

Oder manuell:

```
Lege .grok/skills/verify/SKILL.md an:
name verify, description mit Triggern /verify und "run tests",
Body: exakte Testcommands aus AGENTS.md, dann Lint, Report, kein
↪  Push.

/verify
```

Erfolg: Skill läuft; `grok inspect` listet `project`.

Session 2 – Auto vs. nur Slash

```
---
name: release-notes
description: Draft release notes from commits since last tag.
disable-model-invocation: true
user-invocable: true
---

# „schreib release notes“ soll NICHT auto feuern
/release-notes
```

Session 3 – Hook-Guard testen

1. PreToolUse-Hook installieren, der ein harmloses Pattern denyt (z. B. `echo HOOK_DENY_TEST`).
2. Agent bitten: `echo HOOK_DENY_TEST` auszuführen.
3. Deny-Reason im Scrollback sehen.
4. Hook entfernen/disable.

Trainingsziel: hartes Gate spüren, bevor du es auf `rm` legst.

Session 4 – Marketplace / Plugin (Team-Sim)

```
# nur wenn du eine vertrauenswürdige Quelle hast
grok plugin marketplace add ./path-to-local-marketplace   # oder
↪  org/repo
grok plugin install some-safe-plugin --trust
grok plugin details some-safe-plugin
```

In der TUI `/plugins` → Skills/Hooks/MCP des Plugins prüfen. Untrusted: MCP `blocked`.

Session 5 – Skill + AGENTS.md abgrenzen

```
AGENTS.md enthält nur: "Vor Commit: /verify oder äquivalent grün."
Die Schritte stecken im Skill, nicht in der Bibel.
Bestätige die Trennung und führe /verify aus.
```

8.8 Jetzt selbst machen: Extensions-Parcours (35 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. Project-Skill `verify` anlegen und per `/verify` fahren.
> 2. `description` so schärfen, dass „run the tests“ auto oder manuell wie gewollt greift.
> 3. `grok inspect` – Source `project` verifizieren.
> 4. Globalen oder User-Hook `SessionStart` mit harmlosem `echo` testen.
> 5. `/hooks` öffnen – Hook sichtbar.
> 6. Optional: absichtlich untrusted Project-Hook-Ordner → ohne Trust still.
> 7. Notiere: Welcher Workflow wird als Nächstes ein Plugin?

8.9 Häufige Fehler

Symptom	Ursache	Fix
Skill erscheint nicht	Name/Pfad/disabled	`grok inspect`, Frontmatter
Falscher Skill bei Kollision	Unqualifizierter Name	`/local:name` / `/repo:name` / `/user:name` / `/<plugin>:name`
Auto-Skill nervt	Vage description	`when-to-use` schärfen oder disable-model-invocation
Project-Hooks tot	Kein Folder-Trust	`/hooks-trust` bewusst
Plugin Skills da, Hooks/MCP tot	enable ≠ trust	`--trust` / Trust im Modal, dann `r`
Plugin install, nichts passiert	nicht trusted / disable	`--trust`, enable, `r`

Symptom	Ursache	Fix
Hook „blockt nicht“ bei Crash	fail-open	Script härten, explizit deny

---

8.10 Checkliste

- ☐ Skills: ein Workflow pro Skill, spezifische description
- ☐ Team-Skills unter `.grok/skills/` committed
- ☐ Hooks nur für echte Gates; Trust verstanden
- ☐ Plugins nur aus trusted Sources + Inventory (`details`)
- ☐ AGENTS.md bleibt kurz – Verfahren in Skills
- ☐ `grok inspect` nach Änderungen

Was du jetzt kannst

- Skills anlegen, aufrufen und von AGENTS.md abgrenzen
- Hooks als blockierende Gates einsetzen und Trust erklären
- Marketplaces/Plugins installieren und qualifizierte Skill-Namen nutzen
- Ein Zusammenspiel Skill + Hook + Rules für den Alltag skizzieren

Nächste Schritte

→ Kapitel 09 – MCP-Server und externe Tools: externe Systeme anbinden, ohne den Agenten zum Secret-Store zu machen.

Kapitel 9

MCP-Server und externe Tools

> ⊙ **KERN**
>
> MCP hängt **externe Tools** an denselben Agent-Loop – mächtig für GitHub, Issue-Tracker, DBs – und verdoppelt Trust-, Secret- und Scope-Fragen.

9.1 Was MCP ist (ohne Protokoll-Roman)

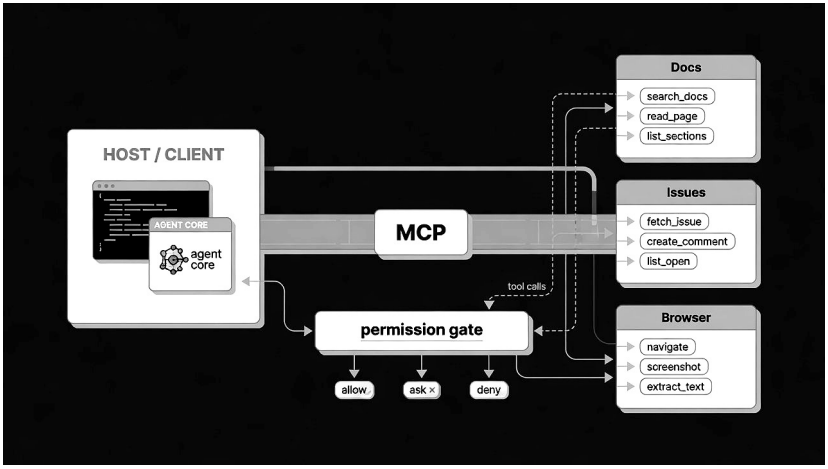


Abbildung 1: MCP ist zusätzliche Tool-Oberfläche — nicht außerhalb der Permissions.

MCP (Model Context Protocol) = Standard, mit dem Server dem Client **Tools** anbieten. Grok startet oder verbindet Server; das Modell entdeckt Tools und ruft sie auf – parallel zu Built-ins (`read_file`, Shell, …).

Beispiel-Server	Typische Tools
GitHub	Issues, PRs, Code-Search
Linear/Jira	Tickets lesen/schreiben

Beispiel-Server	Typische Tools
Postgres/Filesystem	Queries / scoped FS
Internes API-Gateway	Firmenspezifische Aktionen

Wie der Loop wirkt: Prompt → Modell will z. B. Issue anlegen → `search_tool` / `use_tool` → Server führt aus → Ergebnis zurück in den Kontext. Permissions, Always-approve, Deny-Regeln und Hooks gelten **auch** hier (Kapitel 13, 8). Siehe modelcontextprotocol.io fürs Protokoll; dieses Kapitel ist **Nutzung**.

9.2 Konfiguration: stdio vs. HTTP

Primär in `~/.grok/config.toml` unter `[mcp_servers.<name>]`, projektskopiert in `.grok/config.toml`.

stdio (lokaler Prozess)

```
[mcp_servers.filesystem]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-filesystem",
↪  "/path/to/allowed/dir"]
env = { }  # Secrets lieber ${VAR} – siehe unten
enabled = true
startup_timeout_sec = 120   # Default oft 30; Cold-Start npx/uvx eher
↪  120 (wie in vorlagen/)
tool_timeout_sec = 120
tool_timeouts = { slow_op = 300 }
```

Cold-Start `npx`/`uvx` braucht oft mehr als 30s Startup – per Server oder global (Gegenstück: `beispiele/configs/mcp.filesystem.beispiel.toml`):

```
export GROK_MCP_STARTUP_TIMEOUT_SECS=120
# oder Claude-kompatibel MCP_TIMEOUT in ms
```

HTTP / SSE (remote)

```
[mcp_servers.remote-api]
url = "https://mcp.example.com/api"
headers = { "Authorization" = "Bearer ${INTERNAL_MCP_TOKEN}" }
```

Native `url`-Form bevorzugen (Grok spricht HTTP/SSE/OAuth direkt) – nicht unnötig `npx mcp-remote` davorsetzen.

9.3 CLI: Server verwalten ohne TOML-Yoga

```
grok mcp list
grok mcp list --json

# stdio: alles nach -- ist Server-Command
grok mcp add filesystem -- npx -y
↪  @modelcontextprotocol/server-filesystem /path/to/dir

# Env wiederholbar mit -e
grok mcp add postgres -e DATABASE_URL=postgres://localhost/mydb
↪  -- npx -y @modelcontextprotocol/server-postgres

# Remote
grok mcp add --transport http sentry https://mcp.sentry.dev/mcp
grok mcp add --transport http api https://mcp.example.com/mcp \
  --header "Authorization: Bearer ${INTERNAL_MCP_TOKEN}"
grok mcp add --transport sse linear https://mcp.linear.app/sse

grok mcp remove github
grok mcp enable name         # persistiert (u. a.
↪  disabled_mcp_servers)
grok mcp disable name
grok mcp doctor              # alle
grok mcp doctor github       # einer
grok mcp doctor --json
```

Scope:

Flag	Zieldatei
default / `--scope user`	`~/.grok/config.toml`
`--scope project`	`.grok/config.toml` (team-commitbar)

`list` markiert Project-Server mit `(project)`. Namenskonflikt user vs. project:

project **ersetzt** den Server komplett (kein Field-Merge). Remove bei Doppeldefinition: `--scope` angeben.
Server-Namen: nur Letters, Numbers, Hyphen, Underscore.

9.4 Project-Scope und Precedence

Grok walked CWD → Git-Root und lädt `.grok/config.toml` je Level:

Location	Priorität
`~/.grok/config.toml`	niedrig
`<repo>/.grok/config.toml`	mittel
`<cwd>/.grok/config.toml`	hoch

Was du heute legst: Eintrag in `<cwd>/.grok/config.toml` (oder Repo-Root) schlägt User-Global für denselben Server-Namen. Project-Config liefert typischerweise `[mcp_servers]`, `[plugins]`, `[permission]` – viele andere Sektionen nur user-global (Kapitel 14).
Repo-Trust: Project-MCP/LSP/Hooks hängen am **Folder-Trust** (`trusted_folders.toml`, `/hooks-trust`, `--trust`). Untrusted Repo → Server starten nicht „heimlich“.

9.5 Tool-Namen und Aufruf im Agent

Namespace:

```
filesystem__read_file
github__create_issue
linear__save_issue
```

Im Loop:

1. `search_tool` – Tools über Server finden (Schema!).
2. `use_tool` – qualifizierten Namen + Input laut Schema.

Praxis-Prompt:

```
Nutze MCP GitHub: suche offene Bugs mit Label "payments".
Keine Issues schließen. Liste Nummer + Titel.
Zuerst search_tool, dann use_tool — erfinde keine Tool-Namen.
```

Warum search_tool erzwingen? Verhindert halluzinierte Tool-APIs und falsche Parameter.

Große Tool-Results: Truncation (Default ~20 000 Bytes), Spill unter Session-`mcp/`. Override:

```
[mcp]
max_output_bytes = 40000
```

Env: `GROK_MAX_MCP_OUTPUT_BYTES` / `MAX_MCP_OUTPUT_BYTES`. Precedence: requirements > env > repo `.grok/config.toml` > user > default.

9.6 Runtime: `/mcps`-Modal

- Slash: `/mcps` (auch über Extensions-Modal `/plugins` → Tab MCP)
- `Space` enable/disable
- Expand: Tool-Liste
- `r` refresh nach Config-Edit
- `i` OAuth
- `a` add / `x` remove (mit Confirm)

OAuth-Tokens: `~/.grok/mcp_credentials.json` (owner-only `0600`). Dieselbe Sorgfalt wie `auth.json` (Kapitel 2).

9.7 Kompatibilität: Claude, Cursor, `.mcp.json`

Source	Ort	Gate
Native	`config.toml`	immer
Claude	`~/.claude.json` etc.	`[compat.claude] mcps`
Cursor	`.cursor/mcp.json`	`[compat.cursor] mcps`
Standard	Project `.mcp.json` (cwd→root)	Import-Marker-Logik

Merge-Priorität bei Namenskonflikt: **config.toml > Claude > Cursor > `.mcp.json`**.
`grok inspect` zeigt Origin-Tags (`[cursor]`, `[claude]`).

9.8 Secrets und Sicherheit

> ⚠ **ACHTUNG**
>
> Jedes MCP-Tool ist **Code + oft Credentials + Side Effects**. Always-approve + freies GitHub-MCP in einem fremden Repo ist ein Incident mit besserer UX.

Do:

- Secrets als `${ENV}` in Config, nie committed
- Project-MCP nur in trusted Repos
- Deny-Regeln für destruktive MCP-Tools (`MCPTool(sales__delete_*)` – Kapitel 13)
- Scope Filesystem-Server auf **ein** erlaubtes Directory
- OAuth-Hosts mit Full-Disk-Encryption

Don't:

- Production-DB-URLs mit Write im Alltag
- Unbekannte Marketplaces + sofort `--trust`
- Tokens in Tickets/Chat pasten
- MCP „nur kurz global" und vergessen

Subagents erben default **connected** Parent-MCP (Kapitel 5) – spawne Write-fähige Kinder bewusst.

9.9 Beispiel-Setups (kopierbar)

Hosted HTTP (OAuth nach Connect)

```
[mcp_servers.linear]
url = "https://mcp.linear.app/mcp"
enabled = true
```

```
[mcp_servers.sentry]
url = "https://mcp.sentry.dev/mcp"
enabled = true
```

Nach Config: `/mcps` → `r` → bei Bedarf `i` für OAuth.

Intern mit Env-Header

```
[mcp_servers.internal-tools]
url = "https://mcp.internal.example.com/mcp"
enabled = true
headers = { "Authorization" = "Bearer ${INTERNAL_MCP_TOKEN}" }
```

Project-Filesystem (eng)

```
# .grok/config.toml — nur nach Team-Review committen
[mcp_servers.docs-fs]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-filesystem", "./docs"]
enabled = true
```

Vorlagen-Index: `beispiele/configs/` (wachsend), `vorlagen/config.toml.beispiel.toml`.

9.10 Praxis-Sessions

Session 1 – Server adden und doctor

```
grok mcp add filesystem -- npx -y
↪  @modelcontextprotocol/server-filesystem "$(pwd)/docs"
grok mcp doctor filesystem
grok mcp list

grok

Welche MCP-Server sind aktiv? Nenne 3 Tools vom filesystem-Server
über search_tool. Noch nichts Destruktives.
```

Session 2 – Read-only Nutzung

```
Mit filesystem-MCP: liste Markdown-Dateien unter docs/.
Keine Writes. Zusammenfassung in 5 Bullets.
```

Erfolg: Qualifizierte Tool-Namen im Scrollback; Diff im Repo unverändert.

Session 3 – GitHub/Issue-Tracker (wenn konfiguriert)

```
Suche offene Issues mit Label bug, max 5.
Nur lesen. Keine Kommentare, keine Status-Änderungen.
```

Permission-Prompts **lesen** – Issue create ist ein Side Effect.

Session 4 – Project vs. User Scope

```
grok mcp add --scope project team-docs -- npx -y
↪  @modelcontextprotocol/server-filesystem ./docs
# Secret nicht in die committed TOML:
# headers/env mit ${VAR}
git diff .grok/config.toml
```

Erfolg: Team sieht Server-Definition; Token bleibt in Env.

Session 5 – Disable ohne Löschen

In `/mcps`: Server `Space` disable → Agent bitten, Tool zu nutzen → erwartbar Fehler/kein Tool → wieder enable.

Session 6 – Deny-Denken (Vorschau Kap. 13)

```
Angenommen delete_*-Tools existieren: formuliere eine Deny-Regel
und einen PreToolUse-Hook-Matcher für server__delete_*.
Implementiere noch nichts – nur Policy-Skizze.
```

9.11 Jetzt selbst machen: MCP-Parcours (30 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. Einen **eng gescopten** filesystem-MCP auf `./docs` oder Temp-Dir.
> 2. `grok mcp doctor` grün/Fehler lesen und beheben (Timeout, npx).
> 3. Session: search_tool → use_tool, nur lesen.
> 4. Server disable/enable im Modal spüren.
> 5. Optional: einen HTTP-Server mit `${TOKEN}` statt Klartext.
> 6. `grok inspect` – MCP-Origin notieren.
> 7. Notiere: Welches Tool bräuchte eine Deny-Regel im Team?

9.12 Häufige Fehler

Symptom	Ursache	Fix
Server startet nicht	Timeout / PATH / npx	`doctor`, Startup-Timeout erhöhen
Tools „fehlen“	disabled / untrusted project	`/mcps`, Folder-Trust
Auth-Fehler	OAuth nicht connected / Header	`i` im Modal, `${VAR}` expandiert?
Halluzinierte Tools	kein search_tool	Prompt erzwingen
Riesiger Kontext	große Tool-Outputs	max_output_bytes, engere Queries
Secret im Git	Klartext in project config	`${VAR}`, git history prüfen
Windows npx	`.cmd`-Shims	Grok resolved PATHEXT; absolute Pfade ok

---

9.13 Checkliste MCP

- ☐ Server-Zweck und Scope dokumentiert
- ☐ Secrets nur Env / OAuth-Store
- ☐ Project-MCP committed ohne Tokens
- ☐ `mcp doctor` nach Setup
- ☐ Erste Nutzung read-only
- ☐ Deny/Hooks für destruktive Tools geplant

☐ Subagent-Vererbung bedacht

Was du jetzt kannst

- stdio- und HTTP-MCP konfigurieren und per CLI/Modal verwalten
- Tools über Namespace, `search_tool` und `use_tool` korrekt nutzen
- Project- vs. User-Scope, Trust und Secret-Hygiene anwenden
- Typische Startup-, Auth- und Halluzinations-Fallen beheben

Nächste Schritte

→ Kapitel 10 – Headless Mode, Scripting und CI: dieselbe Engine ohne TUI – mit Always-approve, Deny und sauberen Exit-Codes.

Kapitel 10

Headless Mode, Scripting und CI

> ⊙ **KERN**
>
> Headless (`grok -p`) ist dieselbe Engine **ohne TUI**: ein Prompt rein, Tools laufen, Text/JSON raus – in CI nur mit Always-approve **plus** Deny, Allowlist und engem Scope.

10.1 Wann Headless, wann TUI, wann ACP

Einstieg	Nutzen
TUI `grok`	Alltag, Review, Lernen (Kapitel 3)
Headless `grok -p`	Scripts, CI, einmalige Jobs, Pipelines
ACP `grok agent`	langlebige Clients, IDEs (Kapitel 11)

Alltagsschmerz: CI hängt, weil der Agent auf Permission-Prompt wartet, den niemand klickt. Gegenmittel: non-interaktiv starten **und** harte Gates (Deny/Hooks/Sandbox), nicht „yolo ohne Netz“.

10.2 Basisaufruf

```
grok -p "Your prompt here"
# Aliase/Trigger: --single, --prompt-json, --prompt-file
```

Prozess: Prompt → Tool-Loop → stdout → Exit. Kein interaktives Scrollback.

```
grok -p "Explain the top-level layout. Change nothing." --cwd
↪  ~/projects/your-app
grok -p "$(cat prompts/review.md)" -m grok-build
grok -p --prompt-file ./prompts/ci-review.txt --output-format json
```

10.3 Wichtige Flags (Arbeitsset)

In CI brauchst du selten die ganze Flag-Liste – aber **Always-approve + Deny + Output-Format + Turn-Limit** fast immer. Die Tabelle ist Nachschlagewerk; die Sessions unten bauen die typische Kombi.

Flag	Rolle
`-p` / `--single`	Headless-Prompt
`-m` / `--model`	Modell
`-c` / `--continue`	letzte Session im CWD
`-r` / `--resume`	Session per ID/Titel
`--cwd`	Working Directory
`--output-format plain\{}|json\{}|streaming-json\{}|streaming-messages-json`	Ausgabe
`--include-partial-messages`	Text/Thinking-Deltas (nur mit `streaming-messages-json`)
`--yolo` / `--always-approve`	keine Permission-Prompts (`--yolo` = Alias, oft versteckt in `--help`)
`--permission-mode`	u. a. `bypassPermissions`
`--allow` / `--deny`	Permission-Regeln (wiederholbar)
`--rules`	Session-Regeln
`--sandbox <profile>`	OS-Sandbox (Kapitel 13)
`--tools` / `--disallowed-tools`	Tool-Allow/Deny-List (**nur Headless**)
`--agents`	Inline-Subagent-Definitionen (**nur Headless**)
`--max-turns N`	Turn-Limit (**nur Headless**)
`--effort` / `--reasoning-effort`	Reasoning-Level
`--no-auto-update`	keine Update-Checks im Job
`--worktree=` / `-w`	isolierter Tree (Kapitel 6)

Nur Headless: `--tools`, `--disallowed-tools`, `--max-turns`, `--agents` in der TUI → Warnung, ignoriert. In CI genau diese Flags nutzen.

10.4 Output-Formate

plain (Default)

Menschlich lesbarer Text – gut für Logs und `echo`.

json

Ein JSON-Objekt am Ende: u. a. `text`, `stopReason`, `sessionId`, `requestId`, bei Modell-Erreichen oft `usage`, `num_turns`, `modelUsage`, Kostenfelder.

```
grok -p "Review changes for bugs" --output-format json --yolo \
  | jq -r '.text'
```

streaming-json

NDJSON-Eventstrom (native ACP-Session-Updates) – für UIs und Live-Pipelines.

streaming-messages-json

NDJSON im Anthropic-Messages-API-Wire-Format – für Clients, die dieses Streaming-Protokoll erwarten. Mit `--include-partial-messages` kommen zusätzlich inkrementelle `stream_event`-Deltas (Text/Thinking); ohne dieses Flag und bei anderen Formaten wird das Flag ignoriert (mit Warnung).

Praxis: CI-Summary → `json` + `jq`. Streaming-Dashboards → `streaming-json`. Messages-API-kompatible Clients → `streaming-messages-json`.

10.5 Permissions in Automation

Ohne TUI blockiert **ask** den Job. Deshalb:

```
grok -p "Run unit tests and summarize failures" \
  --always-approve \
  --deny 'Bash(rm -rf *)' \
  --deny 'Bash(git push *)' \
  --allow 'Bash(npm test *)' \
  --allow 'Bash(npm run lint *)' \
  --disallowed-tools "web_search,web_fetch"
```

Mechanik	Wirkung
`--always-approve`	keine Klick-Prompts
`--deny`	**gewinnt immer** – auch über YOLO
`--allow`	enge Freigaben
`--disallowed-tools`	Tool komplett weg
`--tools`	nur diese Built-ins (+ MCP-Meta oft weiter da)
Hooks	PreToolUse deny (Kapitel 8)
Sandbox	Kernel-FS/Netz (Kapitel 13)

Subagents in CI:

```
# keine Subagents
--disallowed-tools "Agent"
# nur explore sperren
--disallowed-tools "Agent(explore)"
```

Zwei Namensräume, nicht vermischen: Headless`--tools` / `--disallowed-tools` nutzen **interne Tool-IDs** (Shell oft `run_terminal_cmd`). Permission-Rules und viele Hook-Matcher nutzen Prefixe bzw. Aliase wie `Bash(...)` / `run_terminal_command` (Kapitel 8 und 13). Bei Unklarheit: Guide oder `grok inspect`.

10.6 Tool-Filter: enge Reviews

```
# Nur lesen
grok -p "Explain architecture risks" \
  --tools "read_file,grep,list_dir" \
  --output-format json

# Kein Shell, kein Web
grok -p "Static review of src/api" \
  --disallowed-tools
  ↪  "run_terminal_cmd,web_search,web_fetch,search_replace"
```

Wann Allowlist: untrusted PR, Secret-Repos, „nur begutachten“.
Wann Denylist: normaler Fix-Job, aber ohne Push/Web.

10.7 Sessions in Scripts

```
# multi-step pipeline über eine Session
# -s erwartet UUID; macOS-uuidgen liefert Uppercase → lowercasen
SID=$(uuidgen | tr '[:upper:]' '[:lower:]')   # oder feste
↪  UUID-Policy
grok -p "Step 1: find flaky tests" -s "$SID" --always-approve
↪  --output-format json
grok -p "Step 2: propose minimal fix" -r "$SID" --always-approve
↪  --output-format json

# Fork statt append
grok -p "Alternative approach" -r "$SID" --fork-session
↪  --always-approve
```

`-s` erzeugt **neue** Session mit UUID (Fehler wenn ungültig/belegt) – resumed wird mit `-r`/`-c`.

10.8 GitHub Actions – Gerüst

```
# .github/workflows/grok-review.yml (Skizze)
name: Grok review
on:
  pull_request:
    types: [opened, synchronize]

jobs:
  review:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: read
    steps:
      - uses: actions/checkout@v4
      - name: Install Grok
        run: curl -fsSL https://x.ai/cli/install.sh | bash
      - name: Review diff
        env:
          XAI_API_KEY: ${{ secrets.XAI_API_KEY }}
```

```
      run: |
        set -euo pipefail
        export PATH="$HOME/.grok/bin:$PATH"
        git fetch origin "${{ github.base_ref }}"
        RANGE="origin/${{ github.base_ref }}...HEAD"
        grok -p "Review the git diff $RANGE for bugs and
↪  security issues.
        Read-only. No commits, no push, no network installs.
        Output: severity-ordered findings with file:line." \
          --always-approve \
          --output-format json \
          --deny 'Bash(git push *)' \
          --deny 'Bash(rm -rf *)' \
          --disallowed-tools "web_fetch" \
          --max-turns 30 \
          --no-auto-update \
          | tee /tmp/grok-out.json
        jq -r '.text' /tmp/grok-out.json >>
↪  "$GITHUB_STEP_SUMMARY"
```

Lauffähiges Gegenstück im Repo:
`beispiele/configs/github-actions-grok-review.yml` (inkl. `PATH`).

> ⚠ **ACHTUNG**
>
> `XAI_API_KEY` nur als Secret. Kein Always-approve ohne Deny. Keine Write-Permissions auf den Default-Branch für den Job-Token, wenn du nur reviewen willst.

Auth in CI: API-Key (Kapitel 2), nicht Browser-Login.

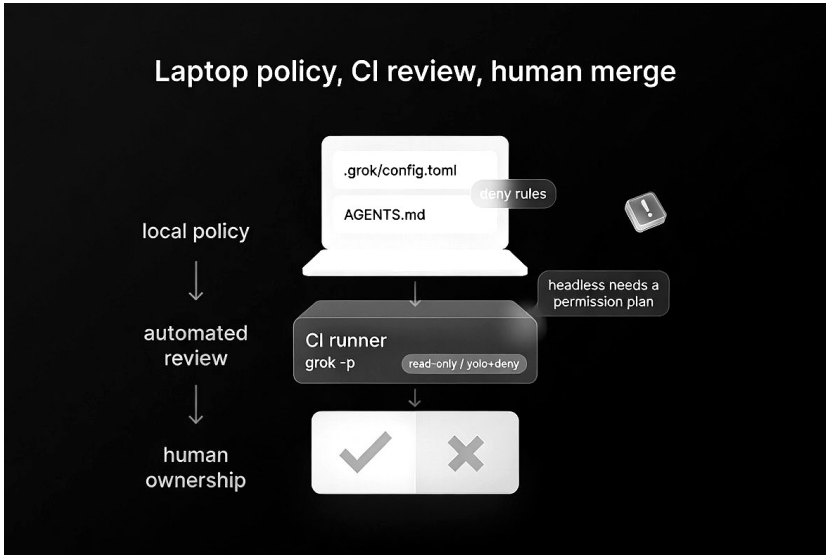


Abbildung 1: Policies auf dem Laptop, Review in CI, Merge durch Menschen.

10.9 Lokale Script-Muster

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(git rev-parse --show-toplevel)"

grok -p "Summarize uncommitted changes (git status + diff). No
↪  edits." \
  --always-approve \
  --deny 'Bash(git commit *)' \
  --deny 'Bash(git push *)' \
  --output-format plain

# Fail the pipeline if model reports blockers (heuristisch)
text=$(grok -p "…" --output-format json --always-approve | jq -r
↪  '.text')
if grep -qiE 'critical|must fix|blocker' <<<"$text"; then
  echo "$text"
  exit 1
fi
```

Heuristiken sind unscharf – für harte Gates lieber strukturierten Prompt („JSON findings array“) und `jq` auf Felder.

10.10 Praxis-Sessions

Session 1 – Read-only Headless

```
grok -p "List the three riskiest modules. Change nothing." \
  --tools "read_file,grep,list_dir" \
  --output-format plain
```

Session 2 – YOLO + Deny

Gates wie in 10.5; hier ein anderes Deny-Paar (Force-Push + Installer-Muster):

```
grok -p "Run the unit test suite and quote the first failure." \
  --always-approve \
  --deny 'Bash(git push --force *)' \
  --deny 'Bash(curl *)' \
  --max-turns 20
```

Session 3 – JSON für Downstream

```
grok -p "Return ONLY a markdown table: file | issue | severity." \
  --output-format json --always-approve \
  | jq -r '.text' > /tmp/findings.md
```

Session 4 – Multi-Turn Resume

```
grok -p "Identify one flaky test" -c --always-approve
↪  --output-format json
grok -p "Now propose a minimal fix plan only" -c --always-approve
```

Session 5 – CI dry-run lokal

Workflow-Prompt aus 10.8 gegen einen echten `git diff origin/main...HEAD` laufen lassen, Exit-Code und Summary prüfen.

10.11 Jetzt selbst machen: Headless-Parcours (25 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. Read-only `-p` mit `--tools` Restrict.
> 2. Testlauf mit `--always-approve` und zwei `--deny`.
> 3. `--output-format json | jq .sessionId,.num_turns,.text`.
> 4. `--max-turns 3` und beobachten, ob abgeschnitten wird.
> 5. Mini-Script in `scripts/grok-review.sh` committen (ohne Secrets).
> 6. Optional: GHA-Workflow als Draft-PR nur auf Fork.

10.12 Häufige Fehler

Symptom	Ursache	Fix
Job hängt	ask ohne TTY	`--always-approve` + Deny
Agent pusht/installt	zu weites YOLO	Deny, disallowed-tools, sandbox
Leere Ausgabe	falsches format/jq	`plain` testen, raw JSON ansehen
Zu teuer/lang	kein Turn-Limit	`--max-turns`, enger Prompt
Auth fail	kein Key in CI	`XAI_API_KEY` Secret
Tools-Flag greift nicht	in TUI benutzt	Headless `-p`

10.13 Checkliste CI

- ☐ API-Key als Secret
- ☐ Always-approve **und** Deny-Minimum
- ☐ Scope im Prompt (read-only vs. fix)
- ☐ Output-Format fürs Parsing
- ☐ max-turns / Timeouts im Runner
- ☐ Least-privilege GitHub-Permissions
- ☐ Keine Secrets im Prompt-Log

Was du jetzt kannst

- Headless-Aufrufe mit Output-Formaten und Session-Flags fahren
- YOLO mit Allow/Deny/Tool-Filter und Turn-Limits absichern
- Ein CI-Gerüst und lokale Scripts ohne hängende Prompts bauen

Nächste Schritte

→ Kapitel 11 – ACP, IDE-Integration und Bots: langlebige Agent-Server für Editoren und eigene Clients.

Kapitel 11

ACP, IDE-Integration und Bots

> ⊙ **KERN**
>
> ACP macht Grok zum **langlebigen Agent-Server** (JSON-RPC): IDEs und eigene Clients steuern Sessions – dieselbe Engine wie TUI/Headless, andere Lebensdauer und Transport.

11.1 Headless vs. Agent Mode

	Headless `grok -p`	**Agent** `grok agent …`
Lebensdauer	ein Prompt, dann Exit	Prozess bleibt / reconnect
Transport	CLI stdout	stdio, WebSocket serve, Relay
Typische Clients	Scripts, CI	IDEs, SDKs, Bots
Permissions	immer non-interaktiv denken	interaktiv möglich, oft YOLO

Merksatz: Einmal-Job → `-p`. Editor/Bot mit mehreren Turns → `agent`.

11.2 Was ACP abdeckt

Agent Client Protocol über JSON-RPC:

- Sessions erzeugen, laden, fortsetzen
- Prompts und gestreamte Antworten
- Tool-Call-Updates
- Reasoning/Thought-Streams
- Permission-Requests (wenn nicht always-approve)

Architektur skizziert:

```
IDE / SDK / Bot
      │  JSON-RPC (stdio oder WS)
```

```
      ▼
grok agent
  Session Manager · Tools · MCP
```

11.3 stdio – der lokale Standardweg

```
grok agent --always-approve stdio
grok agent --always-approve --model grok-build stdio
```

Flags **nach** `agent` und **vor** dem Mode-Namen; mode-spezifische Flags **danach** (`serve --bind …`).

Flag	Bedeutung
`-m` / `--model`	Modell
`--always-approve` / `--yolo`	ohne Permission-UI
`--reauth`	Auth vor Start
`--agent-profile <PATH>`	Profil-Datei
`--leader` / `--no-leader`	shared Leader vs. lokal

Clients: ACP-fähige IDEs/Editoren (z. B. Zed, Neovim, Emacs-Integrationen), Custom-Tools, SDKs. Konkrete Marketplace-Extensions ändern sich – Konzept bleibt: Client startet `grok agent … stdio` und spricht ACP.

Transport-Fokus: stdio ist der lokale Standard (Client startet den Prozess). Härtung (Always-approve + Deny + Sandbox) bündelst du in 11.8 und Kapitel 13 – hier nur der Startweg.

Session-seitig YOLO (Client-Meta, Implementierung abhängig):

```
{
  "cwd": "/path/to/project",
  "mcpServers": [],
  "_meta": { "yoloMode": true }
}
```

11.4 Server Mode (WebSocket)

```
grok agent --always-approve serve \
  --bind 127.0.0.1:2419 \
  --secret YOUR_TOKEN
```

- Client verbindet per WebSocket, auth mit Secret.
- Ohne `--secret`: generierter Token beim Start, oder Env `GROK_AGENT_SECRET`.
- State überlebt Client-Reconnects.

> ⚠ **ACHTUNG**
>
> `--bind 0.0.0.0` ohne Netz-Härtung ist ein offener Agent mit Tool-Rechten. Default: **localhost**. Secret wie ein API-Key behandeln. Reverse-Proxy + TLS, wenn remote.

11.5 Relay / Headless-Agent über WebSocket

```
grok agent --always-approve headless \
  --grok-ws-url wss://your-relay.example.com/ws
```

Für Browser/Remote-Clients über gemeinsamen Relay – Ops-Thema (Auth, Allowlists, Logging). Nicht „mal eben öffentlich“.

11.6 Session-Lifecycle (Protokoll-Basics)

1. **initialize** – Capabilities
2. **session/new** – CWD, optionale MCP-Liste, Meta (yolo, …)
3. **session/prompt** – User-Messages
4. **session/update** – Stream (Text, Tools, Thoughts)
5. **Permissions** – Approve/Deny oder Mode-Policy

Als **Nutzer** siehst du das in der IDE als Chat + Diffs. Als **Integrationsbauer** loggst du JSON-RPC (Permissions, CWD, Tool-Calls) – sonst debuggst du „hängt“ und „falscher Tree“ blind. Teste mit always-approve + Deny.

ACP-Extensions (Auszug aus Guide): u. a. Git-Worktree-Methoden (`x.ai/git/worktree/*`), Session-Helpers – nützlich für isolierte IDE-Sessions (Kapitel 6).

11.7 IDE-Alltag: was gleich bleibt

Alltagsschmerz: Falscher CWD, Diff akzeptiert ohne Lesen, Always-approve „weil die IDE fragt nervt“. Unabhängig vom Editor-Skin gelten dieselben Steuerknöpfe:

Thema	In der IDE genauso wichtig
CWD / geöffnetes Projekt	falscher Tree = falsche Edits
`AGENTS.md`	wird geladen wie in der TUI
Plan Mode	wo die Integration ihn exposed
Diff-Review	nie blind Accept
MCP/Hooks	Trust-Folder, Plugin-Trust
Always-approve	nur mit Deny/Sandbox

Lernpfad: Konzepte in der TUI lernen (Kapitel 2–9), IDE als Surface dazunehmen – nicht umgekehrt.

11.8 Bots und langlebige Worker

Typische Muster:

1. **PR-Bot:** Webhook → Worker startet `grok -p` (einfach) oder hält `agent serve` (komplex, multi-turn).
2. **Eval-Harness:** ACP stdio, feste Prompts, JSON-Metriken.
3. **Pair-Bot:** serve auf localhost, UI im Intranet.

```
# Eval-Skizze: stdio-Client steuert Turns
# Harte Limits: [permission] deny in .grok/config.toml (agent-CLI
↪  listet --deny nicht)
grok agent --always-approve stdio
```

```
# .grok/config.toml (Bot-Worker)
[permission]
deny = [
  "Bash(rm -rf *)",
  "Bash(git push --force *)",
  "Bash(git push -f *)",
]
```

Hardening-Checkliste für Bots:

- ☐ Secret/API-Key nur Env
- ☐ Always-approve + Deny (Config/Hooks) + ideal Sandbox
- ☐ Bind localhost oder mTLS
- ☐ max cost/turns clientseitig
- ☐ Audit-Log der Prompts (ohne Secrets)
- ☐ Least-privilege Repo-Token (read vs. write)

11.9 Praxis-Sessions

Session 1 – stdio smoke (lokal)

```
# Terminal A
grok agent --always-approve stdio
# Terminal B / Client: initialize + session/new + kurzer Prompt
# (SDK oder minimaler JSON-RPC-Client)
```

Ohne eigenen Client: IDE-Extension mit Grok-ACP verbinden und einen Read-only-Prompt senden.

Session 2 – serve + Secret

```
export GROK_AGENT_SECRET="dev-only-change-me"   # nur lokal/Env –
↪  nicht committen
grok agent --always-approve serve --bind 127.0.0.1:2419 --secret
↪  "$GROK_AGENT_SECRET"
```

Mit WS-Client connecten; falsches Secret muss scheitern.

Session 3 – IDE Diff-Disziplin

In der IDE: kleiner Edit-Auftrag → Diff im Editor reviewen → Accept/Reject bewusst. Vergleiche mit TUI-Scrollback-Diff (Kapitel 3).

Session 4 – YOLO vs. ask im Agent

Einmal **ohne** `--always-approve` (wenn Client Permissions supported): Permission-UI erscheint.
Dann mit YOLO + Deny: spüre den Unterschied für Automation.

Session 5 – Bot-Policy skizzieren

```
Entwirf eine Policy für einen PR-Review-Bot:
erlaubte Tools, Deny-Liste, max-turns, ob -p oder agent serve,
welche GitHub permissions. Noch kein Code.
```

11.10 Jetzt selbst machen: ACP-Parcours (20 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. `grok agent --help` / Mode-Hilfe lesen.
> 2. stdio **oder** serve einmal starten und sauber beenden.
> 3. Secret-Handling notieren (Env vs. Flag).
> 4. In deiner IDE: ob/wie Grok/ACP angebunden ist – einen Read-only-Turn.
> 5. Diff accept **nach** Lesen.
> 6. Bot vs. Headless: eine Tabelle „wann was“ für dein Team skizzieren.

11.11 Häufige Fehler

Symptom	Ursache	Fix
Client hängt auf Permissions	kein YOLO, kein UI	`--always-approve` + Deny
Connection refused	falscher Bind/Port	`127.0.0.1`, Port prüfen
Unauthorized	Secret mismatch	`GROK_AGENT_SECRET`
Agent schreibt im falschen Repo	cwd in session/new	CWD explizit setzen
Öffentlich erreichbar	0.0.0.0 ohne Schutz	localhost / VPN / Auth

---

11.12 Checkliste

- ☐ Transport gewählt (stdio vs. serve)
- ☐ Always-approve nur mit Gates
- ☐ Secret und Bind gehärtet

- ☐ CWD/Trust korrekt
- ☐ Diff-Review-Prozess in der IDE
- ☐ Bot least privilege

Was du jetzt kannst

- ACP von Headless unterscheiden und stdio/serve starten
- IDE- und Bot-Integrationen mit denselben Safety-Hebeln absichern
- Einen minimalen Lifecycle (initialize → session → prompt) erklären

Nächste Schritte

→ Kapitel 12 – Memory, Sessions und Kontextmanagement: was auf Disk liegt, wie du resumest, compactest und Memory bewusst einsetzt.

Kapitel 12

Memory, Sessions und Kontextmanagement

> ⊙ **KERN**
>
> Sessions speichern den **kompletten Verlauf** auf Disk; Memory speichert **übergreifende Fakten** (experimentell); Compact und neue Sessions halten den **Kontextfenster-Alltag** lebensfähig.

12.1 Vier Speicher, die du nicht vermischen solltest

Alltagsschmerz: Eine 4-Stunden-Session driftet, Kosten steigen, der Agent widerspricht sich. Compact hilft, verliert aber Nuancen. Memory hilft über Tage – und kann Falsches „erinnern“, wenn du Müll speicherst. `AGENTS.md` ist kein Session-Speicher, sondern der Team-Kompass – trotzdem landet es im Kopf neben den dreien und wird oft verwechselt.

Speicher	Was	Lebensdauer
Kontextfenster	was das Modell *jetzt* sieht	Turn / bis Compact
Session	voller Verlauf + Tools + Snapshots	`~/.grok/sessions/…`
Memory	indexierte Fakten über Sessions	`~/.grok/memory/…` (opt-in)
AGENTS.md	Team-Kompass (weich, Kapitel 7)	Repo – nicht session-lokal

---

12.2 Session-Layout auf Disk

Basis: `~/.grok` bzw. `$GROK_HOME`. **Warum das Layout kennen?** Resume und Debug hängen an `updates.jsonl` (autoritativ), nicht am Titel in der UI. Feedback und Rewind-Punkte sitzen daneben – wer nur `summary.json` anstarrt, sieht Metadaten, nicht den Gesprächsstrom.

```
~/.grok/sessions/<encoded-cwd>/<session-id>/
  summary.json
```

```
  updates.jsonl          # autoritativer Conversation-Stream
  chat_history.jsonl
  plan.json              # Tasks/TODO-State
  rewind_points.jsonl    # Snapshots für /rewind
  signals.json
  feedback.jsonl         # Ratings/Feedback (falls genutzt)
  compaction_checkpoints/
  subagents/             # Meta; Kinder leben im normalen
↪  Session-Tree
```

`encoded-cwd` = URL-encoding des Working Directories (bei Extremlänge Slug+Hash + `.cwd`).

12.3 Sessions steuern

Warum IDs statt Titel in Scripts? Titel sind lesbar und kollidieren; IDs sind stabil. `list`/`search` sind die Inventar-Tools; Löschen ist permanent.

Ziel	TUI	CLI
Neu	`/new` (`/clear`)	neuer `grok`-Start
Resume	`/resume`, Welcome, `Ctrl+S`	`grok -r <id\{}|title>`, `grok -r` (latest), `grok -c`
Fork	`/fork [--worktree\{}|--no-worktree] [directive]`	`--fork-session` mit resume
Rename	`/rename` / `/title`	–
Dashboard	`/dashboard` (`/sessions`)	–
Quit	`/quit`	Exit-Code Headless
Home	`/home`	–

```
grok sessions list
grok sessions list --limit 50
grok sessions search "rate limit"
grok sessions delete <session-id>   # permanent (CLI); TUI:
↪  /delete oder /resume → d y
```

`list` gruppiert oft nach Worktree-Label.

12.4 Rewind vs. Compact

`/rewind` (oder 2× Esc, idle, leerer Prompt)

1. Rewind-Punkte (pro User-Prompt)
2. Punkt wählen
3. **Dateien** auf Snapshot zurück
4. Conversation abschneiden

> ⚠ **ACHTUNG**
>
> Rewind ändert die Disk. Ungesicherte spätere Edits sind weg, außer Git hat sie. Vor riskanten Rewinds: `git status` / commit.

`/compact [Hinweis]`

Komprimiert History im **Kontext**, speichert Checkpoints unter der Session. Optionaler Hinweis steuert, was bleiben soll:

```
/compact keep the auth decision: cookie sessions, no JWT
```

Auto-Compact default bei **85 %** Kontext:

```
[session]
auto_compact_threshold_percent = 85
```

Plan Mode: Compact behält Plan-Mode-State (Kapitel 4).

Situation	Mittel
Agent hat falschen Pfad eingeschlagen, Dateien kaputt	`/rewind` (+ Git)
Kontext voll, Richtung stimmt	`/compact` mit Hinweis
Thema komplett neu	`/new`
Parallel anderer Ansatz	`/fork` (+ Worktree)

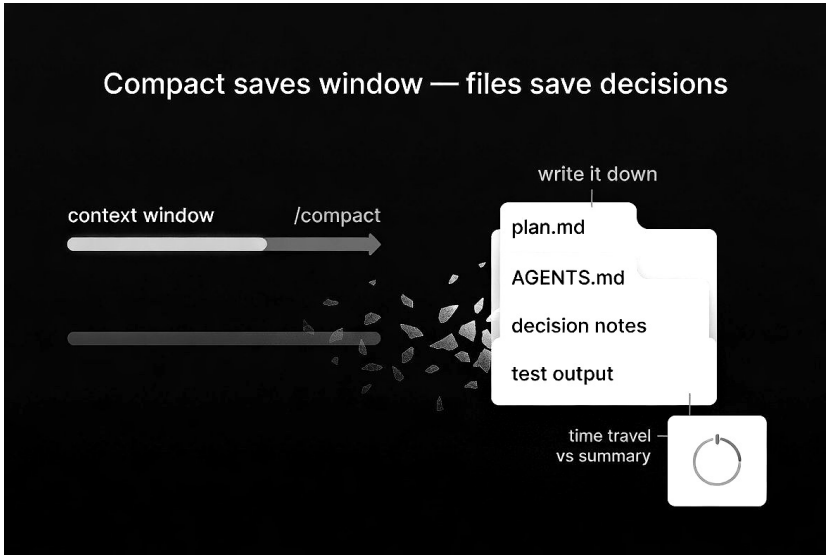


Abbildung 1: Compact spart Fenster — Dateien retten Entscheidungen.

12.5 Kontext lesen und steuern

```
/context          # Breakdown: System, Messages, Reasoning, Free,
↪  Tools/Skills/MCP-Kosten
/session-info     # Auth, Model, Turns, Usage — Aliases /status
↪  /info
```

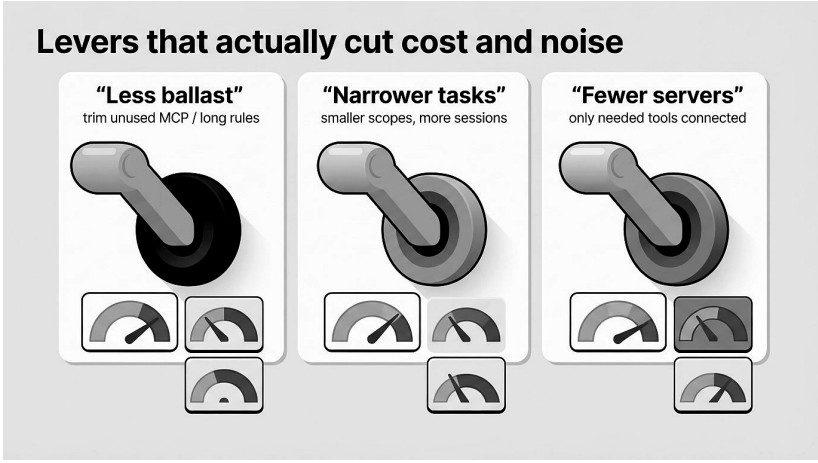


Abbildung 2: Weniger Ballast, engere Tasks, weniger Server — die Hebel, die zählen.

Hygiene-Regeln:

1. Enge Prompts > lange Chats.
2. Große Logs nicht ungefiltert pasten – Datei + `@` oder Pfad.
3. Nach Meilenstein: compact **oder** new.
4. Stabile Fakten → AGENTS.md; episodische Learnings → Memory/flush.
5. MCP-Outputs sind groß – Queries eng (Kapitel 9).

12.6 Memory (experimentell, default aus)

Einschalten

```
grok --experimental-memory
export GROK_MEMORY=1
```

```
[memory]
enabled = true
```

Aus (gewinnt hart)

```
grok --no-memory
export GROK_MEMORY=0
```

Priorität: `--no-memory` > `--experimental-memory` > `GROK_MEMORY` > config > default off.
Mid-session: `/memory on|off` (session-scoped, schreibt nicht zwingend config). Modal: `/memory`, Toggle `t`.

Speicherorte

Pfad	Scope
`~/.grok/memory/MEMORY.md`	global
`~/.grok/memory/<slug>-<hash8>/MEMORY.md`	Workspace
`…/sessions/`	Session-Logs

Workspace-Identität: `origin` als `org/repo` wenn Git-Remote, sonst Pfad.
Clones/Worktrees desselben origin teilen Memory – Feature und Risiko.
Index: SQLite FTS5 + optional Vektoren.

Automatik vs. reichhaltig

- **Session-Ende:** schlanke Metadaten-Summary (Topics, Counts) – kein LLM, trivial Sessions skip. Abschalten: `session.save_on_end = false`.
- **`/flush`:** reichhaltige Extraktion (Decisions, Patterns) – vor Compact oder Tagesende.
- **`/dream`:** Konsolidierung in Topics.
- **`/remember …`:** sofortige Notiz (laut Doku auch ohne vollen Memory-Stack verfügbar).

```
/remember staging deploy uses the eu-west cluster
/flush
/dream
```

Policy: Keine Secrets ins Memory. Falsche Erinnerungen aktiv löschen/korrigieren im `/memory`-UI oder per CLI (`grok memory clear --workspace|--global|--all`, mit `--yes` ohne Prompt). Memory ersetzt **nicht** AGENTS.md für Team-Standards.

12.7 Praxis-Sessions

Session 1 – Session finden und fortsetzen

```
grok sessions list
grok -c
```

```
/session-info
/context
```

Session 2 – Compact mit Hinweis

Lange Explore-Session simulieren (viele Reads), dann:

```
/compact keep the chosen approach: outbox pattern, no event
↪  sourcing rewrite
```

Prüfen: `/context` freier, Kernentscheidung noch bekannt.

Session 3 – Rewind nach Fehlversuch

```
# absichtlich schlechten Edit erlauben
# dann:
/rewind
```

Git-Status vergleichen; verstehen, was zurückrollte.

Session 4 – Memory opt-in

```
GROK_MEMORY=1 grok
```

```
/remember We use pnpm, never npm, in this monorepo.
/flush
```

Neue Session mit Memory: „Welchen Package-Manager nutzen wir?“ – Erwartung: Erinnerung oder AGENTS.md; beides abgleichen.

Session 5 – New vs. Compact

Zwei Features hintereinander: Feature A abschließen → `/new` für Feature B. Vergleich: beide in einer Session mit nur Compact – wo driftet die Qualität?

12.8 Jetzt selbst machen: Kontext-Parcours (25 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. `grok sessions list` und eine alte Session resumieren.
> 2. `/context` und `/session-info` screenshot/notieren.
> 3. Einmal `/compact` mit spezifischem Keep-Hinweis.
> 4. Einmal `/rewind` auf eine sichere Spielwiese (Git clean strategy).
> 5. Memory kurz enablen, `/remember` + neue Session, dann `--no-memory`.
> 6. Team-Regel skizzieren: wann new, wann compact, was in AGENTS.md vs. Memory.

12.9 Häufige Fehler

Symptom	Ursache	Fix
„Er hat alles vergessen“	Compact/new ohne Keep	Hinweis formulieren; AGENTS.md
Kontext voll	lange Tools/MCP	engere Tools, new session
Rewind-Schock	uncommittete Arbeit	Git zuerst
Falsche Memory	Müll geflusht	Memory editieren/löschen
Memory leak über Repos	shared origin hash	bewusst; secrets meiden
Resume falsche Session	Titel-Kollision	UUID

12.10 Checkliste

- ☐ Session-IDs für Automation
- ☐ Compact-Schwelle verstanden
- ☐ Rewind nur mit Git-Netz
- ☐ Memory opt-in + keine Secrets
- ☐ AGENTS.md für Team-Wahrheit
- ☐ `/context` bei Qualitätsabfall

Was du jetzt kannst

- Session-Storage erklären und Sessions listen/resumen/forken
- Compact, Rewind und New gezielt wählen
- Memory experimentell nutzen und von AGENTS.md trennen

Nächste Schritte

→ Kapitel 13 – Permissions, Sandbox und Sicherheit: Modes, Allow/Deny, Folder-Trust und Kernel-Sandbox.

Kapitel 13

Permissions, Sandbox und Sicherheit

> ⊙ **KERN**
>
> Permissions entscheiden **ob** ein Tool-Call durchkommt; die Sandbox begrenzt **was das OS dem Prozess erlaubt** – beides brauchst du, und Deny gewinnt immer.

13.1 Bedrohungsmodell in einem Absatz

Ein Coding-Agent mit Shell und Netzwerk ist ein **Remote-gesteuerter Junior mit root-nahem Potential im User-Account**. Risikoquellen: Prompt-Injection aus untrusted Code/Issues, zu weites Always-approve, MCP mit Secrets, `curl|bash`, Force-Push, `.env`-Leaks. Gegenmittel: **Modes + Rules + Hooks + Sandbox + Trust + Review**.

13.2 Permission Modes

Jeden Tool-Call entscheidet der **Mode**, *wie oft* du klickst – Rules entscheiden, *was nie geht*. Modes ohne Deny sind Komfort; Deny ohne Mode-Verständnis ist reines Rätselraten.

Mode	Verhalten	Alltag
`default` / **ask**	Read-only oft frei; Rest fragt	interaktive TUI
`acceptEdits`	Edits ohne Prompt	lokal, Diffs später
`auto`	Classifier lässt vieles durch, eskaliert/blockt Rest	weniger Klicks
`dontAsk`	nur Vorab-Erlaubtes / read-only Builtins	strikte Allowlists
`bypassPermissions` / **always-approve**	Tools laufen ohne Prompt*	CI, ACP, Automation

Mode	Verhalten	Alltag
`plan`	Compat-Name – echtes Plan Mode separat (Kap. 4)	–

* **Deny-Regeln und Hooks greifen weiter** unter Always-approve. Shell-`ask`-Segmente können weiter prompten; **non-Shell-`ask`** oft nicht – Details im User-Guide Permissions.

Setzen:

```
# TUI
# Shift+Tab · Ctrl+O · /always-approve · /auto · /settings

grok --always-approve -p "…"
grok --permission-mode auto
grok --yolo   # Alias
```

```
[ui]
permission_mode = "ask"   # oder always-approve, auto, …
```

Org-Lock Always-approve aus:

```
# requirements.toml
[ui]
disable_bypass_permissions_mode = true
```

Always-approve und auto sind gegenseitig exklusiv (Always-approve gewinnt, wenn beides angefordert).

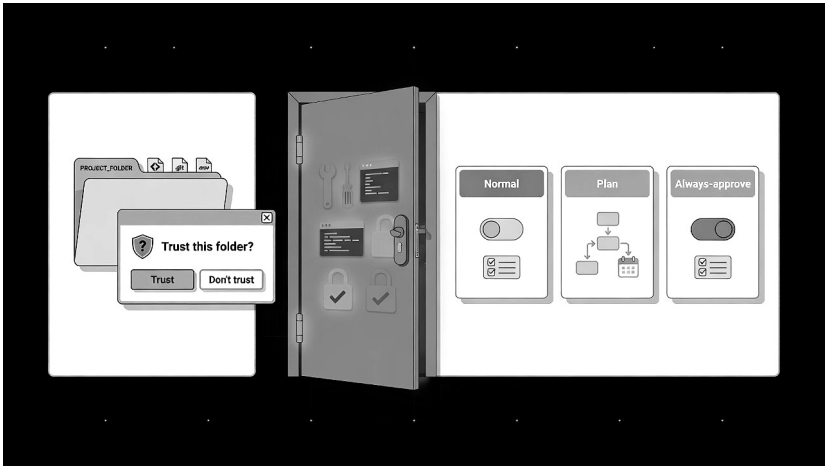


Abbildung 1: Trust ist die grobe Tür; Normal / Plan / Always-approve steuern die Session.

13.3 Autorisierungs-Pipeline

Reihenfolge bei jedem Tool-Call:

1. **PreToolUse-Hooks** – können **deny** (allow skippt nicht die Rest-Checks)
2. **Permission Rules** – `deny > ask > allow`
3. **Remembered grants** (interaktive „Always allow“, project-scoped) – nicht unter YOLO
4. **Built-in auto-approvals** (read-only Tools + read-only Shell-Segmente)
5. **Mode-Policy** (prompt / auto / deny)

Always-approve verkürzt nach Step 2: Deny/Hooks/relevante Shell-ask bleiben; remembered grants entfallen.

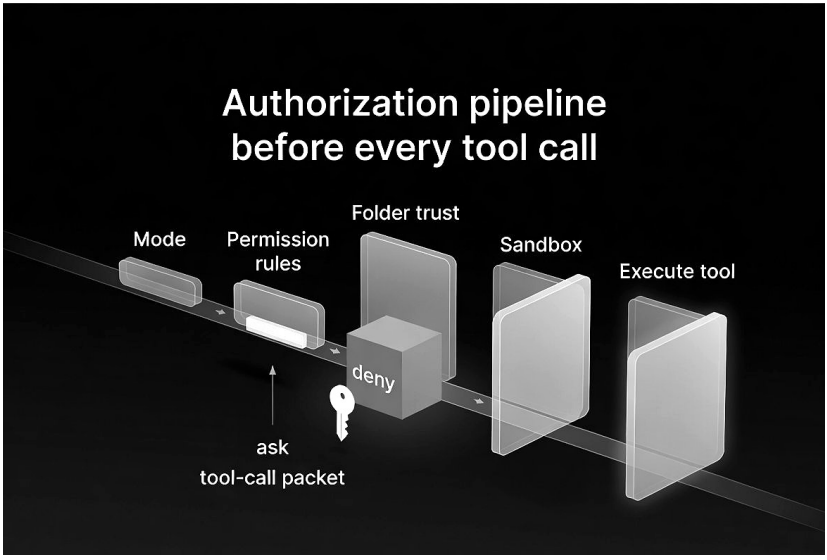


Abbildung 2: Mode → Rules → Trust → Sandbox — die Kette vor dem Tool-Call.

13.4 Was default nie fragt (solange kein ask/deny)

Warum das zählt: Ohne Segmentierung wäre jede Shell-Kette ein einziges Permission-Urteil – entweder nervt sie ständig oder sie ist gefährlich freizügig. Der Harness splittet Chains und erlaubt harmlose Teile, blockt riskante.

Tools u. a.: `read_file`, `list_dir`, `grep`, `web_search`, `todo_write`, Subagent-Control, Skill-Invoke.

Shell-Segmente (word-boundary), u. a.: `ls`, `cat`, `pwd`, `git status|diff|log|…`, `rg`/`grep` (ohne riskante Preprocessor), read-only `kubectl get|logs|describe`.

`ls && rm -rf /` → `ls` read-only, `rm` nicht. Chaining wird segmentiert.

13.5 Rules konfigurieren

CLI

```
grok -p "Review API" \
  --allow 'Bash(git *)' \
```

```
  --allow 'Read' \
  --deny 'Bash(rm -rf *)' \
  --deny 'MCPTool(sales__delete_*)'
```

config.toml

```
[permission]
allow = [
  "Bash(npm test *)",
  "Bash(npm run lint *)",
  "Bash(cargo test *)",
]
deny = [
  "Bash(rm -rf *)",
  "Bash(git push --force *)",
  "Bash(git push -f *)",
  "Read(**/.env)",
  "Read(**/.env.*)",
  "Read(secrets/**)",
  "Read(**/secrets/**)",
]
```

> ***Matching:*** *Bare-Filename wie `Read(.env)` matcht nur den **exakten** String `.env` – nicht `subdir/.env`. Für Tiefe: `**/.env`. Pfade werden nicht kanonisiert; wo der Agent absolute Paths liest, zusätzlich absolute Patterns setzen.*

Oder strukturierte `rules = [{ action = „allow“, tool = „bash“, pattern = „git *“ }, …]` (siehe aktuelle Doku-Syntax).

Scopes

Scope	Datei	Teilen
Global	`~/.grok/config.toml`	nein
Project	`.grok/config.toml`	ja (commit)

Scope	Datei	Teilen
Personal im Projekt	`.claude/settings.local.json`	nein (gitignore; Claude-Compat-Pfad, von Grok nativ gelesen – kein Claude Code nötig)
Interactive grants	intern, per project	nein

Discovery: `.grok/config.toml` von Root→CWD. **Global deny schlägt project allow.**

Syntax-Beispiele: `Bash(git *)`, `Edit(**/*.rs)`, `Read(src/**)`, `MCPTool(server__*)`, bare `Grep`.

13.6 Folder-Trust

Project-**Hooks**, **MCP**, **LSP** aus dem Repo laufen erst nach Trust (`/hooks-trust`, `--trust`, Store `~/.grok/trusted_folders.toml`). Ein Trust-Grant cascaded in Subdirs und gilt für MCP+LSP+Hooks gemeinsam.

```
GROK_FOLDER_TRUST=0   # ungated – nur wenn du die Konsequenz
↪  willst
```

> ⚠ **ACHTUNG**
>
> Untrusted Repo + sofort `--trust` + YOLO + MCP = willentliche Code-Ausführung aus dem Tree. Trust ist eine **Sicherheitsentscheidung**, kein UX-Klick.

13.7 Sandbox (OS-Level)

Default: **aus**. Kernel-Primitiven: Landlock (Linux), Seatbelt (macOS).

```
grok --sandbox workspace
grok --sandbox read-only
grok --sandbox strict
grok -p "Review only" --sandbox read-only --always-approve
```

Profile	Read	Write	Child-Netz*	Nutzen
`off`	frei	frei	frei	Default
`workspace`	überall	CWD + `~/.grok` + tmp	an	Alltag
`devbox`	überall	fast alles außer `/data`	an	Disposable VM
`read-only`	überall	nur grok/tmp	Linux block	Review
`strict`	CWD+System	CWD+grok/tmp	Linux block	Untrusted

* Child-network block **Linux**; macOS oft no-op.

Custom:

```
# ~/.grok/sandbox.toml oder .grok/sandbox.toml
[profiles.project]
extends = "workspace"
restrict_network = true
deny = ["**/.env", "**/*.pem", "/data/shared-secrets"]
read_only = ["/data"]
read_write = ["/tmp/scratch"]

grok --sandbox project
```

Unter workspace/read-only/strict: Kernel **write-deny** auf globale Hook-Pfade (`~/.grok/hooks/`, …), damit Sandbox-Agenten User-Hooks nicht umschreiben. Symlink-Tricks → Start verweigert.

13.8 Team-Baseline (empfohlen)

```
# .grok/config.toml (committed)
[permission]
deny = [
  "Bash(rm -rf *)",
  "Bash(curl *)",
  "Bash(wget *)",
  "Bash(git push --force *)",
  "Bash(git push -f *)",
  "Read(**/.env)",
```

```
  "Read(**/.env.*)",
  "Read(**/secrets/**)",
]
allow = [
  "Bash(npm test *)",
  "Bash(npm run lint *)",
  "Bash(pytest *)",
  "Bash(git status *)",
  "Bash(git diff *)",
  "Bash(git log *)",
]
```

Plus:

- PreToolUse-Hook für org-spezifische Guards (Kapitel 8)
- CI: always-approve + dieselbe Deny-Liste + ideal Sandbox
- Keine Production-Credentials im Agent-Env

13.9 Praxis-Sessions

Session 1 – Ask spüren

TUI Normal Mode: Shell `npm test` erlauben, `rm -rf /tmp/nonexistent-xyz` **ablehnen** und Erklärung fordern.

Session 2 – Deny schlägt YOLO

```
grok -p "Run: rm -rf /tmp/should-be-blocked-by-deny" \
  --always-approve \
  --deny 'Bash(rm -rf *)'
```

Erwartung: block / Agent muss umplanen.

Session 3 – Read-only Sandbox Review

```
grok -p "Security review of auth module. No file writes to
↪  project." \
  --sandbox read-only \
  --always-approve \
  --disallowed-tools "search_replace"
```

Session 4 – Project Deny committen

`.grok/config.toml` Deny-Minimum anlegen, `grok inspect` / Session testen, PR an Team.

Session 5 – Trust bewusst

Frisches Demo-Repo mit project hook: ohne Trust still, mit `/hooks-trust` aktiv – dann Trust-Entscheidung dokumentieren.

13.10 Jetzt selbst machen: Safety-Parcours (30 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. Mode-Tabelle aus 13.2 auswendig skizzieren.
> 2. Zwei Denys + einen Allow in User- oder Project-Config.
> 3. YOLO+Deny-Test (Session 2).
> 4. Einmal `--sandbox workspace` im Alltag.
> 5. Einmal `--sandbox read-only` für Review.
> 6. Folder-Trust-Store-Pfad notieren; **kein** Blind-Trust unbekannter ZIPs.
> 7. Team-Baseline-PR öffnen.

13.11 Häufige Fehler

Symptom	Ursache	Fix
CI hängt	ask ohne TTY	always-approve
YOLO „ignoriert Deny“	Missverständnis	Deny bleibt; Syntax prüfen
Allow greift nicht	global deny / Pattern	Matching-Guide, ** vs *
Sandbox start fail	Hooks-Symlink / Policy	`/doctor`, Profile
Netz in strict auf macOS	no-op child net	nicht auf macOS verlassen
`.env` im Kontext	Read erlaubt	Deny `Read(**/.env)` (+ absolute Forms)

13.12 Checkliste Sicherheit

- ☐ Default interaktiv: ask (nicht YOLO)
- ☐ Committed Deny-Minimum
- ☐ CI: YOLO + Deny + max-turns
- ☐ Sandbox für untrusted/review
- ☐ Trust nur bekannte Repos
- ☐ MCP destruktive Tools denylist
- ☐ Diff-Review vor Commit

Was du jetzt kannst

- Modes und die Autorisierungs-Pipeline erklären
- Allow/Deny in CLI und Config setzen und begründen
- Sandbox-Profile wählen und Folder-Trust bewusst handhaben
- Eine Team-Baseline skizzieren und testen

Nächste Schritte

→ Kapitel 14 – Konfiguration und Custom Models: Precedence, `config.toml`, Modelle und typisches „Config greift nicht“-Debugging.

Kapitel 14

Konfiguration und Custom Models

> ⊙ **KERN**
>
> Grok liest Einstellungen in fester **Precedence** (CLI > Env > config.toml > managed > Defaults) – wer die Schichten kennt, debuggt „greift nicht“ in Minuten statt Stunden.

14.1 Precedence (auswendig lernen)

1. **CLI-Flags** (`--yolo`, `-m`, `--sandbox`, …)
2. **Environment Variables** (`XAI_API_KEY`, `GROK_MEMORY`, …)
3. **`config.toml`** (`~/.grok/config.toml` und projektrelevante Teile)
4. **Managed / requirements** (Org: `managed_config.toml`, `requirements.toml`)
5. **Built-in Defaults**

Project `.grok/config.toml`: liefert vor allem `[mcp_servers]`, `[plugins]`, `[permission]` (und verwandte team-sharebare Blöcke). Viele UI-/Model-Defaults bleiben **user-global**. MCP-Namen: Project **ersetzt** User-Server gleichen Namens komplett (Kapitel 9).

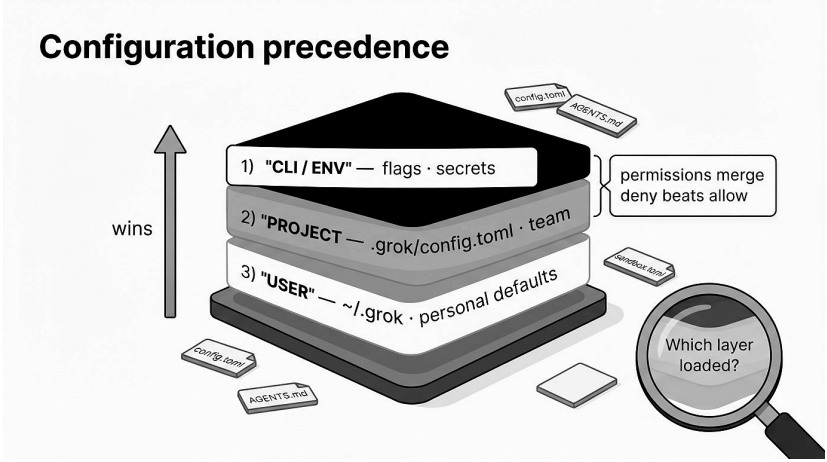


Abbildung 1: CLI/Env schlägt Project schlägt User — Deny und harte Gates gewinnen.

14.2 Wo liegt was?

Config greift nicht? Zuerst wissen, *welche Datei* und *welche Schicht* – `grok inspect` spart Rate-Orakel.

Pfad	Rolle
`~/.grok/config.toml`	Haupt-User-Config
`~/.grok/sandbox.toml`	Sandbox-Profile
`~/.grok/auth.json`	Login-Tokens
`~/.grok/mcp_credentials.json`	MCP-OAuth
`~/.grok/trusted_folders.toml`	Folder-Trust
`~/.grok/sessions/`	Sessions (Kap. 12)
`~/.grok/memory/`	Memory (Kap. 12)
`~/.grok/skills/`, `hooks/`, `rules/`	User-Extensions
`<repo>/.grok/config.toml`	Project MCP/Plugins/Permissions
`$GROK_HOME`	Override der Home-Basis
`/etc/grok/…`	System managed (Org)

`grok inspect` / `grok inspect --json` – **erste Anlaufstelle**, was geladen wurde.

14.3 config.toml – die nützlichen Sektionen

Datei nur mit Overrides füllen; Defaults greifen sonst. **Heute typisch setzen:** Model-Default, Permission-Deny, ggf. MCP/Plugins. UI-Flags (`simple_mode`, `vim_mode`) und Experimente nur, wenn du sie brauchst.

```
[cli]
auto_update = true

[models]
default = "grok-4.5"
web_search = "grok-4.5"
# temperature, top_p, max_completion_tokens, max_retries, …
# extra_headers = { "X-Request-Tags" = "team=example" }

[ui]
permission_mode = "ask"
vim_mode = false
simple_mode = true
screen_mode = "fullscreen"          # oder minimal
show_thinking_blocks = true
remember_tool_approvals = false
default_selected_permission = "allow_once"  # sicherere Default-Zeile

[features]
telemetry = false
codebase_indexing = true
remote_fetch = true                 # false in air-gap
lsp_tools = false

[session]
auto_compact_threshold_percent = 85
load_envrc = true

[tools]
```

```
respect_gitignore = false

[memory]
enabled = false

[subagents]
enabled = true
```

UI-Hinweis: `simple_mode` = Prompt-Editor (readline vs. vim-input). `vim_mode` = Scrollback-Navigation. Unabhängig (Kapitel 3).

Permission-Default-Zeile: `default_selected_permission` steuert nur den **ersten** Prompt einer Session; danach sticky. Env-Override: `GROK_DEFAULT_SELECTED_PERMISSION`.

Theming/Pager: separate `pager.toml` / Settings-UI – im Buch bewusst knapp; Details im User-Guide *Configuration* / *Theming*.

14.4 Compat-Schalter (Claude / Cursor)

Grok scannt default Claude- und Cursor-Pfade für Skills, Rules, Hooks, MCPs. Abschalten z. B.:

```
[compat.claude]
skills = false
rules = false
hooks = false
mcps = false

[compat.cursor]
skills = false
rules = false
hooks = false
mcps = false
```

Env-Pendants existieren (`GROK_CLAUDE_SKILLS_ENABLED=false`, …). Wenn „fremde“ Skills/MCP spuken: Compat + `grok inspect`.

14.5 Modelle wählen

```
grok models                  # verfügbare Modelle listen
grok -m grok-build -p "…"
```

TUI: `/model grok-build` (`/m`), Effort `/effort high`, Picker `Ctrl+M` (Scrollback-Fokus).

```
[models]
default = "grok-4.5"
```

Default-Hosted-Modelle brauchen keine Extra-Endpoint-Config – Auth per Login/Key (Kapitel 2). **IDs und Namen driften** mit Releases; `grok models` und Picker sind Source of Truth.

14.6 Custom Models / alternative Endpoints

Für Self-Hosted, Proxy, Drittanbieter in `~/.grok/config.toml`:

```
[model.my-model]
model = "model-id"                          # Wire-ID an die API
base_url = "https://api.example.com/v1"
name = "Display Name"                       # Picker-Label
api_backend = "chat_completions"            # oder responses |
↪  messages
env_key = "MY_MODEL_API_KEY"                # Key aus Env – nicht
↪  api_key in TOML committen
# extra_headers = { … }
```

`api_backend`	Protokoll
`chat_completions` (default)	OpenAI `/v1/chat/completions`
`responses`	OpenAI Responses
`messages`	Anthropic Messages

Provider-Header (z. B. Anthropic `x-api-key`) über `extra_headers`. Keys nicht committen – Env-Referenzen nutzen, wo unterstützt.

```
grok -m my-model -p "ping"
/model my-model
```

Air-gap: `[features] remote_fetch = false`, fester Model-Katalog, eigener `base_url`.

14.7 Skills-, MCP-, Permission-Config (Querverweis)

Thema	Kapitel	Config-Ort
Skills paths/ignore/disabled	8	`[skills]`
MCP servers	9	`[mcp_servers.*]`, `[mcp]`
Permissions	13	`[permission]`
Subagents	5	`[subagents.*]`
Sandbox	13	`sandbox.toml`
Marketplaces	8	`[[marketplace.sources]]`
Hooks in TOML	8	`[[hooks.Event]]`

Vorlage: `vorlagen/config.toml.beispiel.toml`.

14.8 Debug-Workflow: „Config greift nicht“

1. **Welcher Prozess?** TUI vs. `-p` vs. `agent` – Flags unterschiedlich.
2. **`grok inspect`** – was ist geladen (Rules, MCP, Skills, Origin)?
3. **Precedence** – überschreibt CLI/Env deine TOML?
4. **Project vs User** – liegst du im richtigen CWD/Repo?
5. **Trust** – project MCP/hooks still?
6. **Compat** – kommt der Wert aus Claude/Cursor-Datei?
7. **Syntax** – TOML-Validität, falscher Tabellenname.
8. **Reload** – neue Session; MCP-Modal `r`.
9. **`/doctor`** – Terminal/Sandbox/Clipboard.
10. **Minimal repro** – leere Config + ein Override.

```
grok inspect --json | jq 'keys'
# je nach Schema: skills, mcp, instructions filtern
```

14.9 Praxis-Sessions

Session 1 – Default-Modell setzen

```
[models]
default = "grok-4.5"   # oder aktuelle ID aus `grok models`

grok -c
/session-info
```

Session 2 – sicherere Permission-UI

```
[ui]
default_selected_permission = "allow_once"
remember_tool_approvals = false
permission_mode = "ask"
```

Neuen Session-Start: erster Prompt sollte nicht auf „always all sessions“ kleben.

Session 3 – Project Permission vs. User Model

Project: Deny-Liste. User: `default` model. Prüfen: Deny greift im Repo, Modell kommt aus User-Config.

Session 4 – Custom Model Smoke (wenn Endpoint vorhanden)

```
[model.local-proxy]
model = "my-weights"
base_url = "http://127.0.0.1:8000/v1"
name = "Local Proxy"

grok -m local-proxy -p "Reply with pong only."
```

Session 5 – Compat-Noise abschalten

Cursor-Skills nerven → `[compat.cursor] skills = false` → `grok inspect` ohne Cursor-Skills.

Session 6 – Air-gap Flags

```
[features]
remote_fetch = false
telemetry = false
```

14.10 Jetzt selbst machen: Config-Parcours (20 Minuten)

> ☑ **JETZT SELBST MACHEN**
>
> 1. `~/.grok/config.toml` anlegen/backupen.
> 2. Ein UI- und ein Model-Override setzen.
> 3. `grok inspect` vor/nach.
> 4. CLI `-m` überschreibt Default – verifizieren.
> 5. Project `.grok/config.toml` mit einem Deny.
> 6. Debug-Checkliste 14.8 einmal an einem echten „greift nicht"-Fall durchspielen (oder simulieren).

14.11 Häufige Fehler

Symptom	Ursache	Fix
TOML-Änderung wirkungslos	CLI/Env höher	Flags/Env prüfen
Model unbekannt	Tippfehler / Katalog	`grok models`
MCP nur lokal	project untrusted / scope	Trust, list –
Fremde Skills	Compat an	compat.*.skills false
Zwei Wahrheiten	user + project conflict	inspect, eine Quelle
Secret in TOML committed	Klartext	Env, rotieren

14.12 Checkliste

- ☐ Precedence verstanden
- ☐ User-Config versioniert/backup (ohne Secrets)

- ☐ Project nur sharebare Sektionen
- ☐ `grok inspect` im Werkzeugkasten
- ☐ Model-Default bewusst
- ☐ Compat nur so weit wie nötig
- ☐ Custom Endpoints ohne committed Keys

Was du jetzt kannst

- Config-Schichten und Precedence anwenden und debuggen
- `config.toml`-Kernsektionen und Compat-Schalter setzen
- Modelle wählen und Custom-Endpoints anbinden
- Project- vs. User-Scope für Team-Configs trennen

Nächste Schritte

→ Kapitel 15 – Reale Workflows und Use-Cases: die Hebel aus Kapitel 2–14 in durchgängigen Produktions-Szenarien.

Kapitel 15

Reale Workflows und Use-Cases

> ⊙ **KERN**
>
> Hier greifen die Hebel aus Kapitel 2–14 in **durchgängigen Produktions-Szenarien** – von Plan bis PR, von Bugfix bis CI-Bot – zum Nachbauen, nicht zum Anstaunen.

Dieses Kapitel ist das längste im Buch und bewusst **usage-lastig**. Jeder Use-Case folgt demselben Raster:

1. **Situation** – Alltagsschmerz
2. **Ziel** – messbar
3. **Hebel** – welche Grok-Features
4. **Ablauf** – Prompts, Commands, Checks
5. **Definition of Done**
6. **Fallstricke**

Du brauchst nicht alle Szenarien linear. Nimm das, was diese Woche brennt.

15.1 Karte: welcher Use-Case wann?

#	Use-Case	Primär-Hebel
1	Feature von Plan bis PR; Übungsvariante todo-cli	Plan Mode, AGENTS.md, Verify-Skill
2	Bugfix mit Repro	Explore, enger Scope, Tests
3	Review-only (kein Edit)	read-only Tools/Sandbox
4	Refactor / Migration	Plan, Worktree, Subagents
5	Fremdes Repo onboarding	Explore, AGENTS.md bootstrap
6	CI flaky / rot	Headless, Logs, minimal fix
7	PR-Review-Bot	-p, Deny, JSON-Output

#	Use-Case	Primär-Hebel
8	Parallel zwei Tracks	Worktrees, Subagents
9	Security-pass vor Release	Deny, Sandbox, checklists
10	Doku + Changelog	Skills, @-Files
11	Hotfix Produktion	Scope, kein YOLO, human gate
12	Monorepo Package-Arbeit	Nested AGENTS.md, CWD

15.2 Use-Case 1 – Feature von Plan bis PR

Situation: Ticket „CSV-Export für Invoices“. Mehrere sinnvolle API-Shapes.
Ziel: Ein PR mit Tests, ohne Event-Sourcing-Nebenkrieg.
Hebel: `/plan` → Approve → Implementierung → `/verify`-Skill → Commit/PR.

```
cd ~/projects/billing-api
grok

/plan CSV export for invoices.
Constraints: reuse existing streaming helpers in pkg/export;
no new framework; auth same as GET /invoices; tests required.
Out of scope: PDF, email delivery.
```

Review im Plan-Preview: Dateiliste, Verify (`pytest tests/export -q`), kein Redis „nice to have“. Inline-Kommentar falls nötig → a.

```
Implement the approved plan. Small commits mentally: handler,
↪  service, tests.
Stop after tests green. Do not push.

/verify

Draft a PR title and body from the diff. No git push unless I say
↪  so.
```

DoD: Plan freigegeben, Tests grün, PR-Text reviewt, menschlicher Push.
Fallen: Approve ohne Verify; Scope-Creep „gleich noch Excel“.

Buch-Variante: todo-cli (atomare Writes)

Wenn du das Standard-Übungsrepo aus Kapitel 2–4 nutzt, ist derselbe Bogen ohne Billing-API trainierbar. Pfad vom **Buch-Repo-Root** (PDF-only / CWD: siehe Kapitel 2.8):

```
cd beispiele/todo-cli
grok
```

```
/plan Atomare Writes in store.save (temp-Datei, dann replace).
Constraints: JSON-Form behalten; pytest -q grün; kein neues
↪  Framework.
Out of scope: neues CLI-Subcommand, Netzwerk.
```

Nach Approve: implementieren → `/verify` (bzw. Skill im Repo) → Diff lesen → Commit-Text skizzieren, **kein** Push ohne dich.

DoD: Plan freigegeben, `pytest -q` grün, diff reviewt. Ideal als **erste** End-to-End-Übung vor echten Prod-Repos.

15.3 Use-Case 2 – Bugfix mit Repro

Situation: `test_retry_timeout` flaky in CI, lokal selten.
Ziel: Root cause + minimaler Fix + stabiler Test.

```
Explore only first: find all call sites and timing assumptions for
↪  retryPayment.
Change nothing. Return path:line list and 2 hypotheses.
```

Optional: `explore`-Subagent (Kapitel 5).

```
Reproduce flaky test_retry_timeout: run it 20 times or with
↪  --count if supported.
Report failure rate and stack.
Still no product code fix.
```

```
Minimal fix for the root cause you confirmed. Only files listed.
↪  Keep public API.
Then run tests/payments until green twice.
```

DoD: Repro dokumentiert, Fix ≤ geplanter Blast Radius, Test grün ≥2×.
Fallen: „Rewrite retry framework“; Fix ohne Repro.

15.4 Use-Case 3 – Review-only Session

Situation: Fremder PR, du willst Findings, keine Edits.
Ziel: Severity-liste mit file:line.

```
# --tools: interne Tool-IDs (Shell oft run_terminal_cmd).
# --deny: Permission-Prefix Bash(...), siehe Kap. 10 und 13.
grok -p "Review git diff origin/main...HEAD for bugs and security
↪  issues.
Read-only. No edits, no commits, no network installs.
Output: severity-ordered findings with file:line and short fix
↪  hint." \
  --tools "read_file,grep,list_dir,run_terminal_cmd" \
  --deny 'Bash(git commit *)' \
  --deny 'Bash(git push *)' \
  --deny 'Bash(rm *)' \
  --sandbox read-only \
  --always-approve \
  --output-format json \
  --max-turns 25
```

Oder interaktiv mit Mode ask und explizitem „change nothing“.
DoD: JSON/Markdown Findings; Working Tree clean.
Fallen: Always-approve ohne Deny; Agent „hilft“ mit Auto-Fix.

15.5 Use-Case 4 – Refactor / Migration

Situation: Legacy `payments/v1` → `v2` Client. Hoher Blast Radius.
Ziel: Plan + isolierte Implementierung + grüne Suite.

```
/plan Migrate internal callers from payments/v1 to v2 client.
Keep v1 facade deprecated for one release. List all call sites.
Verification: unit + one integration test path.
```

Nach Approve:

```
Spawn general-purpose subagent with isolation=worktree to
↪  implement phase 1
(facade + 3 highest-traffic callers). Parent tree clean.
Report worktree path, diffstat, test results. No apply until I
↪  review.
```

DoD: Phasen im Plan; Worktree grün; Apply/PR bewusst.
Fallen: Big-bang ohne Facade; shared Tree + Parallel-Writer.

15.6 Use-Case 5 – Fremdes Repo onboarding

Situation: Neues Team-Repo, null Kontext.
Ziel: Mentales Modell + Minimal-AGENTS.md.

```
cd /path/to/new-repo && grok

Onboarding explore only. Map: stack, entrypoints, how to
↪  test/lint,
where config lives, obvious dangers (scripts, secrets paths).
Change nothing. Then draft a minimal AGENTS.md content in chat (do
↪  not write yet).

Write AGENTS.md with only: stack bullets, exact test/lint
↪  commands, 4 don'ts.
No essay.

grok inspect
```

DoD: Du kannst Tests starten; AGENTS.md committed oder PR.
Fallen: Agent „modernisiert“ Build-System beim Onboarding.

15.7 Use-Case 6 – CI rot / flaky Pipeline

Situation: `main` rot nach Merge. Log ist 2000 Zeilen.
Ziel: Erste Ursache + Fix-PR oder Revert-Empfehlung.

```
# Log extrahieren
gh run view --log-failed > /tmp/ci-fail.log   # oder Artefakt
↪  speichern

grok -p "CI failed. Log file: /tmp/ci-fail.log
Find root cause. Quote the first real error, not only the last
↪  exit code.
Propose minimal fix OR revert recommendation. Do not push." \
```

```
  --always-approve \
  --cwd "$(pwd)" \
  --deny 'Bash(git push *)' \
  --deny 'Bash(git push --force *)' \
  --max-turns 30 \
  --output-format json | jq -r '.text'
```

Policy dieses Jobs: Writes und Commits sind erlaubt (minimaler Fix im Tree); **Push/Deploy bleiben blockiert**. Für Review-only siehe UC7 / Kap. 10 (Tool-Allowlist ohne Shell-Write).
Interaktiv: `@/tmp/ci-fail.log` anhängen statt riesigem Paste (Kontext!).
DoD: Eine Hypothese mit Evidenz; nächster Schritt klar.
Fallen: Ganzes Log in den Prompt; YOLO ohne Deny; unbemerktes Force-Push-Pattern.

15.8 Use-Case 7 – PR-Review-Bot (Headless)

Situation: Jeder PR soll automatische Erst-Findings bekommen.
Ziel: Comment/Summary ohne Write auf main.
Siehe auch Kapitel 10 (GHA-Gerüst). Kernprompt:

```
You are a PR review bot. Diff range: origin/main...HEAD.
Focus: correctness, security, missing tests. Ignore style nits.
Output markdown for PR comment. Read-only. No git write.
```

Policy-Bundle:

- `XAI_API_KEY` Secret
- `--always-approve` + Deny push/rm
- `--sandbox read-only` wo möglich
- `--max-turns`
- GitHub token: `pull-requests: write` nur wenn Comment gewollt, sonst summary artifact

DoD: Job grün/rot nach eurer Policy; keine Secrets im Log.
Fallen: Bot mit Write-Permissions auf Default-Branch.

15.9 Use-Case 8 – Zwei parallele Tracks

Situation: Feature-Flag-Doku und Bugfix gleichzeitig, Deadline.
Ziel: Keine Diff-Vermischung.

```
Track A worktree: docs-only feature flag documentation + sample
↪  config.
Track B worktree: fix formatDate timezone + unit test.
Disjoint files. No apply until both summaries reviewed.
```

Oder zwei Terminals:

```
grok --worktree=docs-flag "…"
grok --worktree=fix-date "…"
```

DoD: Zwei Branches/PRs oder zwei clean applies.
Fallen: Beide Tracks an `package.json`.

15.10 Use-Case 9 – Security-Pass vor Release

Situation: Release-Kandidat, Security-Checkliste.
Ziel: Findings + bestätigte Non-Findings.

```
Security pass, read-only:
1) secret patterns (.env committed?, hardcoded keys)
2) authz on new endpoints
3) dependency scripts / postinstall risks
4) dangerous shell in CI configs
Do not fix yet. Severity table.
```

```
# optional hardening
grok -p "…" --sandbox strict --always-approve \
  --disallowed-tools "search_replace,web_fetch"
```

Danach separate Session für Fixes mit Plan Mode bei API-Änderungen.
DoD: Tabelle; Owner pro Finding; kein „sieht gut aus“ ohne Scan.
Fallen: Nur Dependency-CVE-Scan, kein Authz.

15.11 Use-Case 10 – Changelog und Release Notes

Situation: Tag steht bevor, Commits unruhig.
Ziel: Nutzer-verständliche Notes.

```
/create-skill   # optional: release-notes skill mit
↪  disable-model-invocation

Draft release notes from commits since v1.4.0.
Group: Features, Fixes, Breaking. Audience: API consumers.
No internal ticket IDs. Output markdown only. Change no code.

git log v1.4.0..HEAD --oneline
```

DoD: Notes reviewt; Breaking klar markiert.
Fallen: Agent erfindet Features, die nicht im Diff sind – gegen `git log` halten.

15.12 Use-Case 11 – Hotfix Produktion

Situation: Sev-2, fehlerhafte Null-Check in Parser, Fix muss raus.
Ziel: Minimaler Patch, maximale Kontrolle.
Regeln (hart):

- Kein Always-approve
- Kein MCP mit Prod-Credentials
- Ein File wenn möglich
- Mensch merged

```
HOTFIX only: Null check in parseInvoiceLine for empty description.
Reproduce with a unit test first (failing), then fix, then test
↪  green.
Do not refactor neighbors. Do not touch deploy scripts.
Show diff and test output. Wait for my explicit ok before any git
↪  commit.
```

DoD: Test + Fix; menschlicher Commit/Deploy-Runbook.
Fallen: „Solange wir schon da sind…“; YOLO in Prod-Checkout.

15.13 Use-Case 12 – Monorepo Package-Fokus

Situation: `packages/frontend` und `packages/backend` mit anderen Konventionen.
Ziel: Agent hält Package-Grenzen ein.

```
cd packages/backend && grok
```

```
You are in backend package. Respect nested AGENTS.md.
Add endpoint GET /healthz following existing handlers.
No React, no frontend package edits. Tests for backend only.
```

Root-AGENTS.md: Workspace-Commands. Nested: Stack-spezifisch (Kapitel 7).
DoD: Diff nur unter `packages/backend`.
Fallen: `grok` im Monorepo-Root ohne Scope → Cross-Package-Edits.

15.14 Meta-Workflow: Explore → Plan → Code → Verify → PR

Zusammensetzung aus den Use-Cases:

```
1 Explore   „Ändere nichts“ / explore-Subagent
2 Plan      /plan + Constraints + Verify
3 Approve   Preview a/s/c/q
4 Code      eng, ideal Worktree bei Risiko
5 Verify    Skill oder exakte Commands aus AGENTS.md
6 Review    frische Session oder zweiter Mensch
7 PR        Text vom Agent, Push vom Menschen (Default)
```

Persistenz: Entscheidungen in Ticket/PR/AGENTS.md – nicht nur im Chat (Kapitel 12).

15.15 Jetzt selbst machen: Ein Use-Case end-to-end

> ☑ **JETZT SELBST MACHEN**
>
> 1. Wähle Use-Case 1 **oder** 2 an einem echten Side-Projekt.
> 2. Schreibe den Prompt **vor** dem Start auf (Ziel, Nicht-Ziel, Verify).

3. Fahre den Ablauf inkl. Diff-Review.
4. Notiere: Welcher Hebel hat am meisten Rework verhindert?
5. Übernimm eine Lücke in AGENTS.md oder einen Skill.

15.16 Checkliste vor dem nächsten Agent-Tag

- ☐ AGENTS.md mit echten Commands
- ☐ Deny-Minimum committed
- ☐ Plan Mode bei Ambiguität
- ☐ Verify vor „fertig“
- ☐ Diff gelesen
- ☐ Session nicht als Müllhalde missbraucht
- ☐ Secrets raus aus Prompt und Memory

Was du jetzt kannst

- Zwölf Produktions-Szenarien auf Grok-Hebel abbilden
- Einen Feature- und einen Fix-Flow ohne Scope-Chaos fahren
- CI/Bot- und Review-only-Muster absichern
- Monorepo- und Hotfix-Disziplin anwenden

Nächste Schritte

→ Kapitel 16 – Best Practices, Tipps und Troubleshooting: Prompt-Engineering, Symptom→Fix und Team-Rituale.

Kapitel 16

Best Practices, Tipps und Troubleshooting

> ⊙ **KERN**
>
> Die meisten „Grok spinnt“-Tickets sind **Kontext-, Permission-, Trust-, Scope- oder Erwartungsprobleme** – gelöst mit Systematik, nicht mit lauteren Prompts.

Stand der Fakten im Buch: **Juli–Aug 2026** (CLI **0.2.x**); UI-Labels und einzelne Commands können driften.

Schnellpfad Symptom → Fix: 16.6.

Prompt-Rezepte: 16.2.

16.1 Best Practices – kompakt

Alltag: Ticket, Diff, CI-Fail – die folgenden Blöcke decken den Großteil der Fälle. Details und Kanon sitzen in den Fachkapiteln; hier die kompakte Reprise.

Prompts

1. **Ziel + Nicht-Ziel + Verify** in jedem nicht-trivialen Auftrag
2. Dateipfade nennen, wenn bekannt
3. „Ändere nichts“ bei Explore explizit
4. Fehler: komplette Meldung, nicht „geht nicht“
5. Ein Thema pro Prompt-Block; Queue bewusst nutzen

Repo

1. `AGENTS.md` kurz, Commands exakt
2. Deny für Secrets und Force-Push von Tag 1
3. Skills = Verfahren, Rules = Normen, Hooks = Zwang
4. Kleine PRs – auch wenn der Agent 40 Dateien anbietet

Sessions

1. Ein Thema pro Session
2. `/context` lesen, bevor du „lauter“ wirst
3. `/plan` vor multi-file Ambiguität
4. `/compact` **mit Hinweis**, nicht blind
5. Entscheidungen in Dateien/PR persistieren
6. Vor Merge: frische Review-Session oder Mensch

Sicherheit

1. Kein Always-approve auf Alltagsrechnern mit Prod-Zugang ohne Deny
2. MCP wie Dependencies reviewen
3. Folder-Trust nur für bekannte Repos
4. CI least privilege + API-Key als Secret

Parallelität

1. Explore parallel, Writes serial oder Worktrees
2. Parent orchestriert Integration-Verify
3. Plan-Parent sperrt Kind-Edits **nicht** automatisch

16.2 Prompt-Engineering für Grok Build

Grok Build ist ein **Agent mit Tools**, kein reiner Chat. Gute Prompts steuern den **Harness**. Englisch in Tool-Steuerung und DoD ist oft robuster – Formulierungen unten sind Copy-Paste, kein Stilbruch zum deutschsprachigen Buch.

Schablone (Copy-Paste)

```
Ziel: <was am Ende wahr sein soll>
Nicht-Ziel: <explizit out of scope>
Kontext: <Ticket/Branch/Constraints>
Dateien: <pfade wenn bekannt>
Vorgehen: <explore | plan | implement>
```

```
Verify: <exakter Command>
Stop: <wann warten / kein push>
```

Vorher / Nachher

Schwach:

```
Mach die App besser und fix bugs.
```

Stark:

```
Ziel: flaky test_retry_timeout stabil grün (2×).
Nicht-Ziel: kein Retry-Framework-Rewrite, keine API-Änderung.
Vorgehen: erst Repro + Ursache, dann minimaler Fix.
Verify: pytest tests/payments/test_retry.py -q
Stop: kein commit/push ohne mein ok.
```

Explore vs. Implement

Intent	Formulierung
Nur verstehen	„Ändere nichts. Keine Edits.“
Plan	`/plan …` + Constraints
Implement	„Implement approved plan…“ + Verify
Review	„Read-only findings table…“

Tool-Steuerung in Worten

```
Use explore subagent for research; parent only summarizes.
Do not spawn write-capable subagents while planning.
Prefer @path over wide repo scans when path is known.
First search_tool before any MCP use_tool.
```

Compact-Hints

```
/compact Behalte freigegebenen Plan, geänderte Dateien, letzten
↪  Test-Fail.
Verwirf Explore-Umwege und gescheiterte Zwischenversuche.
```

Anti-Patterns

- „Sei gründlich“ ohne Scope → endloses Lesen
- Widersprüchliche Regeln in Prompt vs. AGENTS.md
- Secrets im Prompt „nur kurz“
- Fünf Features in einer Session
- Approve ohne Diff

16.3 Session-Hygiene (Ritual)

Zeitpunkt	Aktion
Start	CWD prüfen, `/session-info`, Mode lesen
Vor großem Feature	`/plan`
Bei Drift	`/context` → compact mit Hint oder `/new`
Nach Meilenstein	Entscheidung in PR/AGENTS.md
Vor Merge	Review-Session oder zweiter Mensch
Ende	`/quit`; optional `/flush` wenn Memory an

Faustregel: Wenn du denselben Korrekturprompt zum dritten Mal tippst, ist das Fenster oder der Plan kaputt – nicht „das Modell dumm“.

16.4 Team-Rituale

1. **AGENTS.md im Onboarding** – erster PR jedes Neuen.
2. **Deny-Baseline** committed.
3. **Plan first** Label auf architektonischen Tickets.
4. **Verify-Skill** statt „tests liefen irgendwie“.
5. **Human push** als Default; Bot nur comment/summary.
6. **Quarterly rules review** – tote Commands löschen.
7. **Incident blameless:** War es YOLO, Trust, MCP, fehlender Scope?

16.5 Systematisches Debuggen

```
1. Symptom benennen (hängt / falscher Diff / ignoriert Regeln /
↪  Auth)
2. Surface? TUI / -p / agent
3. Mode? ask / auto / plan / always-approve (ggf. acceptEdits /
↪  dontAsk)
4. grok inspect · /context · /session-info · /doctor
5. CWD und geladene AGENTS.md/Rules
6. Trust / MCP enabled?
7. Deny/Allow Patterns
8. Minimal repro mit engem Prompt
```

Nicht parallel zehn Hebel drehen – eine Variable pro Versuch.

16.6 Symptom → Fix

Symptom	Erste Checks	Fix
`grok: command not found`	PATH, neue Shell	Installer, `which grok`
Login-Schleife	`auth.json`, Key-Vorrang	`grok logout && grok login`
Key greift nicht	Session-Token aktiv	logout, nur `XAI_API_KEY`
TUI Keys tot	Terminal/tmux	`/doctor`, `Ctrl+X`
Hängt in CI	Permission ask	`--always-approve` + Deny
Schreibt trotz „nur lesen“	weicher Prompt / Subagent	explizit + explore/read-only
Ignoriert AGENTS.md	CWD, gitignore, Länge	`grok inspect`, kürzen
Plan Mode schreibt Repo	Kind-Subagent	explore/plan only
Parallel-Chaos	shared tree	worktrees
Kontext drift	langes Fenster	compact+hint / new
Rewind-Schock	uncommitted	Git zuerst
MCP fehlt	disabled, trust, doctor	`/mcps`, trust
Halluzinierte MCP-Tools	kein search_tool	Prompt erzwingen

Symptom	Erste Checks	Fix
YOLO „ignoriert Deny“	Pattern falsch	Syntax, `deny` testen
Sandbox start fail	hooks symlink	`/doctor`, profile
Zu teuer	MCP/tools/noise	engere Tools, new session
Skill erscheint nicht	name/path/disabled	`grok inspect`
Falsches Modell	default/CLI	`/model`, `grok models`
Config „tot“	Precedence	Kap. 14 Debug-Flow
Force-push im Diff	fehlende Deny	Deny + Review
`.env` im Kontext	Read erlaubt	Deny `Read(**/.env)` (+ absolute Forms)

16.7 Häufige Missverständnisse

Glaube	Realität
Always-approve = alles erlaubt	Deny/Hooks greifen weiter
Plan Mode sperrt Subagents	Nein – Kind kann schreiben
Compact speichert alles Wichtige	Verlustbehaftet – Hint + Dateien
Memory ersetzt AGENTS.md	Nein – Team vs. episodisch
Mehr Tokens = bessere Fixes	Oft mehr Drift
IDE braucht andere Regeln	Dieselbe Engine, andere UI

16.8 Praxis-Sessions (Meta)

Session A – Prompt-Refactor

Nimm einen schwachen Prompt aus eurem Chat-Verlauf, baue die Schablone 16.2, fahre beide, vergleiche Diff-Größe und Turns.

Session B – Symptom-Drill

Simuliere absichtlich: leeres CWD, YOLO ohne Deny, vages „fix everything“. Protokolliere, was schiefgeht – dann die Fixes aus 16.6.

Session C – Team-Baseline-PR

Ein PR nur mit: AGENTS.md-Kern, Deny-Minimum, optional verify-Skill. Kein Feature-Code.

16.9 Jetzt selbst machen

> ☑ **JETZT SELBST MACHEN**
>
> 1. Pinne die Schablone 16.2 neben den Monitor (oder als Snippet).
> 2. Laufe einmal `/context` + `/session-info` in einer echten Session.
> 3. Trage euer Top-3-Symptom aus 16.6 in ein Team-Doc.
> 4. Einen Deny und einen AGENTS.md-Command diese Woche schärfen.

16.10 Checkliste „gesunder Agent-Alltag“

- ☐ Scope in jedem Auftrag
- ☐ Mode bewusst
- ☐ Diff gelesen
- ☐ Verify gelaufen
- ☐ Kontext nicht als Deponie
- ☐ Secrets raus
- ☐ Human ownership vor Prod

Was du jetzt kannst

- Stabile Prompt-Schablonen und Session-Rituale anwenden
- Symptome systematisch auf Harness-Ursachen mappen
- Team-Baseline und Debug-Reihenfolge fahren

Nächste Schritte

→ Kapitel 17 – Vergleich mit Claude Code, Codex und Cursor: ehrlich, tabellarisch, wann welches Werkzeug.

Kapitel 17

Vergleich mit Claude Code, Codex und Cursor

> ⊙ **KERN**
>
> Alle vier sind **agentische Coding-Werkzeuge** – sie unterscheiden sich in Surface, Policy-Modell, Ökosystem und Team-Fit; „das beste“ ist das, das **ihr steuern und reviewen** könnt.

Vergleiche driften mit Releases (Stand **Juli–Aug 2026**, Grok CLI **0.2.x**). Nutze die **Achsen**, nicht Marketing-Slogans. Primärquellen: jeweilige Produktdoku; dieses Kapitel orientiert für Entscheider und Power-User.

17.1 Die Vergleichsachsen

Achse	Frage
Surface	TUI, IDE-first, Desktop, Web, Headless, ACP?
Rules	Wie persistiert ihr Team-Normen?
Permissions	Ask/Allow/Deny, Sandbox, Trust?
Plan	Erst Ansatz freigeben?
Parallel	Subagents, Worktrees?
Extensibility	Skills, Hooks, MCP, Plugins?
Automation	CI/Headless/Agent-Server?
Compat	Kann man bestehende Claude/Cursor-Dateien mitnutzen?
Vendor/Auth	Welches Konto, welche Datenresidenz?

17.2 Kurzportraits

Grok Build (`grok`)

- Terminal-TUI first (xAI; Auth oft SpaceXAI-OAuth), Headless `-p`, ACP `grok agent`
- Regeln: `AGENTS.md`, `.grok/rules/`; Compat zu CLAUDE.md / Cursor über `[compat.*]` und `grok inspect` (Kapitel 14)
- Plan Mode mit Preview/Kommentaren, Subagents, Worktrees
- Skills, Hooks, Plugins/Marketplaces, MCP
- Config: `config.toml`, Sandbox-Profile, Folder-Trust

Stärke: breiter Harness im Terminal + Automation + Compat.
Passend wenn: Team CLI-affin ist, CI/Agent-Server will, multi-tool Ökosystem braucht.

Claude Code (`claude`)

- Anthropic; Terminal + IDE-Extensions + Desktop/Web Surfaces
- Regeln: `CLAUDE.md`, `.claude/settings.json`, Rules, Hooks
- Starkes Permission-Modell (allow/deny/ask), Enterprise-Policies
- Plan-/Workflow-Features produktseitig eigen; Skills/MCP/Hooks etabliert

Stärke: ausgereiftes Policy- und Team-Storytelling, multi-surface.
Passend wenn: Anthropic-Stack, bestehende CLAUDE.md-Kultur, Enterprise-Controls.

Codex (OpenAI)

- OpenAI-Coding-Agent (CLI/Cloud-Varianten je nach Produktstand)
- Fokus oft auf Repo-Tasks und OpenAI-Modellfamilie
- Policy- und Integrationsdetails stark versionsabhängig – vor Kauf/Rollout aktuelle Doku lesen

Stärke: enge Anbindung an OpenAI-Modelle und Ökosystem.
Passend wenn: Organisation bereits OpenAI-first arbeitet und Codex freigeschaltet ist.

Cursor

- IDE-first (Fork-Ökosystem um VS Code), starker Editor-Workflow
- Rules/Skills/MCP im Cursor-Modell; Inline + Agent-Chat
- Sehr stark bei Selection-Kontext, Multi-File im Editor, UX für viele Devs

Stärke: niedrigste Reibung für IDE-allgemeine Entwickler.
Passend wenn: Team primär im Editor lebt und Diffs dort reviewt.

17.3 Feature-Matrix (orientierend)

Legende: ● stark / etabliert · ◐ vorhanden / anders gelöst · ○ schwach oder nicht Fokus · ? stark versionsabhängig
Codex-Spalte: bewusst unvollständig (?) – OpenAI-Codex-Surfaces und Policies driften schnell; kein Pseudo-Vergleich. Vor Rollout Primärdoku lesen (Stand 2026H2).

Fähigkeit	Grok Build	Claude Code	Codex	Cursor
Interaktive Terminal-TUI	●	●	◐/?	○ (IDE)
IDE-native UX	◐ (ACP)	●	◐/?	●
Headless / CI	●	●	◐/?	◐
Agent-Protokoll/Server	● ACP	◐	?	◐
Project rules files	● AGENTS.md	● CLAUDE.md	?	● Rules
Allow/Deny Permissions	●	●	?	◐
OS Sandbox profiles	●	◐	?	◐
Plan-then-code UX	●	●	?	◐
Subagents / parallel	●	●	?	◐
Git worktree isolation	●	◐	?	◐
Skills / slash workflows	●	●	?	●
Hooks	●	●	?	●
MCP	●	●	?	●

Fähigkeit	Grok Build	Claude Code	Codex	Cursor
Plugins / marketplaces	●	●	?	◐
Cross-tool file compat	● liest Claude/Cursor	◐	?	○

Lies die Matrix so: Sie hilft bei der **Architekturwahl**, ersetzt aber keinen PoC mit eurem Repo und eurer Security-Policy.

17.4 Regeln und Kompatibilität

Bedürfnis	Praxis
Ein Repo, mehrere Agents	Kurze, konsistente Kernregeln; Grok liest oft `CLAUDE.md` + Cursor-Pfade mit
Eine Source of Truth	Eine Datei „gewinnt“, andere verlinken – keine widersprüchlichen Package-Manager-Sätze
Migration Claude → Grok	`CLAUDE.md` behalten, schrittweise `AGENTS.md` + `.grok/`
Migration Cursor → Grok	Rules/Skills mitnehmen wo Compat greift; `grok inspect` prüfen
Strikt Grok-only	Compat-Flags aus (Kapitel 14)

Schwesterprojekt-Stil und Lessons aus dem Claude-Buch: `../ki-claude` – inhaltlich parallel, produktspezifisch getrennt (`ADAPTIONEN.md`).

17.5 Wann welches Tool?

Situation	Tendenz
Team lebt in CLI, CI, Worktrees	**Grok Build** oder **Claude Code**
Team lebt in VS Code/Cursor-UX	**Cursor** (+ optional CLI-Agent parallel)

Situation	Tendenz
Anthropic Enterprise + bestehende CLAUDE.md	**Claude Code**
OpenAI-Standardisierung	**Codex** evaluieren
Multi-Agent-Strategie	Rules teilen; Surfaces wählen; **ein** Bot-Policy-Modell
Untrusted Code Review	Tool mit starkem read-only/Sandbox (Grok: `--sandbox read-only`)
Maximale IDE-Diff-Integration	Cursor oder Claude/IDE-Extensions

Hybrid ist normal: Cursor zum Tippen, Grok/Claude für schwere multi-file + CI. Teuer wird Hybrid erst bei **widersprüchlichen Rules** und doppelten YOLO-Politiken.

17.6 Migrations-Skizzen

Von Claude Code zu Grok Build

1. `claude/CLAUDE.md` behalten.
2. `grok` installieren, `grok inspect` – was wird geladen?
3. Deny/Permissions in `.grok/config.toml` spiegeln.
4. Skills nach `.grok/skills/` oder Compat nutzen.
5. CI-Job parallel mit `grok -p` PoC.
6. Team-Schulung Plan Mode + Diff-Ritual (identisch im Geist).

Von Cursor zu Grok Build

1. Export/Verständnis der Cursor Rules.
2. Compat an, `inspect`, Noise abschalten.
3. Schwere Tasks in TUI üben (Permissions sichtbar).
4. Optional ACP-Anbindung in der IDE.

Von Grok zu „nur IDE“

1. AGENTS.md behalten.

2. Automation (CI) nicht verlieren – Headless bleibt oft der CLI-Agent.

17.7 Ehrliche Grenzen aller Agents

Unabhängig vom Vendor:

- Kein Ersatz für Ownership und Review
- Kontextfenster endlich
- Prompt-Injection über untrusted Content
- MCP = erweiterte Angriffsfläche
- „Grün“ ≠ production-ready

Wer das verinnerlicht, wechselt Tools ohne Kult.

17.8 Praxis-Session: Decision Record

```
Schreibe ein Architecture Decision Record (ADR) für unser Team:
Welchen Coding-Agenten standardisieren wir 2026H2?
Kriterien: CLI, CI, IDE, Permissions, MCP, Kostenmodell, Skills.
Empfehlung + Hybrid-Option. Keine Marketing-Floskeln.
Ändere keine Repo-Dateien außer docs/adr/XXXX-coding-agent.md wenn
↪  ich zustimme.
```

Zuerst nur im Chat reviewen, dann schreiben lassen.

17.9 Checkliste Tool-Auswahl

- ☐ Surfaces gelistet, die das Team wirklich nutzt
- ☐ PoC auf **eurem** Repo (nicht Demo-ToDo)
- ☐ Permission/Deny-Story getestet
- ☐ CI-Story skizziert
- ☐ Rules-Kompatibilität geprüft
- ☐ Security/Trust abgenommen
- ☐ Exit-Kriterien für den PoC (2 Wochen)

Was du jetzt kannst

- Coding-Agents entlang stabiler Achsen vergleichen
- Grok Build gegen Claude Code, Codex und Cursor positionieren
- Migrations- und Hybrid-Strategien ohne Regel-Chaos skizzieren

Nächste Schritte

→ Kapitel 18 – Anhang Schnellreferenz: Cheatsheets für den Alltag im echten Repo.

Kapitel 18

Anhang – Schnellreferenz

> ⊙ **KERN**
>
> Spickzettel zum Nachschlagen beim Arbeiten – nicht Ersatz für die Kapitel 0–17.

So nutzen: Im echten Repo offen halten. Befehl/Pfad suchen → bei Unklarheit ins Kapitel springen. UI-Labels können mit Releases driften (Stand **Juli–Aug 2026**, CLI **0.2.x**); `grok --help`, `/` und User-Guide bleiben autoritativ.

18.1 Install & Auth

Plattform	Befehl
macOS/Linux/WSL/Git Bash	`curl -fsSL https://x.ai/cli/install.sh \{}| bash`
Version pinnen	… `\{}| bash -s 0.2.0` (Beispiel-Pin; `grok --version` / Release-Notes)
Windows PowerShell	`irm https://x.ai/cli/install.ps1 \{}| iex`
PS + Version	`$env:GROK_VERSION=„0.2.0“; irm https://x.ai/cli/install.ps1 \{}| iex`

```
grok --version
grok update
grok doctor
grok doctor --json

grok login                  # default OAuth
grok login --device-auth    # headless/remote
grok logout
export XAI_API_KEY="xai-..."
```

Credentials: `~/.grok/auth.json` (owner-only). Details: Kapitel 2.

18.2 Start & Sessions

```
grok
grok "initial prompt"
grok --cwd ~/projects/app
grok -c                          # continue last
grok -r <session-id-or-title>
grok -m grok-build
grok --worktree=feat "…"         # = verhindert Prompt-als-Name
grok -w --ref main "…"
grok --yolo / --always-approve
grok --sandbox workspace
grok --rules "…"

grok sessions list
grok sessions search "keyword"
grok sessions delete <id>
grok inspect
grok models
```

TUI	Wirkung
`/new` `/clear`	neue Session
`/resume`	Session-Picker
`/rename`	Titel
`/fork [--worktree\{}|--no-worktree]`	verzweigen
`/rewind` (`/undo`)	Turn + Files zurück
`/compact [hint]`	Kontext komprimieren
`/context`	Fenster-Breakdown
`/session-info` `/status`	Meta
`/quit` `/exit`	beenden
`/home`	Welcome

18.3 Modes & Permissions

Hebel	Wirkung
`Shift+Tab`	Normal → Plan → Always-approve
`Ctrl+O` / `/always-approve`	YOLO toggeln
`/auto`	Auto-Mode (Feature)
`/plan` `/view-plan`	Plan Mode
`--allow` / `--deny`	Rules
`[permission]` in config	Allow/Deny Listen

Pipeline: Hooks → **Deny > Ask > Allow** → remembered grants → built-in readonly → Mode.
Deny gewinnt immer – auch unter Always-approve.

```
grok -p "…" --always-approve \
  --deny 'Bash(rm -rf *)' \
  --deny 'Bash(git push --force *)' \
  --allow 'Bash(npm test *)'
```

Sandbox: `--sandbox off|workspace|devbox|read-only|strict` (Built-ins) oder **Profilname** aus `sandbox.toml` (z. B. `--sandbox project`).
Trust: `/hooks-trust`, `--trust`, `~/.grok/trusted_folders.toml`.
Kapitel 4, 13.

18.4 Plan Mode Approval (Preview)

Key	Aktion
`a`	Approve (+ comments)
`s`	Request changes
`c`	Comment on lines
`y`	Plan in Clipboard
`q`	Quit plan
`Tab`	Preview ↔ Prompt

Plan-Datei: `~/.grok/sessions/<encoded-cwd>/<id>/plan.md`.

18.5 TUI-Shortcuts (Alltag)

Key	Aktion
`Tab`	Prompt ↔ Scrollback
`Enter`	senden
`Esc`	Turn cancel (Draft bleibt)*
`Ctrl+C`	clear/cancel-Semantik
`Ctrl+P` / `?`	Palette
`Ctrl+M`	Model-Picker **oder** Multiline
`Ctrl+S`	Sessions
`Ctrl+T`	Todos
`Ctrl+G`	Tasks-Pane
`Ctrl+B`	Command backgrounden
`Ctrl+L` / `/plugins`	Extensions-Modal*
`Ctrl+.` / `Ctrl+X`	Shortcuts-Hilfe
`F2`	Settings
`@`	Datei anhängen
`!`	Shell-Mode
`↑`	Prompt-History

* Fullscreen+Vim: Esc cancelt oft nicht – `Ctrl+C`. Auf VS-Code-Familie ist `Ctrl+L` oft **Interject** – Extensions dort über `/plugins` / `/hooks`. Details: Kapitel 3.

18.6 Slash-Commands (Auswahl)

Command	Zweck
`/plan` `/view-plan`	Plan Mode
`/model` `/m` `/effort`	Modell
`/always-approve` `/auto`	Permissions
`/compact` `/context`	Kontext
`/memory` `/flush` `/dream` `/remember`	Memory (default oft aus; `/flush`/`/dream`/`/memory` brauchen Opt-in – Kap. 12; `/remember` oft auch ohne)
`/hooks` `/plugins` `/marketplace` `/skills` `/mcps`	Extensions
`/config-agents` `/personas`	Agents
`/doctor`	Diagnose
`/goal` `/workflow` `/workflows`	Goals/Workflows
`/dashboard`	Agent-Dashboard (Fullscreen)
`/deep-research`	Background-Research-Workflow
`/loop`	wiederkehrend
`/btw`	Side-Question
`/create-skill`	Skill bauen
`/login` `/logout` `/usage` `/privacy`	Account
`/import-claude`	Claude-Settings importieren
`/imagine` `/imagine-video`	Media Generation
`/settings` `/theme`	UI
`/copy` `/export`	Export
`/minimal` `/fullscreen` `/vim-mode`	Render/Nav
`/tutorial` `/docs`	Hilfe

Skills mit `user-invocable: true` → `/skill-name`. Qualifiziert: `/local:name`, `/repo:name`, `/user:name`, oder `/<plugin-name>:skill`.

18.7 Headless & ACP

```
# Headless
grok -p "prompt"
grok -p "…" --output-format
↪  plain|json|streaming-json|streaming-messages-json
# partial deltas nur mit streaming-messages-json:
#   --include-partial-messages
grok -p "…" --tools "read_file,grep,list_dir"
grok -p "…" --disallowed-tools "Agent,web_search"
grok -p "…" --max-turns 30
grok -p --prompt-file ./p.txt

# ACP (Deny/Allow: [permission] in config – nicht als
↪  agent-CLI-Flag)
grok agent --always-approve stdio
grok agent --always-approve serve --bind 127.0.0.1:2419 --secret
↪  TOKEN
```

Kapitel 10–11.

18.8 Subagents & Worktrees

Typ	Edits
`general-purpose`	ja
`explore`	nein
`plan`	nein

Capability: `read-only` | `read-write` | `execute` | `all`.
Isolation: `none` | `worktree`.
TUI: `Ctrl+G` Tasks.
Disable: `GROK_SUBAGENTS=0` / `[subagents] enabled=false`.

```
grok --worktree=label "…"
grok -w -r <session-id>
/fork --worktree …
```

Kapitel 5–6.

18.9 Pfade-Cheatsheet

Was	Wo
User config	`~/.grok/config.toml`
Project config	`.grok/config.toml`
Sandbox	`~/.grok/sandbox.toml`, `.grok/sandbox.toml`
Auth	`~/.grok/auth.json`
MCP OAuth	`~/.grok/mcp_credentials.json`
Trust	`~/.grok/trusted_folders.toml`
Sessions	`~/.grok/sessions/<cwd>/<id>/`
Memory	`~/.grok/memory/`
User skills	`~/.grok/skills/`
Project skills	`.grok/skills/`
Hooks	`~/.grok/hooks/`, `.grok/hooks/`
Rules	`.grok/rules/`, `AGENTS.md`
Global rules	`~/.grok/rules/`
`$GROK_HOME`	Basis-Override

Precedence: CLI > Env > config.toml > managed/requirements > defaults.

18.10 MCP (kurz)

```
grok mcp list
grok mcp add filesystem -- npx -y
↪  @modelcontextprotocol/server-filesystem ./docs
grok mcp add --transport http name https://… --header
↪  "Authorization: Bearer ${TOKEN}"
grok mcp enable name
grok mcp disable name
```

```
grok mcp doctor
grok mcp remove name
```

```
[mcp_servers.name]
command = "npx"
args = ["-y", "…"]
# oder url = "https://…"
enabled = true
```

Tools: `server__tool` via `search_tool` / `use_tool`. Modal: `/mcps`. Kapitel 9.

18.11 Prompt-Schablone

```
Ziel:
Nicht-Ziel:
Dateien:
Vorgehen: explore | plan | implement
Verify: <command>
Stop: kein push / warte auf ok
```

Compact:

```
/compact Behalte <Entscheidungen>. Verwirf <Explore-Müll>.
```

Kapitel 16.

18.12 Symptom → Fix (Mini)

Symptom	Fix
command not found	PATH / reinstall
CI hängt	`--always-approve` + deny
Regeln greifen nicht	`grok inspect`, CWD
Kontext drift	`/compact` hint / `/new`
MCP tot	doctor, trust, enable
`.env` im Kontext	Deny `Read(**/.env)` (+ absolute Forms)

Symptom	Fix
Keys tot in tmux	`/doctor`, `Ctrl+X`

Voll: Kapitel 16.6.

18.13 Vorlagen & Beispiele

Pfad	Inhalt
`vorlagen/AGENTS.md.minimal.md`	Minimal-Regeln
`vorlagen/AGENTS.md.team.md`	Team-Baseline
`vorlagen/config.toml.beispiel.toml`	Config-Skizze (User/Project)
`vorlagen/config.permissions.toml`	Permissions-Minimum
`vorlagen/config.mcp.toml`	MCP-Snippets
`vorlagen/PLAN.md.skeleton.md`	Plan-Skeleton
`vorlagen/hooks/`, `vorlagen/skills/`, `vorlagen/plugins/`	Hooks, Verify, Plugin-Baseline
`beispiele/`	Mitmach-Index
`ADAPTIONEN.md`	Schwesterprojekt-Mapping

18.14 Glossar (Mini)

Einstieg auch in Kapitel 1.11.

Begriff	Bedeutung
Harness	Tools, Permissions, Memory, Loop ums Modell
Tool-Call	Einzelner freigegebener Handlungsschritt (read/shell/edit/…)
Session	Ein Arbeitsgespräch inkl. Kontext und History
CWD	Working Directory – Startordner von `grok`

Begriff	Bedeutung
Diff	Änderungsansicht (+/–); vor Allow und vor Commit lesen
Plan Mode	Read-only Planung + Freigabe-UI
Compact	Kontext komprimieren (mit optionalem Keep-Hinweis)
Deny	Harte Block-Regel; gewinnt auch unter Always-approve
Subagent	Kind-Session, eigener Kontext
Worktree	Isolierte Git-Arbeitskopie
Skill	`SKILL.md`-Workflow, oft `/name`
Hook	Lifecycle-Script, kann deny
MCP	Externe Tools über Standardprotokoll
ACP	Agent Client Protocol (IDE/Server)
Always-approve	YOLO – Deny/Hooks bleiben
Folder-Trust	Gate für project MCP/Hooks/LSP

18.15 Lesepfad (Recap)

Minimum (Anfänger, ~90–120 Min.): 00 → 01 skimmen → 02 (+ `beispiele/todo-cli`) → 03 Day-1 → diese Woche 04 + 07; optional 15.2 todo-cli-Variante. Siehe Vorwort „Anfänger-Lesepfad“.

1. 00 Vorwort · 01 Philosophie
2. 02 Install · 03 TUI
3. 04 Plan · 05 Subagents · 06 Worktrees
4. 07 AGENTS.md · 08 Skills/Hooks · 09 MCP
5. 10 Headless · 11 ACP
6. 12 Memory/Sessions · 13 Safety · 14 Config
7. 15 Use-Cases · 16 BP/Troubleshoot · 17 Vergleich

Was du jetzt kannst

- Install, Sessions, Modes, Plan, Headless und Pfade ohne Suchen nachschlagen
- Symptom-Minis und Prompt-Schablone im Alltag einsetzen
- In die richtigen Kapitel zurückspringen

Nächste Schritte

Zurück in die Praxis: ein echtes Ticket mit Use-Case aus Kapitel 15 und der Schablone aus 18.11. Bei Lücken im Buch: Issue/PR am Repo – und immer die aktuelle CLI-Hilfe gegenlesen.