Java Forum Stuttgart 2026 · KI & Trends

Dein Coding Agent belügt dich nicht — er vergisst dich.

Was wirklich zwischen Tool und LLM passiert — sichtbar gemacht mit einem transparenten API-Proxy.

Frederik Wystup · MeiLuft · XaresAICoder · Do 9. Juli 2026

Über dieses Dokument

Diese Fassung des Foliensatzes ist zum Nachlesen gedacht — auch für alle, die nicht im Raum waren oder nicht alles behalten konnten. Oben steht jeweils die unveränderte Folie aus dem Vortrag, darunter in wenigen Absätzen, was dazu gesagt wurde: eine fette Kernaussage zum Querlesen und eine kurze Erklärung für den Tiefgang.

Alle Token-, Kosten- und Cache-Zahlen sind eigene Messungen, aufgezeichnet mit einem transparenten API-Proxy zwischen Coding Agent und Sprachmodell (Messlauf 07.07.2026). Werkzeug und Proxy sind Open Source: XaresAICoder, github.com/dg1001.

40 Min Vortrag · 36 Folien · Messlauf 07.07.2026 · 4 Tools · 3 Modelle

> **Was aussieht wie ein Kollege mit Projektgedächtnis, ist in Wahrheit ein zustandsloser Textvervollständiger mit sehr guter Vorarbeit.**

Oben stehen vier Kommandozeilen-Agenten nebeneinander: Claude Code (Opus 4.8), OpenAI Codex (gpt-5.5), OpenCode (Big Pickle über OpenCode Zen) und Aider (deepseek/deepseek-chat). Vier Startbildschirme, ein Eindruck — jeder wirkt wie ein Werkzeug, das das geöffnete Projekt bereits kennt.

Dieser Eindruck entsteht aus flüssiger Konversation, Architektur-Überblick und scheinbarer Erinnerung an das, was vorhin besprochen wurde. „Sie wirken, als würden sie dein Projekt verstehen“ ist genau das: Wirkung. Technisch ist jedes dieser Werkzeuge ein zustandsloser Textvervollständiger — zwischen zwei Anfragen speichert das Modell nichts.

„Sehr gute Vorarbeit“ heißt: Was wie Projektgedächtnis aussieht, wird bei jedem Aufruf neu zusammengesetzt und komplett mitgeschickt. Die Frage, die dieser erste Teil technisch beantwortet, lautet deshalb: Woher kommt dieses „Verstehen“ wirklich?

> **Ein Sprachmodell ist zustandslos: Zwischen zwei API-Calls behält es 0 Bytes — jeder Aufruf beginnt bei null, als hätte das Modell den Nutzer nie zuvor gesehen.**

„stateless“ ist der Fachbegriff für den Kern jeder Interaktion mit einem Sprachmodell: Das Modell ist eine reine Funktion — Tokens rein, Tokens raus. Zwischen zwei API-Calls existiert kein Zustand und keine Persistenz. Es gibt keinen inneren Speicher, in dem ein vorheriges Gespräch nachhallt.

Die „0 Bytes“ auf der Folie sind wörtlich gemeint. Jeder Aufruf startet ohne Erinnerung an den vorigen — als säße man einem Gegenüber gegenüber, das nach jedem Satz das Gedächtnis verliert. Der Eindruck eines fortlaufenden Dialogs entsteht erst dadurch, dass die gesamte Historie bei jedem Call komplett neu mitgeschickt wird (Seite 7).

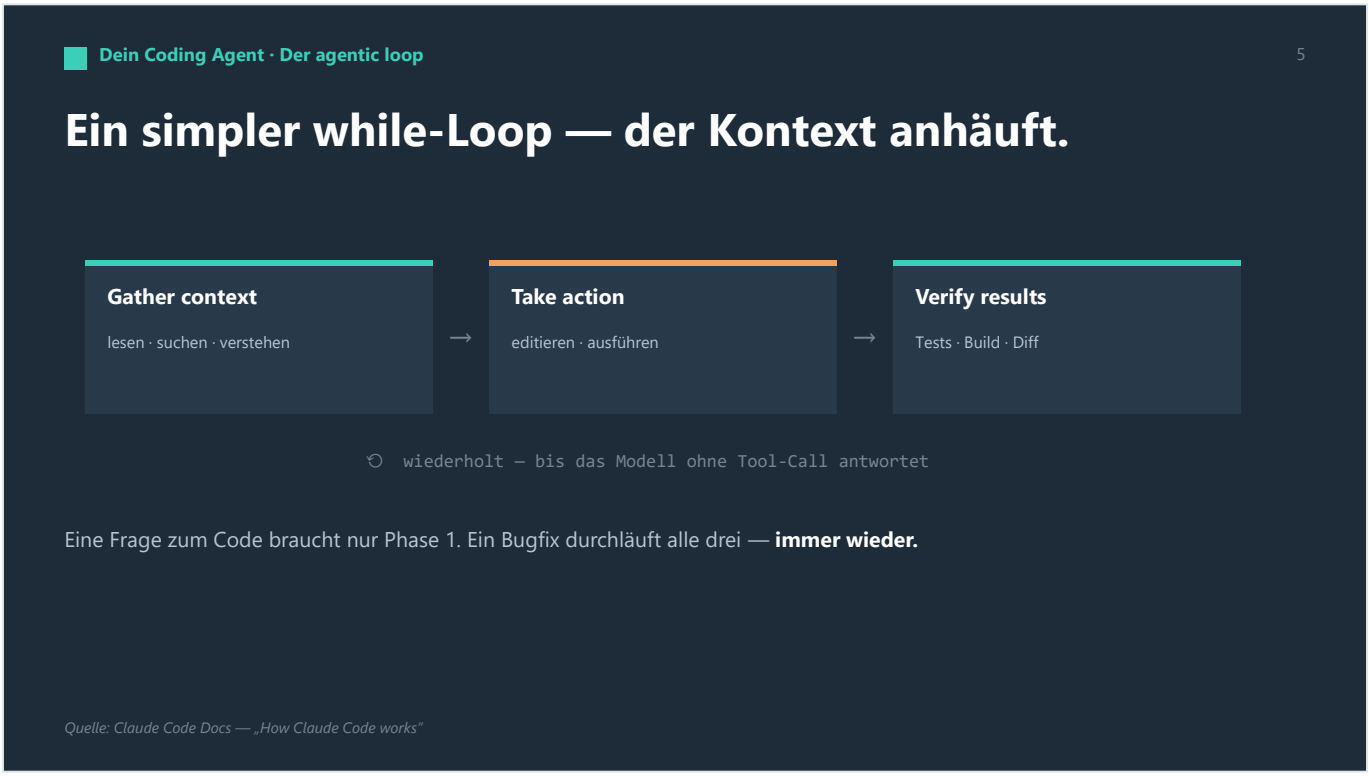
Ein Cache ändert an dieser Zustandslosigkeit nichts — er beschleunigt nur die Wiederverarbeitung des immer gleichen Präfixes, er ersetzt kein Gedächtnis (Seite 21).

> **Das Sprachmodell hat weder Gedächtnis noch Dateizugriff noch eine Shell — jede scheinbare Fähigkeit über die Zeit ist eine Leistung des Tools, nicht des Modells.**

Die Folie stellt vier Dinge nebeneinander, die ein Sprachmodell strukturell nicht tut. Es hat kein Gedächtnis: Nichts überlebt zwischen zwei Calls. Es liest keine Dateien — das Tool liest den Inhalt und reicht ihn als Text durch. Es führt nichts aus — das Tool ruft Befehle auf, nicht das Modell. Und es kennt das Projekt nicht — das Tool lädt den Kontext bei jedem Call neu.

Was im Gespräch wie Erinnerung, Datei- oder Systemzugriff wirkt, ist also jedes Mal Tool-Engineering rundherum. Der Coding-Agent baut diese Fähigkeiten um das Modell herum; das Modell selbst bleibt bei jedem Aufruf gleich blank.

Ein Bild dazu: Das Modell ist der Motor, das Tool ist das ganze Auto. Alles, was wie Intelligenz über die Zeit aussieht, ist eine Leistung des Tools — nicht des Modells. Dieser Unterschied trägt durch den ganzen ersten Teil.



> **Ein Coding Agent ist im Kern ein simpler while-Loop — Kontext sammeln, handeln, prüfen, wiederholen —, und jede Runde häuft weiteren Kontext an.**

Was aussieht wie ausgefeilte Autonomie, ist im Kern ein while-Loop. Die drei Stationen auf der Folie — Kontext sammeln (lesen, suchen, verstehen), handeln (editieren, ausführen), Ergebnisse prüfen (Tests, Build, Diff) — laufen im Kreis, so lange, bis das Modell einmal ohne Tool-Call antwortet.

Wie oft der Kreis sich dreht, hängt an der Aufgabe. Eine bloße Frage zum Code braucht nur die erste Phase und ist nach einem Durchlauf fertig. Ein Bugfix oder ein größeres Feature durchläuft alle drei Stationen — immer wieder.

Anthropic beschreibt das als „a simple, single-threaded master loop combined with disciplined tools“ — kontrollierbare Autonomie aus wenig Mechanik. Entscheidend für alles Folgende ist ein Nebeneffekt: Jede Runde legt weiteren Kontext auf den Stapel, der Loop wächst mit jedem Durchlauf.

Quelle: Claude Code Docs — „How Claude Code works“.

> **Der Satz, den der Mensch tippt, ist meist die kleinste Schicht — jeder Request trägt System-Prompt, Tool-Schemas, History und Projekt-Kontext huckepack, bevor die eigentliche Frage überhaupt beginnt.**

Die Folie zerlegt eine einzige „simple Frage“ in ihre Schichten. Von oben nach unten: der System-Prompt mit Tool-Persona und Regeln (oft 5.000+ Zeichen), die JSON-Schemas aller verfügbaren Werkzeuge, die komplette Conversation History dieser Session und der Datei- und Projekt-Kontext — Auszüge, Repo-Map, AGENTS.md, Tool-Outputs. Erst ganz unten steht der User-Prompt: der eine Satz, den der Mensch tippt.

Diese Anatomie gilt für jedes Tool, nicht nur für eines. Community-Messungen zeigen: Schon der Start eines Requests kostet in der Größenordnung von ~8.000 Tokens, bevor der Nutzer überhaupt etwas eingibt. Der eigentliche Prompt ist also nur ein Bruchteil des Payloads — meist die kleinste Schicht im Stack.

Wer den Agenten verstehen will, schaut deshalb nicht auf den getippten Satz, sondern auf alles, was ihn trägt.

Startkosten ~8.000 Tokens: Community-Messungen (Größenordnung), siehe docs/FACTS.md

Dein Coding Agent · Bei jedem Turn: alles neu

7

Zustandslos heißt: das Tool schickt bei jedem Call den kompletten Verlauf erneut.

> **Dein 50. Prompt ist dramatisch teurer als dein erster.**
>
> System-Prompt + Tools + History — Turn für Turn

> **Ein Coding Agent ist zustandslos: Bei jedem Turn schickt das Tool den kompletten Verlauf erneut — deshalb wird der 50. Prompt dramatisch teurer als der erste.**

Ein Sprachmodell hat kein Gedächtnis zwischen zwei Aufrufen — auf API-Ebene ist es zustandslos. Damit der Agent trotzdem einem Gespräch folgen kann, schickt das Tool bei jedem Turn den kompletten Stack erneut: System-Prompt, Werkzeug-Definitionen und die gesamte bisherige History.

Das hat eine ökonomische Konsequenz: Jeder Turn stellt den gesamten Kontext neu in Rechnung. Die Konversation wächst, die bearbeiteten Dateien werden mehr — und mit ihr wächst der Preis pro Turn. Der 50. Prompt ist dramatisch teurer als der erste, selbst wenn er nur eine kurze Rückfrage enthält.

Wie hoch dieser Preis konkret wird, zeigen die folgenden Kapitel — hier zählt zunächst das Prinzip: zustandslos heißt, jeder Turn bezahlt die gesamte History noch einmal.

Dein Coding Agent · Teil 2

8

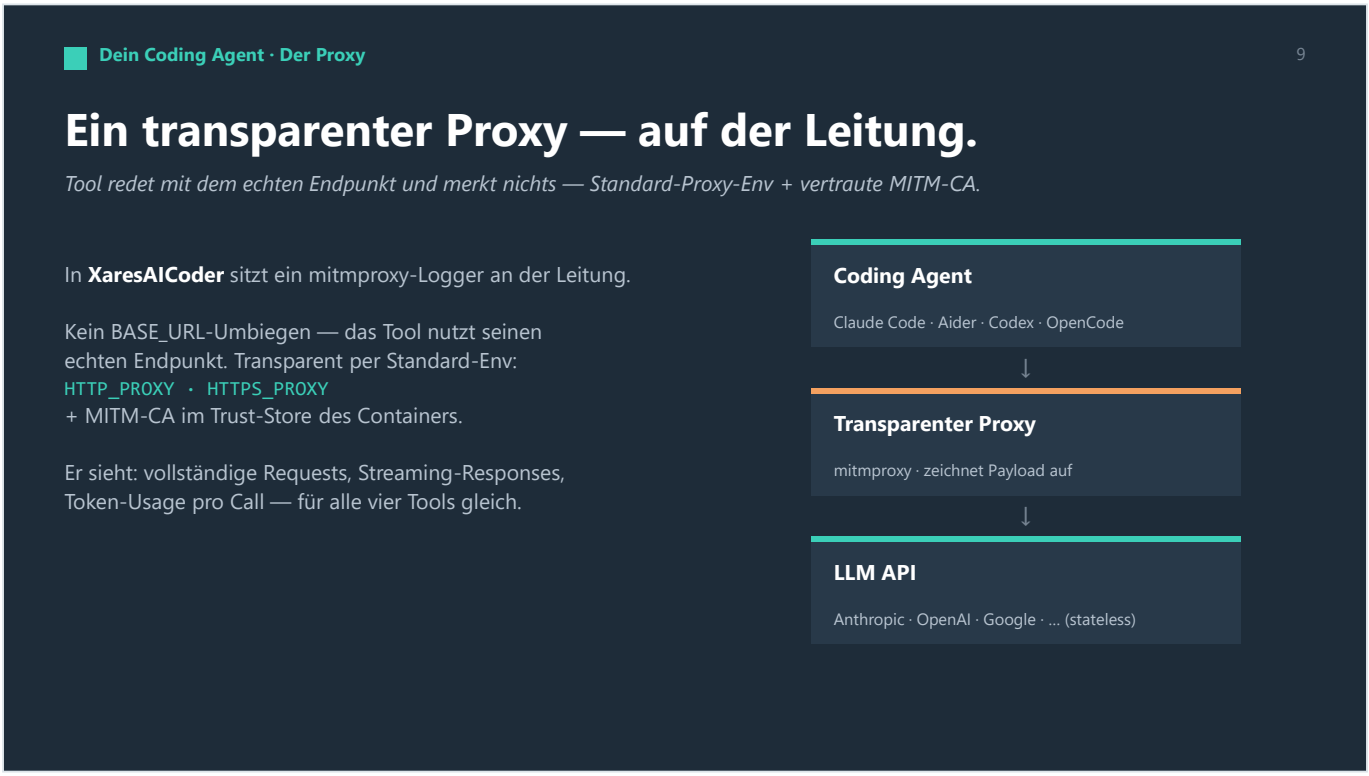
Sichtbar machen.

Ein transparenter Proxy an der einzigen Stelle, durch die jeder Agent muss: dem API-Call.

JFS 2026 · 9. Juli · Frederik Wystup

> **Jeder Agent, egal welches Tool, muss durch denselben Flaschenhals — den API-Call; genau dort setzt ein transparenter Proxy an und macht sichtbar, was wirklich gesendet wird.**

Teil 2 verlässt die Theorie und schaltet einen Messpunkt dazwischen: einen Proxy auf der Leitung, agent-agnostisch und nicht ins Tool eingebaut. Die folgenden Seiten lesen aus den mitgeschnittenen API-Calls ab, welches Modell wirklich antwortet, wie viele Tokens fließen und was Caching spart.

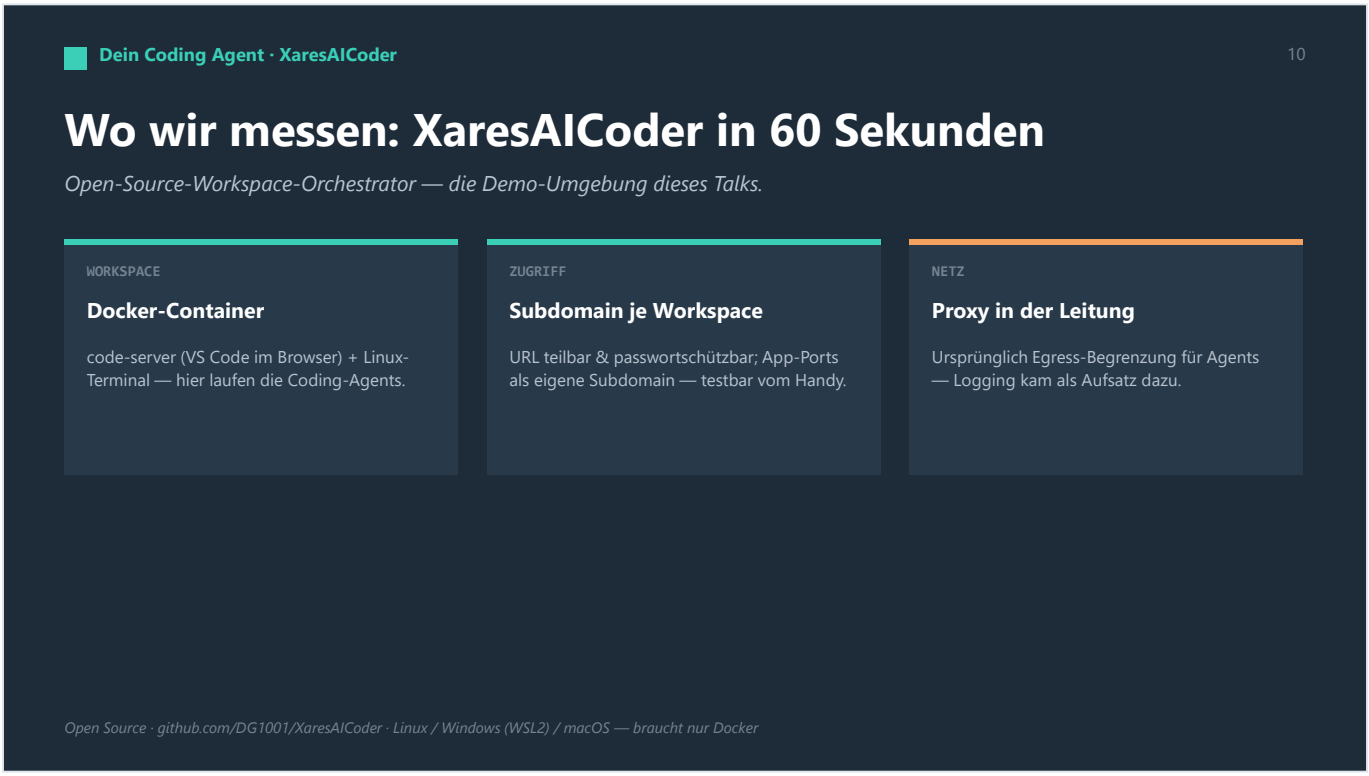


> **Ein mitmproxy auf der Leitung zeichnet jeden Request und jede Response auf — transparent, weil kein Tool umkonfiguriert wird und keines etwas davon merkt.**

Die Messanordnung: In XaresAICoder hängt ein mitmproxy zwischen Coding Agent und LLM-API. Jeder Call läuft durch, wird per TLS sauber ent- und wieder verschlüsselt und vollständig aufgezeichnet — komplette Requests, Streaming-Responses, Token-Usage pro Call, für alle vier Tools gleich.

„Transparent“ ist wörtlich gemeint: Das Tool redet mit seinem echten Endpunkt (api.anthropic.com, …); es wird keine BASE_URL umgebogen. Der Mechanismus sind die Standard-Umgebungsvariablen HTTP_PROXY/HTTPS_PROXY, die auf den mitmproxy zeigen, plus dessen MITM-CA im Trust-Store des Containers. Eine Konfiguration gilt damit für alle vier Tools — der Proxy ist agent-agnostisch, weil er auf der Leitung sitzt statt im Tool.

Diese Abgrenzung trägt später noch einmal: Eine BASE_URL pro Tool umzubiegen wäre gerade nicht transparent — genau dieser Mechanismus taucht auf Seite 31 als Angriffsvektor wieder auf.

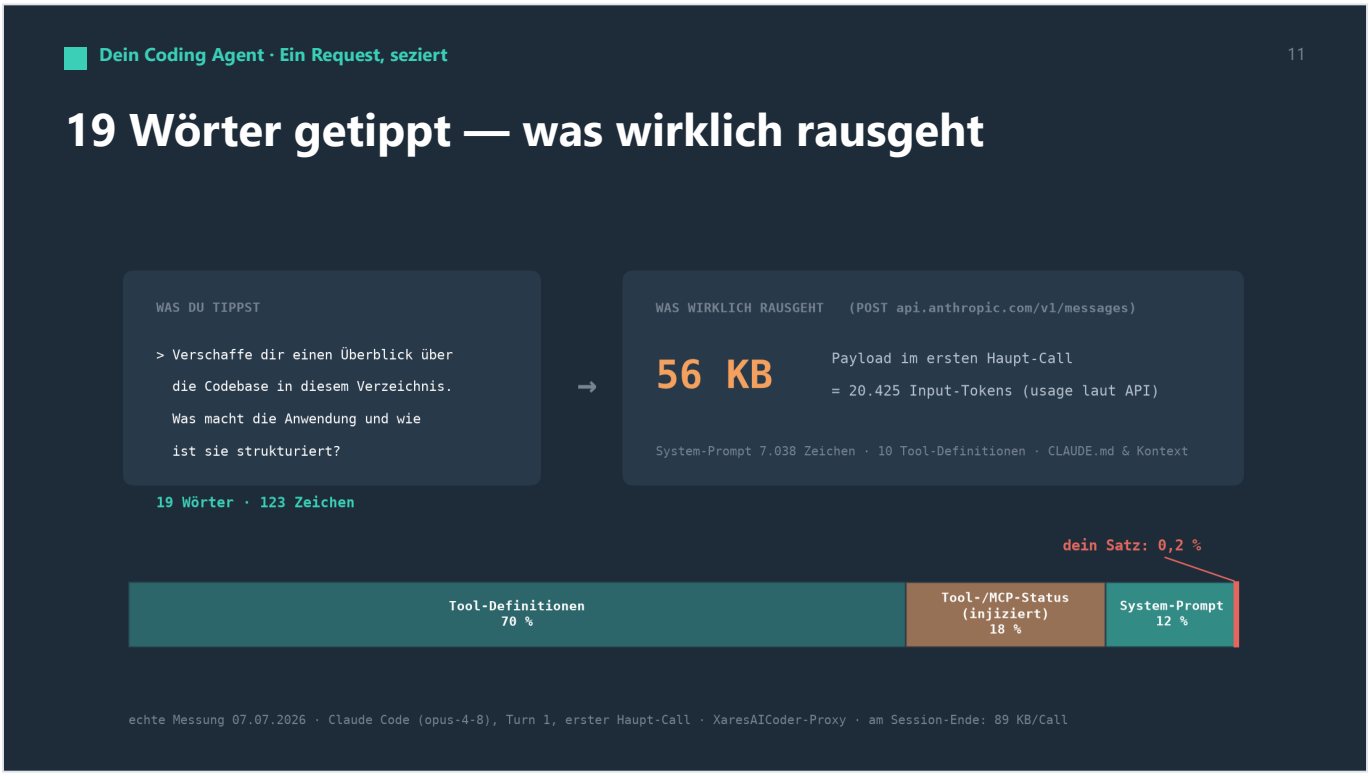


> **XaresAICoder ist der Open-Source-Orchestrator, auf dem diese Demo läuft: jeder Workspace ein Docker-Container mit VS Code im Browser — und ein Proxy in der Leitung.**

XaresAICoder ist ein Open-Source-Workspace-Orchestrator und die Demo-Umgebung dieses Talks — kein Produkt-Vortrag. Jeder Workspace ist ein eigener Docker-Container; darin läuft code-server (VS Code im Browser) samt Linux-Terminal, in dem die Coding-Agents arbeiten. Als Basis genügt Docker — auf Linux, Windows (WSL2) und macOS.

Jeder Workspace bekommt eine eigene Subdomain-URL: teilbar und passwortschützbar. Startet eine Applikation, wird ihr Port als eigene Subdomain eingebunden — sofort vom Handy oder von Kollegen testbar.

Für diesen Talk zählt vor allem die Spalte NETZ: der Proxy in der Leitung. Ursprünglich diente er der Egress-Begrenzung der Agents — eine Sandbox schützt nicht nur das Dateisystem, sondern auch das Netz. Das Request-Logging kam als Aufsatz dazu. Weil alle Container im selben Docker-Netz laufen, hängt er sich einfach dazwischen — und macht sichtbar, was die folgenden Seiten messen.



19 getippte Wörter lösen einen Request aus, in dem der eigene Satz nur 0,2 Prozent ausmacht — der Rest ist Werkzeug- und Kontextverwaltung, noch bevor ein Zeichen Projektcode fließt.

DIE ZAHLEN DER FOLIE

19 Wörter · 0,2 % — Der getippte Prompt — 123 Zeichen — löst den ersten Haupt-Call einer Claude-Code-Session aus und macht nur 0,2 % des Payloads aus.

56 KB · 20.425 Tokens — Payload des ersten Haupt-Calls (usage laut API): System-Prompt mit 7.038 Zeichen, 10 Tool-Definitionen, CLAUDE.md & Kontext — noch ohne jeden Projekt-Inhalt.

70 % — Die Tool-Definitionen dominieren den Payload — der mit Abstand größte Block des ersten Requests.

18 % · ~9,3 KB — „Tool-/MCP-Status“: eine von Claude Code injizierte role:system-Message mit reiner Kapazitäts-Buchhaltung — welche Tools nachladbar sind, welche MCP-Server erst Auth brauchen.

89 KB / Call — Am Session-Ende trägt jeder Call so viel — die Folge-Calls füllen sich mit den Datei-Inhalten, die das Modell per „ls“ und Reads anfordert.

HINTERGRUND

Die Grafik seziert einen echten, am Proxy mitgeschnittenen Request — bewusst als statisches Bild, unabhängig vom Konferenz-WLAN. Es ist der erste Haupt-Call: Bevor irgendein Projektcode fließt, ist fast alles Werkzeug-Verwaltung. Das Modell antwortet darauf zunächst mit „lies Dateien, mach ls“ — erst die folgenden Calls tragen die Inhalte und wachsen mit.

Der Anteil hängt von der Umgebung ab: Dieser Lauf hatte viele MCP-Server, ein nacktes Claude Code hätte hier weniger. Die interaktive Live-Fassung folgt am Ende (Seite 29).

Messdaten: eigener Proxy-Mitschnitt einer Claude-Code-Session (erster Haupt-Call)

> **Vier Tools bekommen dieselbe Aufgabe über dasselbe Protokoll — und bauen dem Modell auf vier völlig verschiedene Arten ein „Gedächtnis“ aus purem Text.**

Teil 3 stellt Claude Code, Aider, Codex und OpenCode nebeneinander. Alle lösen dieselbe Aufgabe über fünf Turns, alle sprechen dasselbe API-Protokoll — doch jedes Tool schnürt einen anders großen Payload und pflegt den Kontext anders. Die folgenden Seiten messen genau diese Unterschiede.

> **Vier Werkzeuge, dieselbe Aufgabe — und vier völlig verschiedene Architekturen: Der Unterschied liegt nicht im Modell, sondern komplett im Tool.**

Die vier Karten zeigen dieselbe Ausgangslage: eine Erkundungsfrage, drei Folgefragen, ein Edit — bewusst einfach gehalten, damit der Vergleich und nicht das Szenario zählt. Jedes Werkzeug löst diese Aufgabe anders.

Aider baut eine deterministische Repo-Map aus Tree-sitter und PageRank; Claude Code ist die Harness mit System-Prompt, CLAUDE.md, Subagents und Kompaktierung (Seite 15); Codex CLI spricht ein eigenes Payload-Format mit eigenen Sandbox-Modi; OpenCode ist die offene, provider-agnostische Variante — Free Tier inklusive, aber mit verstecktem Routing. Nebeneinander im Proxy werden diese Unterschiede sofort sichtbar — die Messwerte dazu folgen auf den nächsten Seiten.

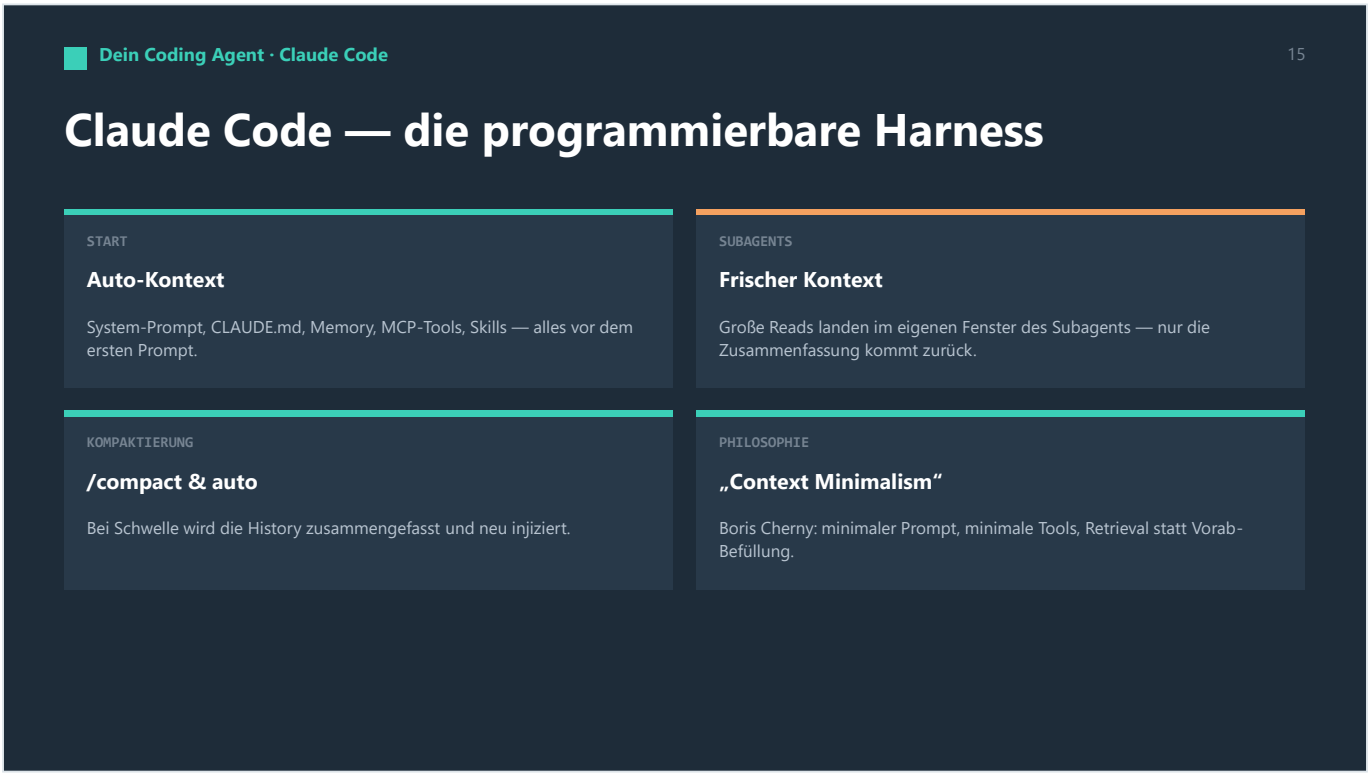
Gemini CLI fehlt bewusst: Google hat es am 18.06.2026 für alle Privatkunden abgeschaltet — wer Workflows darauf gebaut hatte, stand über Nacht ohne Tool da. Deshalb rückt OpenCode als vierter nach — nebenbei ein Argument für offene Werkzeuge.

> **Aider löst das Kontextproblem mit klassischer Informatik: eine komprimierte Repo-Karte aus reinen Signaturen, gewichtet per personalisiertem PageRank, statt das Modell alle Dateien lesen zu lassen.**

Aider steht hier für eine These: Das Kontextproblem lässt sich auch mit klassischer Informatik lösen, nicht nur mit größeren Modellen. Statt „lies alle Dateien“ baut das Tool eine komprimierte Karte des Repos — nur Signaturen, keine Implementierungen.

Die vier Kästen auf der Folie zeigen die Pipeline von links nach rechts: Tree-sitter parst die Codebasis in Syntaxbäume (130+ Sprachen), daraus entsteht ein Graph aus Definitionen und Referenzen, darauf läuft ein personalisierter PageRank — der Google-Algorithmus, mit 50-fachem Gewicht für die gerade bearbeiteten Chat-Dateien —, und heraus kommt eine Karte im Budget von etwa 1k Tokens. Sie wird vor jedem Call neu gebaut und folgt so der Konversation.

Der Effekt: Das Tool weiß selbst, was relevant ist, statt es das Modell erkunden zu lassen — und schickt dadurch deutlich weniger Tokens. Dazu bleibt Aider git-nativ, funktioniert mit beliebigen Modellen (bring-your-own-model) und hält seine Prompts offen.



> **Claude Code ist eine programmierbare Harness — alles um das Modell herum (Loop, Tools, Prompts, Kontext-Management) lässt sich anpassen; Leitprinzip ist „Context Minimalism“.**

Harness heißt wörtlich „Geschirr“, wie beim Pferd: gemeint ist alles, was rund um das Sprachmodell gebaut ist und es einspannt — der Loop, die Tools, die Prompts, das Kontext-Management. Agent = Harness + Modell. „Programmierbar“ heißt: Bei Claude Code lässt sich fast jede Schicht anpassen — Skills, Hooks, Subagents, MCP.

Die vier Blöcke zeigen, wie diese Harness den Kontext bewirtschaftet. Beim Start (Auto-Kontext) liegen System-Prompt, CLAUDE.md, Memory, MCP-Tools und Skills schon vor dem ersten Prompt bereit. Subagents bekommen ein eigenes Kontextfenster: Große Reads landen dort, nur die Zusammenfassung kommt zurück — so bleibt der Hauptkontext klein. Kompaktierung fasst die History bei einer Schwelle zusammen und injiziert sie neu, per /compact oder automatisch.

Die Klammer ist „Context Minimalism“: Boris Cherny, Schöpfer und Lead von Claude Code bei Anthropic, formuliert minimaler Prompt, minimale Tools, Retrieval statt Vorab-Befüllung. Das Prinzip gilt für alle Agents; Claude Code hat es explizit zur Design-Philosophie erklärt. Subagents sind dabei das stärkste Muster gegen verrauschten Kontext.

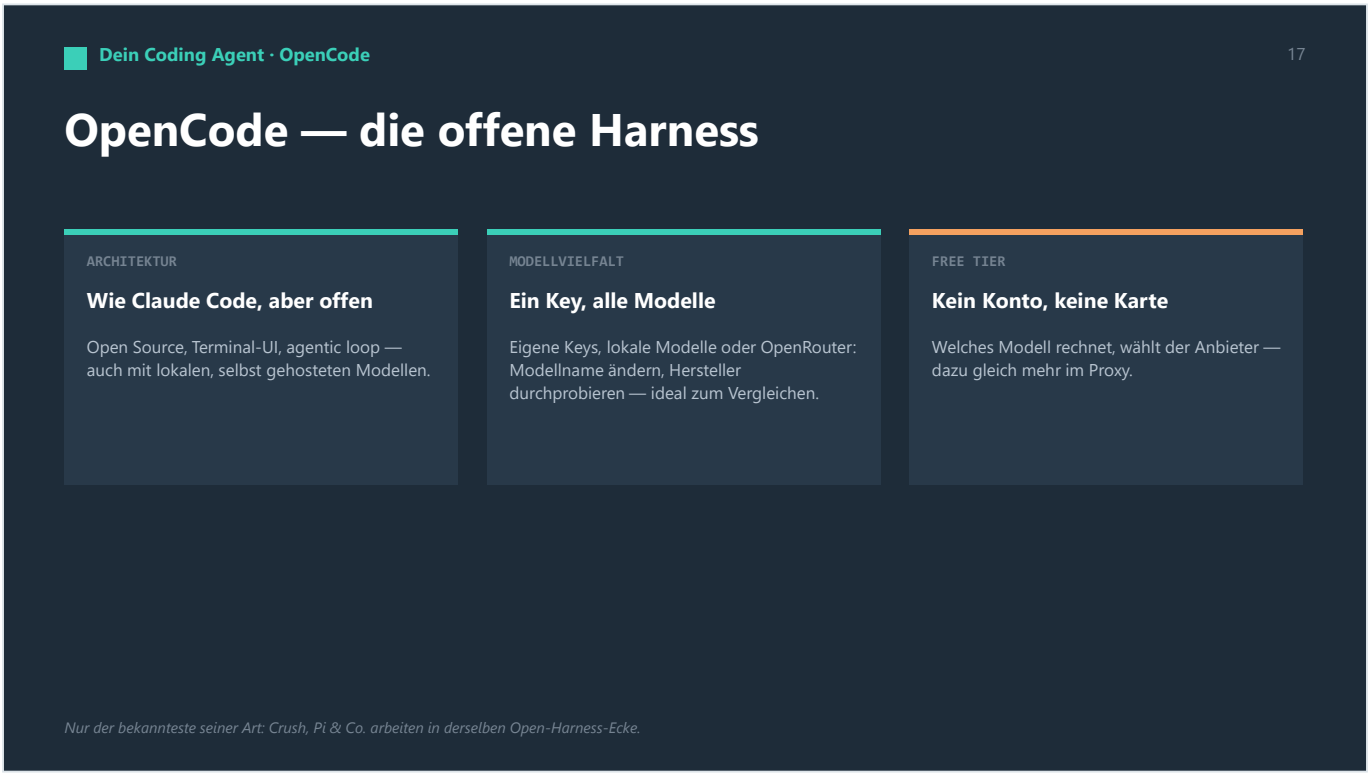
> **Codex CLI ist der architektonische Außenseiter: als einziges der vier Tools spricht es die Responses API statt Chat Completions — ein Unterschied, den der Proxy sofort sichtbar macht.**

Die Code-Blöcke auf der Folie zeigen denselben Auftrag in zwei Formaten: Chat Completions schickt POST /v1/messages mit einem messages-Array (so arbeiten Claude, Aider, Gemini), Codex dagegen POST /v1/responses mit input-Array und store: true. Der Grund für die Responses API: OpenAI kann den Session-Zustand serverseitig halten, das Tool muss weniger erneut senden, der Anbieter kann serverseitig cachen. Kehrseite ist ein proprietäres Format und eine engere Bindung an OpenAI — die anderen bleiben beim portablen Chat-Completions-Stil.

Auch sonst schert Codex aus: geschrieben in Rust, liest AGENTS.md, und bringt eigene Sandbox-Stufen mit — read-only, workspace-write, full-access. Seit etwa Juni 2026 streamt es zudem per WebSocket statt SSE; aufgefallen ist das beim Messen, als unser Proxy-Logger plötzlich nichts mehr sah.

Der Benchmark stützt die These des Vortrags: Auf Terminal-Bench 2.1 (Juni 2026) erreicht Codex CLI mit GPT-5.5 83,4 %. Dieselbe GPT-5.5 kommt im Terminus-2-Harness nur auf 76,4 % — der Harness allein macht 5 Punkte.

Terminal-Bench 2.1 (tbench.ai, Juni 2026), verifiziert 07.07.2026; WebSocket-Beobachtung: eigene Captures 07.07.2026



> **OpenCode ist die quelloffene Antwort auf Claude Code: dieselbe agentic Terminal-Schleife, aber offen, selbst hostbar und mit einem Modellwechsel per bloßer Namensänderung.**

OpenCode funktioniert wie Claude Code, ist aber quelloffen: Terminal-UI, agentic loop, komplett selbst hostbar — auch mit lokalen, selbst betriebenen Modellen. Die eigentliche Stärke ist aber die Modellvielfalt.

Mit eigenen Keys, lokalen Modellen oder einem einzigen OpenRouter-Key genügt es, den Modellnamen zu ändern, um jeden Hersteller durchzuprobieren — ideal zum Vergleichen. Bei Codex und Claude Code ist ein solcher Modellwechsel deutlich umständlicher.

Dazu ein Free Tier ohne Konto und ohne Karte. Welches Modell dann rechnet, bestimmt der Anbieter — der Proxy-Teil zeigt später, dass sich dahinter ein Handle namens „big-pickle“ verbirgt. Und OpenCode ist nur der bekannteste seiner Art: Crush, Pi & Co. teilen dieselbe Open-Harness-Ecke.

> **Ab hier zählen nur noch Messwerte: vier Tools, dieselbe Aufgabe, jeder Request im Proxy mitgeschnitten — am 7. Juli, zwei Tage vor diesem Talk.**

Teil 4 ist das Herzstück des Vortrags: Statt Behauptungen zählen nun Zahlen. Vier Coding-Tools lösen dieselbe Aufgabe, jeder Request läuft durch den Proxy und wird protokolliert — Tokenverbrauch, Cache-Verhalten und Dauer inklusive. Die folgenden Seiten übersetzen diese Rohdaten in echte Zahlen statt Folklore.

Messlauf: eigene Captures 07.07.2026 (XaresAICoder-Proxy)

Der Messlauf im Detail

33

Die 5 Turns — identisch für alle Tools

1 · Erkundung	Verschaffe dir einen Überblick über die Codebase in diesem Verzeichnis. Was macht die Anwendung und wie ist sie strukturiert?
2 · Folgefrage	Wie funktioniert die Authentifizierung und welche Rollen gibt es?
3 · Folgefrage	Welche API-Endpunkte gibt es und welche davon verändern Daten?
4 · Folgefrage	Wie funktioniert das Rate-Limiting und wo wird es angewendet?
5 · Edit	Ändere das Rate-Limit-Fenster von 60 auf 30 Sekunden und passe den betroffenen Test an. Führe danach npm test aus.

Testbasis: Express-API „taskboard" (Boards/Tasks · Bearer-Auth · Fixed-Window-Rate-Limit) · alle 6 Setups lösten Turn 5 korrekt

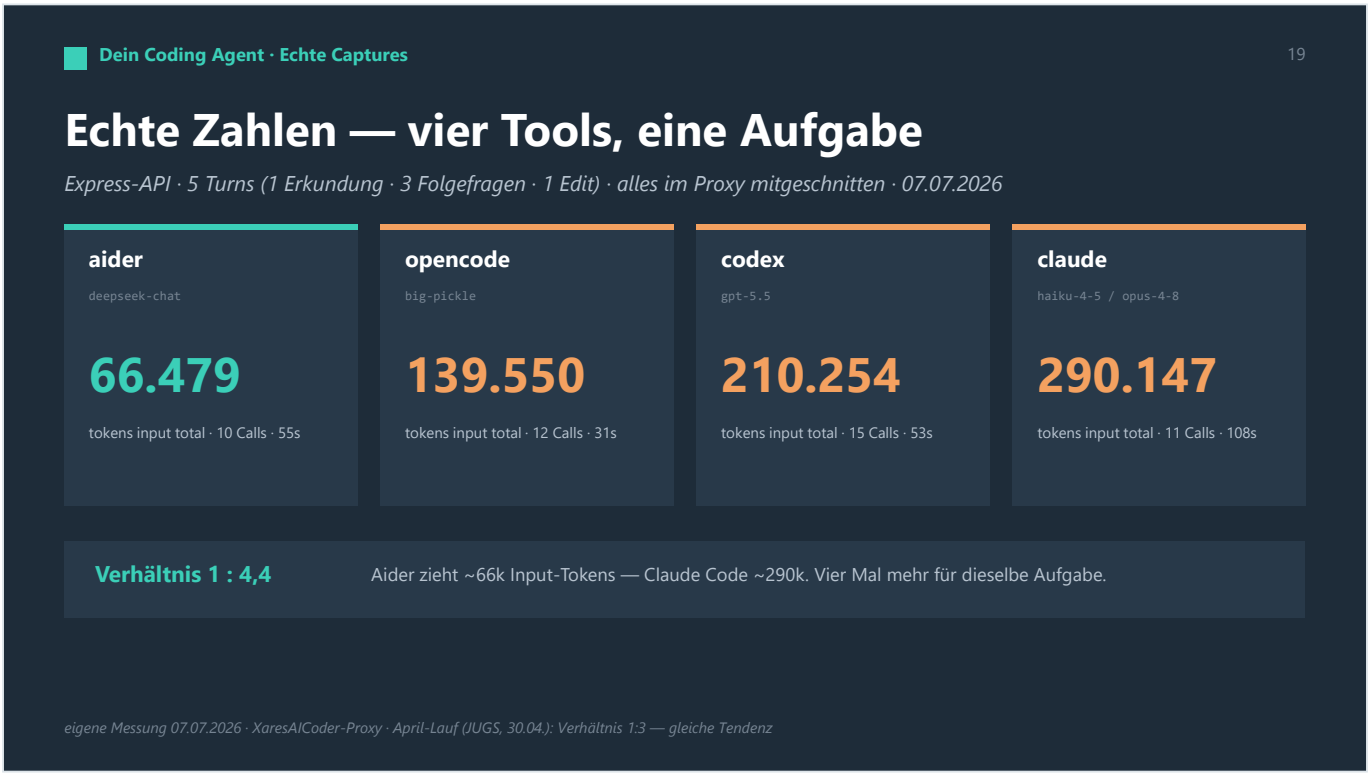
> **Alle Zahlen des Sechs-Setups-Vergleichs stammen aus genau einer Aufgabe: fünf feste Prompts — vier Fragen, ein Edit — auf derselben taskboard-API, die jedes der sechs Setups identisch durchlief.**

Damit die Token-Zahlen der einzelnen Werkzeuge überhaupt vergleichbar sind, bekam jede Konfiguration exakt dieselbe Sequenz — die fünf Prompts oben, Wort für Wort. Nur so misst der Vergleich das Werkzeug und die Harness, nicht zufällig unterschiedliche Aufgaben.

Die Sequenz beginnt mit einer Erkundung der Codebase, gefolgt von drei Folgefragen zu Authentifizierung, API-Endpunkten und Rate-Limiting — reines Lesen, das den Kontext füllt. Turn 5 ist der einzige Schreibvorgang: das Rate-Limit-Fenster von 60 auf 30 Sekunden ändern, den betroffenen Test anpassen, npm test ausführen. Alle sechs Setups lösten diesen Edit korrekt.

Testbasis ist die Express-API „taskboard" (Boards/Tasks, Bearer-Auth, Fixed-Window-Rate-Limit; 12 Dateien) — klein genug, um in einen Kontext zu passen, aber echt genug für mehrstufige Fragen.

Messlauf: eigene Captures 07.07.2026 (taskboard-Express-API, 6 Setups)



> **Dieselbe Aufgabe, vier Tools — und Claude Code zieht mit 290.147 Input-Tokens gut vier Mal so viele wie Aider mit 66.479.**

DIE ZAHLEN DER FOLIE

66.479 — Aider mit deepseek-chat, 10 Calls, 55 s — der schlankste Lauf. Die kompakte Repo-Map hält den mitgesendeten Kontext klein.

139.550 — opencode mit big-pickle, 12 Calls — mit 31 s der schnellste Lauf, aber schon rund doppelt so viel Input wie Aider.

210.254 — codex mit gpt-5.5, 15 Calls, 53 s — die meisten Einzel-Calls im Vergleich, deutlich mehr Input als opencode.

290.147 — Claude Code mit haiku-4-5 / opus-4-8, 11 Calls, 108 s — der höchste Verbrauch bei zugleich langsamster Laufzeit.

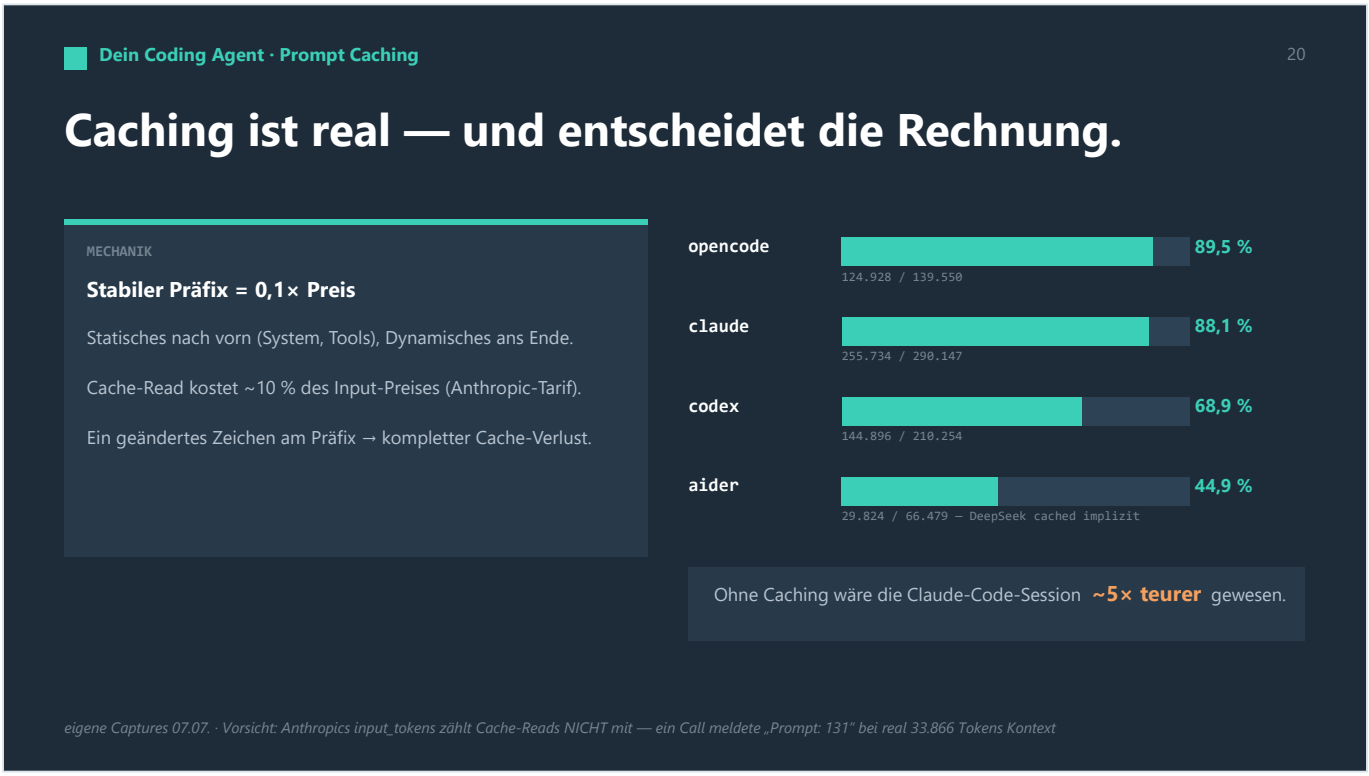
1 : 4,4 — Verhältnis Aider zu Claude Code beim Input — vier Mal mehr Tokens für exakt dieselbe Aufgabe, allein durch die Tool-Architektur.

HINTERGRUND

Alle vier Läufe lösen dieselbe Aufgabe — eine kleine Express-API „taskboard“, fünf Turns (eine Erkundung, drei Folgefragen, ein Edit), am 07.07.2026 vollständig im XaresAICoder-Proxy mitgeschnitten. Der Unterschied im Verbrauch ist reine Architektur: Aider arbeitet mit einer kompakten Repo-Map, Claude Code exploriert autonom und schickt mehr Kontext mit. Weil ein LLM zustandslos ist, werden frühe Tokens Turn für Turn erneut gesendet und bezahlt — der Input dominiert massiv, der Output bleibt winzig.

Der April-Lauf (JUGS, 30.04., andere Codebase) ergab 1:3 — dieselbe Tendenz, reproduzierbar. Warum vier Mal mehr Input trotzdem bezahlbar bleibt, klärt das Caching (Seite 21).

Messdaten: eigene Captures 07.07.2026 · XaresAICoder-Proxy



Caching ist real und entscheidet die Rechnung: Ein stabiler Präfix kostet nur ein Zehntel — ohne ihn wäre die Claude-Code-Session rund fünfmal teurer gewesen.

DIE ZAHLEN DER FOLIE

`0,1× Preis` — Ein stabiler Präfix (System, Tools vorn) wird gecacht; der Cache-Read kostet nur ~10 % des Input-Preises zum Anthropic-Tarif.

`89,5 % · 124.928 / 139.550` — opencode erreicht die höchste Hit-Rate — 124.928 von 139.550 Input-Tokens stammen aus dem Cache.

`88,1 % · 255.734 / 290.147` — Claude Code: 255.734 von 290.147 Tokens gecacht; erst das macht diesen hohen Input pro Session bezahlbar.

`68,9 % · 144.896 / 210.254` — codex liegt in der Mitte — 144.896 von 210.254 Input-Tokens aus dem Cache.

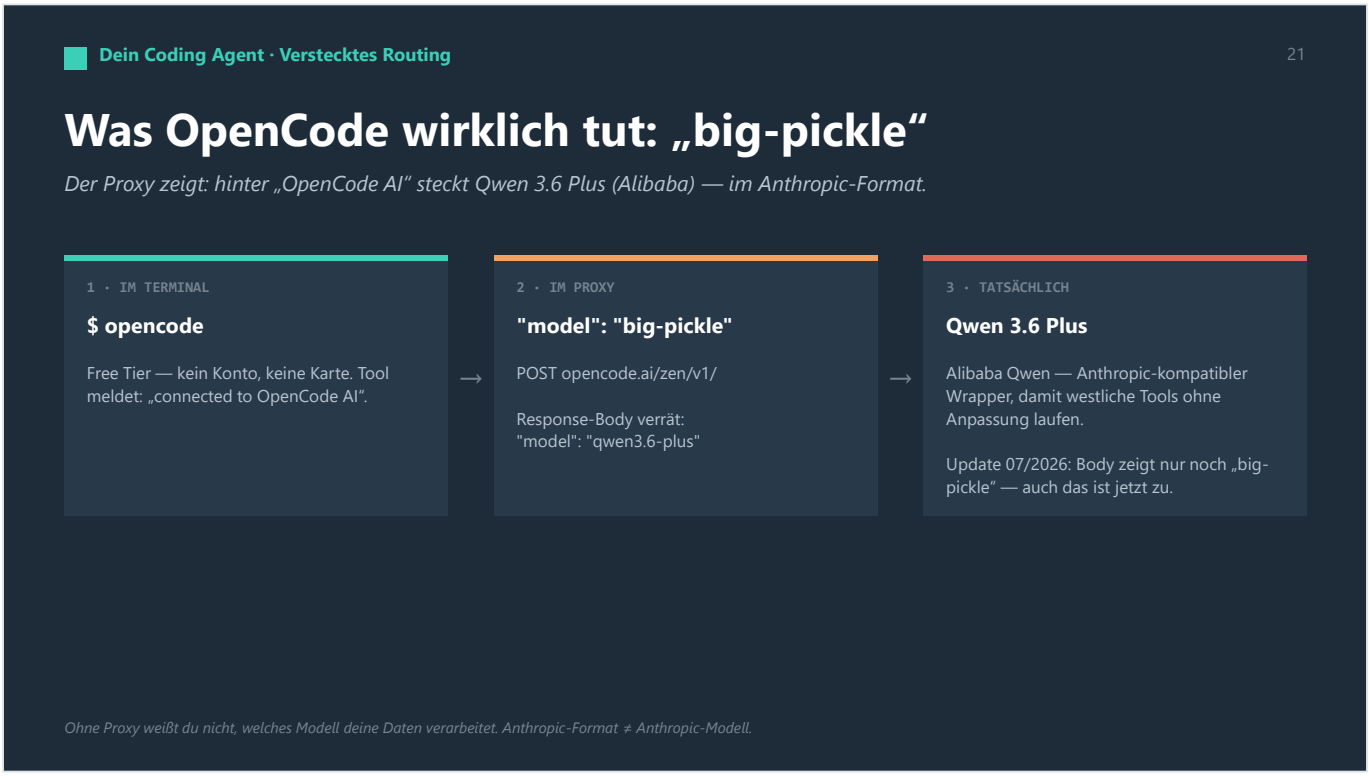
`44,9 % · 29.824 / 66.479` — aider, 29.824 von 66.479: DeepSeek cached implizit ohne Marker; im April noch 0 %, weil aider keine Anthropic-Cache-Marker setzt.

HINTERGRUND

Technisch ist Caching KV-Cache-Wiederverwendung: Beim Prefill berechnet das Modell für jedes Token die Key- und Value-Vektoren jeder Attention-Schicht — der teure, mit der Kontextlänge wachsende Teil. Für einen stabilen Präfix werden diese K/V-Tensoren serverseitig aufgehoben und über Requests hinweg wiederverwendet (mit begrenzter TTL). Weil in kausaler Attention jeder Vektor von allen Tokens davor abhängt, macht ein geändertes Zeichen vorne alles danach ungültig — daher: Statisches nach vorn, Dynamisches ans Ende.

Vorsicht bei der Messung: Anthropics input_tokens zählt Cache-Reads nicht mit — ein Call meldete „Prompt: 131“ bei real 33.866 Tokens Kontext. Wer nur input_tokens beobachtet, unterschätzt seinen Kontext massiv.

Messwerte: eigene Captures 07.07.2026 (XaresAICoder-Proxy); Preisangabe Anthropic-Tarif



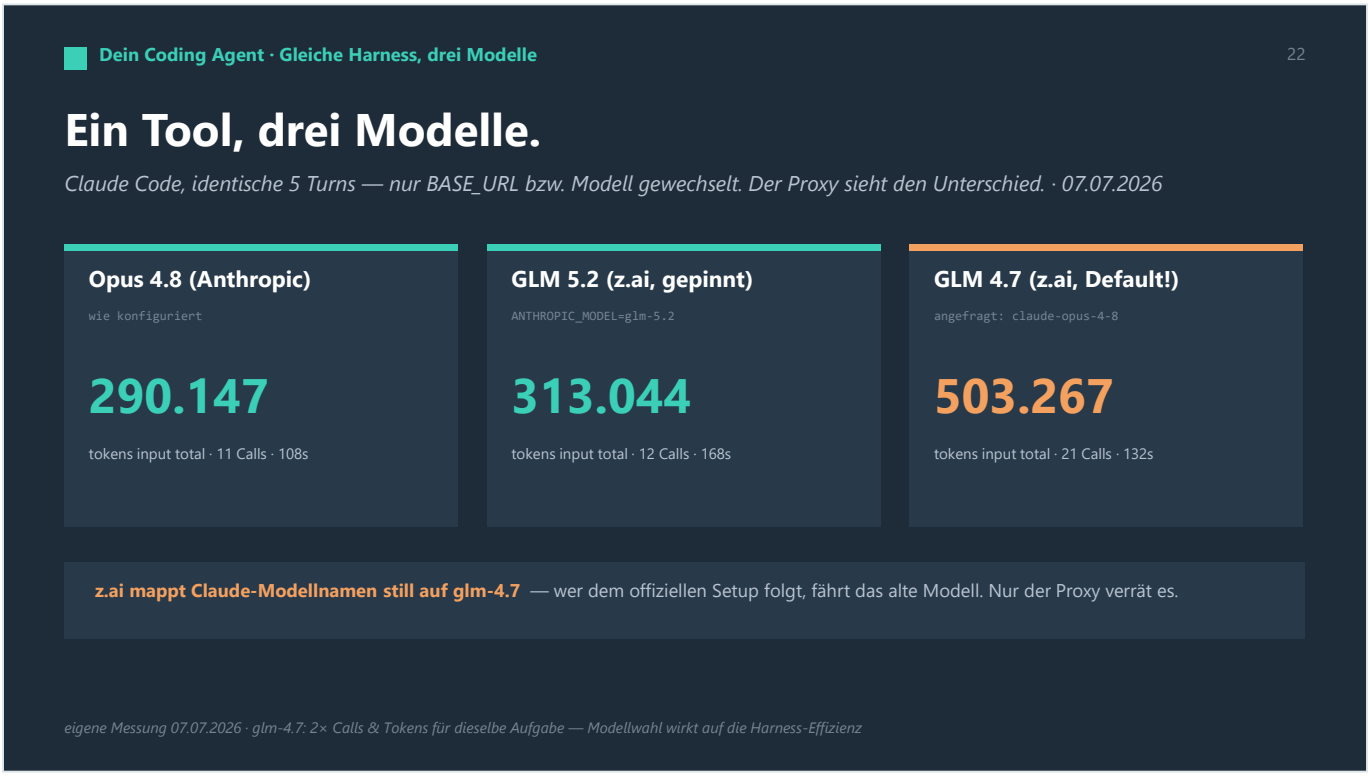
> **Der Proxy entlarvt OpenCodes Free Tier: „OpenCode AI“ ist in Wahrheit Qwen 3.6 Plus von Alibaba — Anthropic-Format bedeutet eben nicht Anthropic-Modell.**

Die Folie verfolgt einen einzigen Aufruf durch drei Ebenen. Im Terminal startet `$ opencode` das Free Tier — kein Konto, keine Karte — und meldet nur „connected to OpenCode AI“. Im Proxy heißt das Modell „big-pickle“, doch der Response-Body an `opencode.ai/zen/v1/` verrät „qwen3.6-plus“. Tatsächlich antwortet Alibabas Qwen 3.6 Plus, verpackt in einen Anthropic-kompatiblen Wrapper, damit westliche Tools ohne Anpassung laufen.

Seit dem Update 07/2026 zeigt der Body nur noch „big-pickle“ — auch dieser Hinweis ist inzwischen zu. Ohne eigene Messung im Proxy bleibt verborgen, welches Modell die Daten verarbeitet.

Das ist kein Skandal, sondern eine Einordnung: Wer professionell arbeitet, nutzt ohnehin kein anonymes Free Tier, und bei OpenCode lässt sich jederzeit explizit ein Modell wählen. Entscheidend bleibt das Prinzip — Anthropic-Format ≠ Anthropic-Modell, und Transparenz entsteht nur durch eigenes Nachmessen. Bei kostenlosen, anonymen Diensten gilt zudem: Der Input fließt vermutlich ins Training — also nie sensible Daten ins Free Tier.

Messdaten: eigener Proxy-Mitschnitt, OpenCode Free Tier (Body-Wert April 2026 / Update 07/2026)



> **Gleiche Harness, identische 5 Turns — nur das Modell wechselt: GLM 4.7 verbrennt fast doppelt so viele Calls und Tokens wie Opus, und nur der Proxy zeigt es.**

DIE ZAHLEN DER FOLIE

`290.147 · 11 Calls · 108s` — Opus 4.8 (Anthropic), wie konfiguriert. Der Referenzlauf, an dem sich die beiden z.ai-Modelle messen lassen.

`313.044 · 12 Calls · 168s` — GLM 5.2 (z.ai), explizit gepinnt via ANTHROPIC_MODEL=glm-5.2. Bei Calls und Tokens fast auf Opus-Niveau — aber mit 168s der langsamste Lauf.

`503.267 · 21 Calls · 132s` — GLM 4.7 (z.ai, Default), obwohl claude-opus-4-8 angefragt war. Rund doppelt so viele Calls und Tokens für dieselbe Aufgabe.

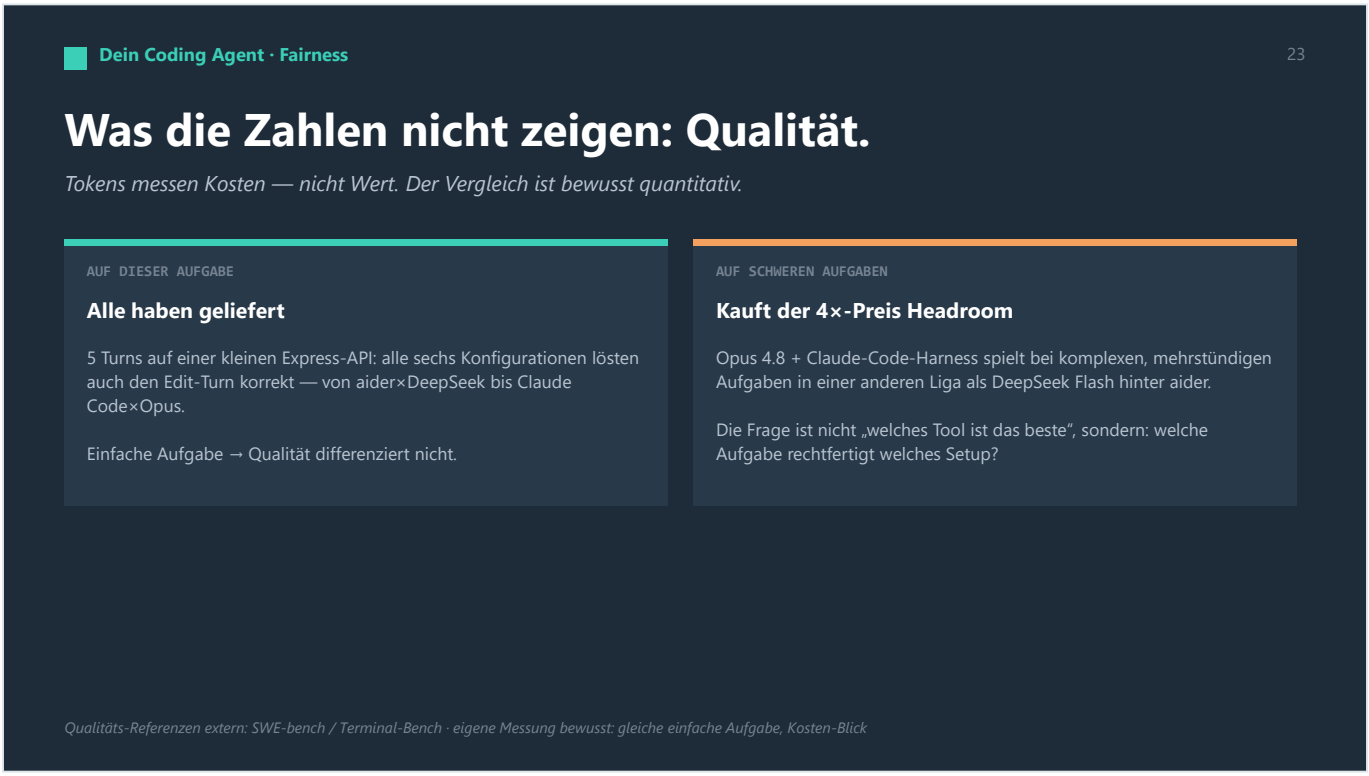
`claude-opus-4-8 → glm-4.7` — z.ai mappt Claude-Modellnamen still auf das ältere glm-4.7. Das offizielle Setup-Skript pinnt kein Modell — wer ihm folgt, fährt unbemerkt 4.7.

HINTERGRUND

Der Aufbau kehrt das Vier-Tools-Experiment um: ein Werkzeug — Claude Code —, dreimal dieselben fünf Turns, nur über ANTHROPIC_BASE_URL gegen z.ai umgelenkt. Wer dem offiziellen Setup-Skript folgt, pinnt kein Modell und fährt unbemerkt das ältere glm-4.7: Request claude-opus-4-8, Response glm-4.7. Erst der Proxy macht die stille Umleitung sichtbar.

Die Zahlen zeigen, dass die Modellwahl auf die Harness-Effizienz durchschlägt: 4.7 braucht rund das Doppelte an Calls und Tokens. Gepinnt auf glm-5.2 liegt z.ai bei Calls und Tokens fast auf Opus-Niveau — nur die Laufzeit bleibt mit 168s die höchste.

Messdaten: eigene Messung 07.07.2026 (Claude Code gegen z.ai)



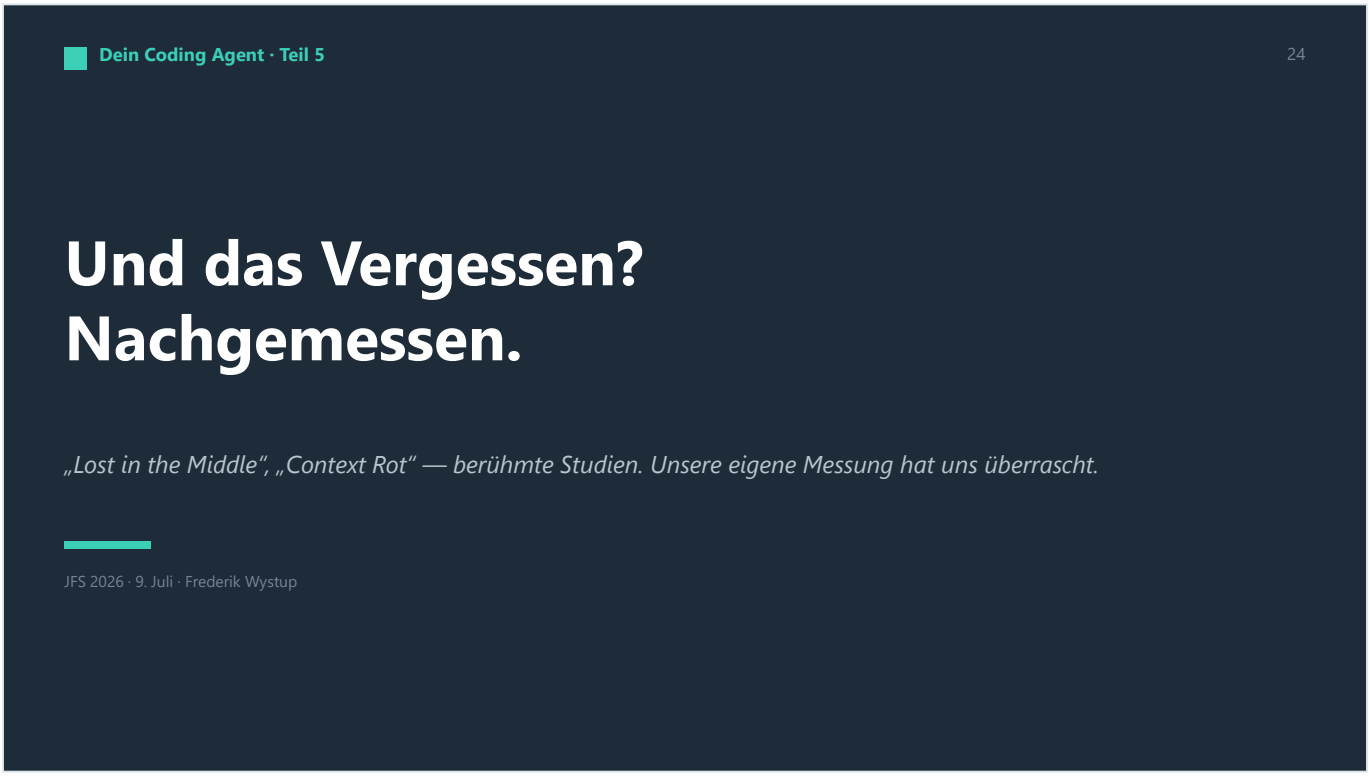
> **Tokens messen Kosten, nicht Wert: Auf der einfachen Aufgabe lieferten alle sechs Setups — auf schweren Aufgaben kauft der 4×-Preis echten Qualitäts-Headroom.**

Nach den Kostenzahlen der vorigen Seiten folgt die Ehrlichkeit: Tokens messen Kosten, nicht Wert. Der Vergleich ist bewusst quantitativ — er sagt nichts über die Qualität des erzeugten Codes.

Auf dieser Aufgabe differenziert Qualität nicht. 5 Turns auf einer kleinen Express-API — eine Erkundung, drei Folgefragen, ein Edit — lösten alle sechs Konfigurationen korrekt, von aider×DeepSeek bis Claude Code×Opus. Bei einer einfachen Aufgabe ist das teuerste Setup schlicht Verschwendung.

Auf schweren Aufgaben kippt das Bild: Opus 4.8 mit der Claude-Code-Harness spielt bei komplexen, mehrstündigen Aufgaben in einer anderen Liga als DeepSeek Flash hinter aider — hier kauft der 4×-Preis echten Headroom. Die Frage ist deshalb nicht „welches Tool ist das beste", sondern welche Aufgabe welches Setup rechtfertigt. Qualitäts-Belege dafür liefern externe Benchmarks wie SWE-bench und Terminal-Bench.

Eigener Messlauf 07.07.2026 (Express-API, 6 Konfigurationen); Qualitäts-Benchmarks SWE-bench / Terminal-Bench extern.



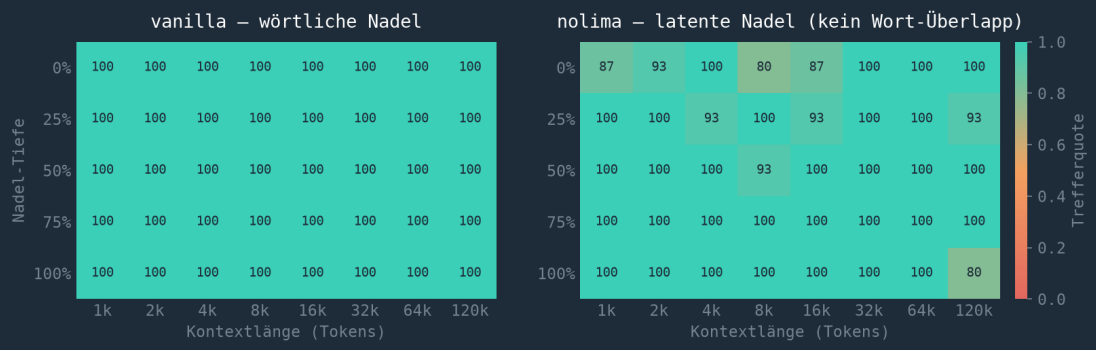
> **„Lost in the Middle“ und „Context Rot“ sind berühmte Studien — doch als wir selbst nachgemessen haben, überraschte uns das Ergebnis.**

Teil 5 nimmt zwei berühmte Befunde — „Lost in the Middle“ und „Context Rot“ — und stellt sie auf den Prüfstand: ein eigener Lab-Lauf statt Zitat. Das Ergebnis überrascht, weil der erwartete Einbruch ausbleibt. Was das für lange Kontexte wirklich bedeutet, klären die nächsten Seiten.

Dein Coding Agent · Eigene Messung

25

Nadel im Heuhaufen: 1k → 120k Tokens



eigene Messung · deepseek-chat (V4 Flash, non-thinking) · 05.07.2026 · n=15/Zelle (vanilla 64k/120k: n=5) · temperature=0 · 1.100 API-Calls

Erwartet: Kollaps mit Länge & in der Mitte. Gemessen: **97–100 % — bis 120k.**

> **Erwartet war der Kollaps — mit der Länge und in der Mitte; gemessen wurden 97–100 % bis 120k Tokens: Der befürchtete Zusammenbruch über den Kontext bleibt aus.**

DIE ZAHLEN DER FOLIE

`1k → 120k Tokens` — Über diese Kontextlänge wandert die Nadel von Position 0 bis 100 % — der klassische Bereich, in dem große Kontexte angeblich „vergessen".

`97–100 %` — Gemessene Trefferquote über beide Nadeltypen bis 120k — kein Kollaps, keine U-Kurve mit Delle in der Mitte.

`1.100 API-Calls` — Umfang des Laufs: Position 0–100 % × Länge 1k–120k, temperature=0, deterministischer Judge.

`vanilla / nolima` — Zwei Nadeltypen: wörtlich auffindbar gegen nur latent umschrieben (nach NoLiMa-Benchmark) — Letzteres ist die deutlich schwerere Aufgabe.

`11/13 Modelle < 50 %` — Was der NoLiMa-Benchmark bei 32k erwarten ließ — auf dem gemessenen DeepSeek V4 Flash reproduziert der Einbruch nicht.

HINTERGRUND

Die Folie prüft empirisch, was die Studien der Vorjahre — „Lost in the Middle", „Context Rot" — vorhersagen: dass Details in großen Kontexten verloren gehen, besonders in der Mitte. Der Test packt eine Nadel in synthetischen Fülltext und variiert Kontextlänge und Position. Gegen die Erwartung findet das günstige DeepSeek V4 Flash beide Nadeltypen bis 120k fast perfekt. Einzelne Dellen bleiben — aber kein Kollaps, keine U-Kurve. Die ehrliche Lesart: Die Modelle sind dramatisch besser geworden. Warum das Vergessen trotzdem ein Thema bleibt, klärt die nächste Seite.

Messdaten: eigenes Context-Rot-Lab (lab/), Lauf 05.07.2026 – DeepSeek V4 Flash, 1.100 Calls, 1k–120k Tokens

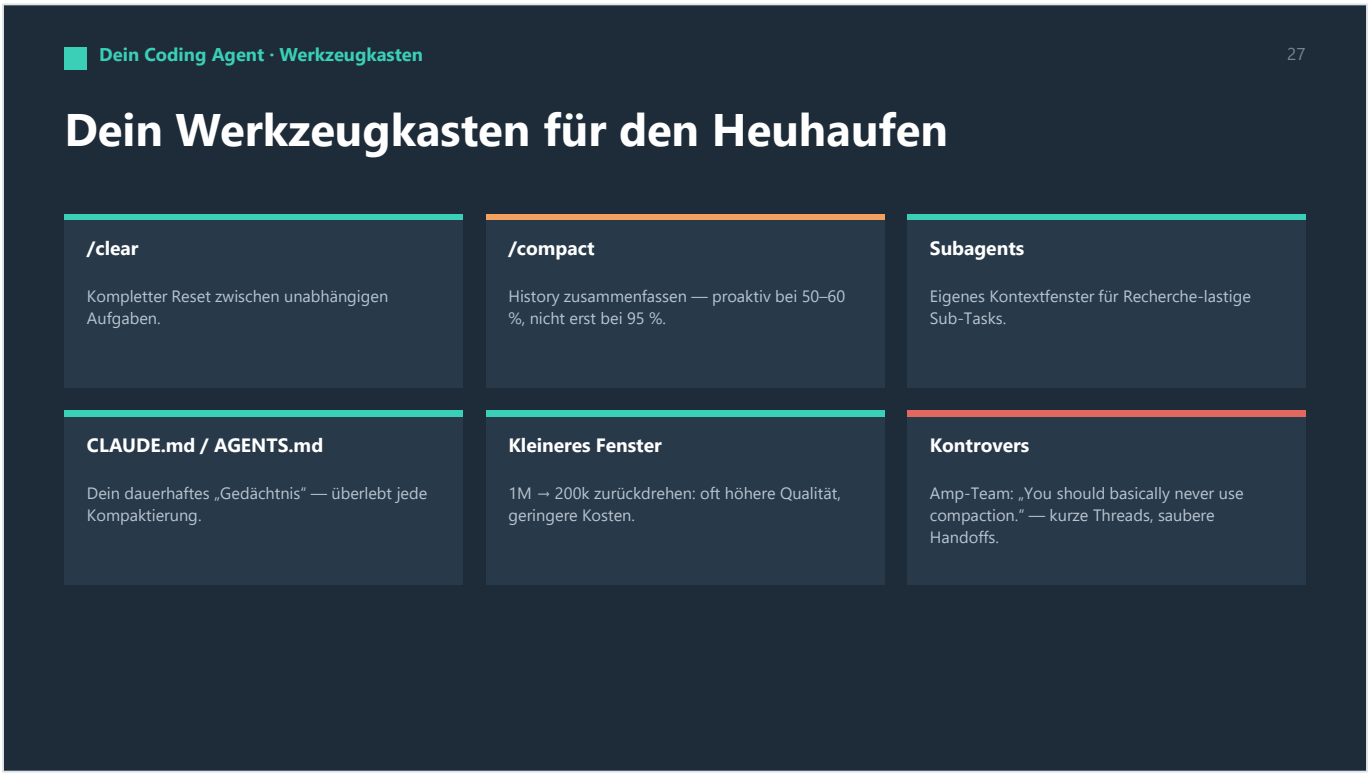
> **Moderne Modelle finden einzelne Fakten zuverlässig — das eigentliche Problem ist der Heuhaufen, den der Agent Turn für Turn selbst aufbaut und neu bezahlt.**

Die Folie trennt zwei Fragen, die gern vermischt werden. Links: Retrieval funktioniert. 2026er-Modelle finden einzelne Fakten selbst in 120.000 Tokens zuverlässig — das klassische „Nadel im Heuhaufen“-Problem ist weitgehend gefixt. Den alten Studienbefund haben wir im eigenen Labor nicht reproduziert.

Rechts steht das eigentliche Problem: Der Agent stopft den Heuhaufen selbst. 290k Tokens für 5 Turns, gemessen im eigenen Lauf — und jeder Turn bezahlt ihn erneut, weil das Modell zustandslos ist (Seite 7). Teuer wird nicht das Finden, sondern das ständige Neu-Schicken.

Der zweite Preis ist Aufmerksamkeit: Verrauschter Kontext bindet Modell-Kapazität, die dann für die eigentliche Aufgabe fehlt. Die externen Studien (Chroma, NoLiMa, Liu et al.) belegen diese Degradation vor allem bei komplexen Aufgaben — und kleinere oder lokale Modelle brechen früher ein.

Messdaten: eigener Messlauf 07.07.2026 (Claude Code, 290.147 Input-Tokens / 5 Turns) · Studien extern: Chroma „Context Rot“ 2025 · NoLiMa/Adobe 2025 · Liu et al., TACL 2024



> **Gegen den wachsenden Kontext gibt es einen Werkzeugkasten: zurücksetzen, zusammenfassen, in Subagents auslagern, ein dauerhaftes Gedächtnis führen und das Fenster bewusst kleiner halten.**

Der Heuhaufen von den vorigen Seiten wächst mit jedem Turn; die Folie sammelt die Werkzeuge, um ihn klein zu halten. „/clear“ setzt zwischen unabhängigen Aufgaben komplett zurück und baut den Kontext frisch auf. „/compact“ fasst die History zusammen — die wichtigste Regel steht daneben: proaktiv bei 50–60 %, nicht erst bei 95 %, wenn ohnehin schon aus verrauschtem Kontext zusammengefasst wird.

Subagents arbeiten mit eigenem Kontextfenster für recherche-lastige Sub-Tasks; „CLAUDE.md“ bzw. „AGENTS.md“ bilden ein dauerhaftes „Gedächtnis“, das jede Kompaktierung überlebt. Das Fenster von 1M auf 200k zurückzudrehen bringt oft höhere Qualität und geringere Kosten.

Der letzte Punkt ist umstritten: Das Amp-Team formuliert die Gegenposition „You should basically never use compaction.“ — lieber kurze Threads und saubere Handoffs als jede Zusammenfassung. Beide Lager eint das Ziel, den Kontext gar nicht erst degradieren zu lassen.

> **In der Live-Demo baut der Agent per opencode eine Todo-App, während der Proxy jeden Request mitschneidet — und der Cache ab Request 2 auf 99 % springt.**

Die vier Kacheln zeigen den Ablauf der Demo: opencode legt im Proxy-Logging-Workspace eine Todo-App an („erzeuge eine todo app"), die fertige App wird über die Port-Subdomain geöffnet, und „detailed docs" landen im Viewer — mit Payload-Wachstum, Tool-Chips und aufklappbaren Sessions.

Der Kern steht in Kachel 4: Klappt man eine Session auf, zahlt Request 1 den Kontext voll (0 %), ab Request 2 kommen 99 % aus dem Cache — die Theorie von Seite 21 wird anfassbar. Cold-Start-Vorbehalt: Weil Claude Codes Cache-TTL 1 Stunde und global ist, zeigt Request 1 nach einem Lauf in der letzten Stunde keine 0 %, sondern Teil-Cache.

Bei Zeitnot entfällt die Live-Demo zugunsten der Backup-Folie „Cache-Aufbau, echt gemessen" (Seite 30) — die Kernaussagen stehen dann bereits. Wer im Viewer nie getippte „<block>"-Sessions sieht, erkennt den Auto-Modus: drei gemessene Guard-Calls zu je rund 39.000 Token, die nur der Proxy zeigt.

Messdaten: eigene Captures 07.–08.07.2026 (docs/WORKSHOP-CAPTURES.md, docs/FACTS.md)

> **Ab dem zweiten Request zahlt der Agent nur noch den neuen Kontext-Rest — bis eine Pause länger als die Cache-TTL alles wieder auf null setzt.**

DIE ZAHLEN DER FOLIE

#1 — 9.857 / 0 % — Der erste Request zahlt den ganzen Kontext voll: 9.857 Prompt-Tokens, nichts aus dem Cache — es gibt noch keinen warmen Präfix.

#2 — 88 / 99 % — Ab Request 2 greift der KV-Cache: von 10.072 Kontext-Tokens sind nur 88 wirklich neu, 99 % kommen aus dem Cache.

#6 — 143 / 12.303 — Der Kontext ist auf 12.303 Tokens gewachsen, voll bezahlt werden aber nur 143 — der Rest liegt im Cache.

9.857 → 13.717 — Über neun Requests wächst der Kontext von 9.857 auf 13.717 Tokens; ab Request 2 liegt die Cache-Quote bei 94–99 %.

neu #1 — 14.116 / 0 % — Neue Session nach einer Pause länger als die Cache-TTL (Claude Code: 1 h): Request 1 wieder 0 % Cache, alle 14.116 Tokens neu bezahlt.

HINTERGRUND

Diese Folie ist der Fallback für die interaktive Live-Demo (Todo-App → Viewer): Klemmt das WLAN oder ein Tool, zeigt sie denselben Effekt statisch — echte Zahlen aus einem OpenCode-Lauf mit dem Prompt „erzeuge eine todo app“. Der Cache-Mechanismus (Prefill/KV-Cache, Seite 21) sorgt dafür, dass ab dem zweiten Request nur noch der neue Kontext-Rest bezahlt wird.

Der Haken ist die Cache-Lebensdauer. „Voll bezahlt“ meint Kontext minus Cache-Read, und ein Cache-Read kostet nur rund ein Zehntel des normalen Preises. Wer länger pausiert als die TTL, startet mit 0 % Cache und bezahlt den kompletten Kontext erneut — die (lange) Pause erhöht buchstäblich die Rechnung.

Messdaten: eigene Captures 07.–08.07.2026 (OpenCode/big-pickle, Prompt „erzeuge eine todo app“)

> **Konfiguration ist Teil der Ausführungsschicht: Dieselbe Klasse „Env-Variable steuert das Traffic-Ziel", die den Proxy trägt, war 2026 zweimal Angriffsvektor — beide Male, bevor der Trust-Dialog erscheint.**

Eine Umgebungsvariable bestimmt, wohin der API-Traffic geht — bei diesem Proxy ist das HTTP_PROXY. Genau diese Klasse, in der eine Env-Variable das Traffic-Ziel steuert, wurde 2026 zweimal zum Angriffsvektor, beide Male, bevor der Trust-Dialog überhaupt erscheint.

CVE-2025-59536 (CVSS 8.7) ist Remote Code Execution: Ein manipuliertes Repo bringt eine .claude/settings.json mit, deren Hooks und MCP-Autostart beim bloßen Öffnen des Repos Code ausführen. CVE-2026-21852 (CVSS 5.3) ist Key-Exfiltration: Das Repo setzt ANTHROPIC_BASE_URL um, der API-Key geht im Klartext an den Angreifer.

Beides ist in Claude Code gepatcht (v1.0.111 / v2.0.65) — aber in bestimmten Versionen, nicht im Modell; wer eine alte fährt, bleibt verwundbar. Die Angriffsfläche „Config = Execution" bleibt tool-übergreifend, ein fremdes Repo ist wie fremder Code zu behandeln. Der Twist: Wer die BASE_URL selbst kontrolliert — eigener Gateway —, macht aus dem Vektor ein Audit-Werkzeug.

CVEs verifiziert 07.07.2026: Check Point Research / Tenable

> **Input-Kompression spart real Tokens — aber wer den Input zu stark beschneidet, zwingt das Modell zum Raten und zahlt am Ende mehr.**

DIE ZAHLEN DER FOLIE

`4.823 → 11 Tokens` — RTK komprimiert den Output von cargo test mit 262 grünen Tests auf 11 Tokens — per Hook, bevor er ins Kontextfenster gelangt.

`~89 % Reduktion` — Schnitt über 2.900 Befehle: RTK („Rust Token Killer“) schrumpft Shell-Output um rund 89 Prozent, greift aber nur bei Bash-Calls.

`+18 % Gesamtkosten` — Die Pointe (RTK Issue #582): Trotz weniger Input stiegen die Gesamtkosten — dokumentiert vom RTK-Projekt selbst als Einzelbericht.

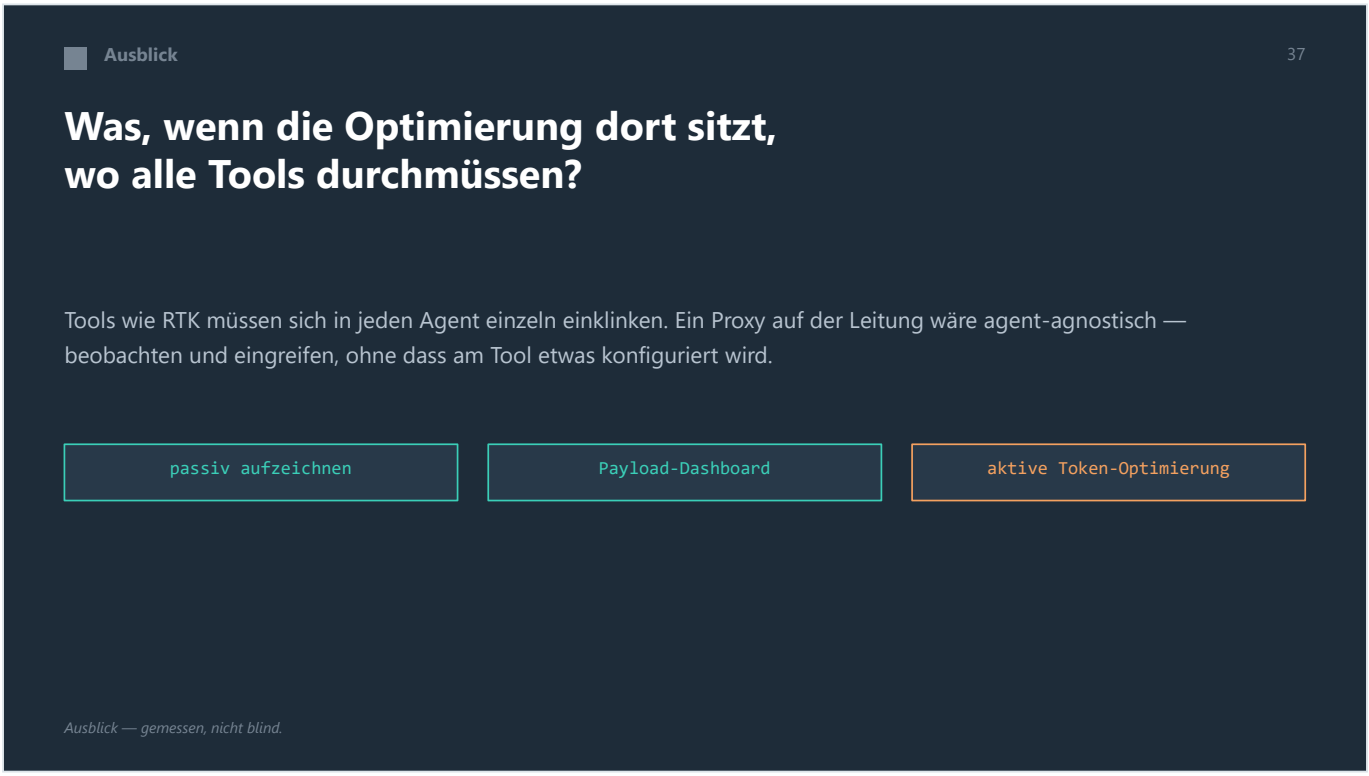
`~50 % mehr Output` — Weniger Input zwang das Modell, fehlende Information selbst zu ergänzen — es produzierte rund die Hälfte mehr Output und fraß die Ersparnis auf.

HINTERGRUND

Die linke Spalte zeigt, dass Kompression funktioniert: Ein Hook fängt den Shell-Output ab, bevor er ins Kontextfenster wandert, und schrumpft ihn — im Schnitt über 2.900 Befehle um rund 89 Prozent.

Die Pointe steht rechts. Genau dieses Kürzen ließ im dokumentierten Fall die Gesamtkosten steigen, weil das Modell die gestrichene Information über mehr Output rekonstruierte. Die Lehre für den Ausblick: Beschneide den Input nie so stark, dass das Modell raten muss, und miss immer die Gesamt-Tokens — Input und Output.

Quelle: RTK-Projekt („Rust Token Killer“), Issue #582 — Einzelbericht; Kompressionswerte laut Folientext.



> **Ein Proxy auf der Leitung — dort, wo alle Tools durchmüssen — könnte agent-agnostisch beobachten und eingreifen, ohne dass am einzelnen Tool etwas konfiguriert wird.**

Die Folie stellt eine Frage: Was, wenn die Optimierung dort ansetzt, wo alle Tools ohnehin durchmüssen? Werkzeuge wie RTK komprimieren zwar, müssen sich aber in jeden Agent einzeln einklinken. Ein Proxy auf der Leitung säße stattdessen agent-agnostisch dazwischen — er beobachtet und greift ein, ohne dass am Tool etwas konfiguriert wird.

Die drei Stufen unter der Frage skizzieren den Ausbaupfad: erst passiv aufzeichnen, dann ein Payload-Dashboard, schließlich aktive Token-Optimierung. Vom reinen Mitschneiden über die Sichtbarmachung bis zum Eingreifen auf der Leitung — jede Stufe baut auf der vorigen auf.

Der Fußtitel „gemessen, nicht blind“ ist die Bedingung: Eingriffe rechtfertigen sich nur mit Messung. Das RTK-Beispiel zuvor hat gezeigt, dass zu aggressives Kürzen nach hinten losgehen kann — Input und Output müssen beide gemessen werden, bevor der Proxy selbst optimiert.

Dein Coding Agent · Fazit 30

Was heißt das für dich?

01 Das Modell ist stateless — alles „Gedächtnis“ ist Tool-Leistung. Versteh dein Tool.

02 Ein Proxy zeigt dir, was dein Tool wirklich sendet — schnell selbst gebaut oder fertig genutzt (XaresAICoder).

03 Context-Management ist die #1-Fähigkeit: öfter bei null anfangen, Subagents, Markdown-Gedächtnis (AGENTS.md).

04 Caching entscheidet die Rechnung — Hit-Rate im Auge behalten (gemessen: 45–90 %).

05 Token-Verbrauch: Faktor 4 zwischen Tools — Architektur- UND Modellwahl sind Kosten-Entscheidungen.

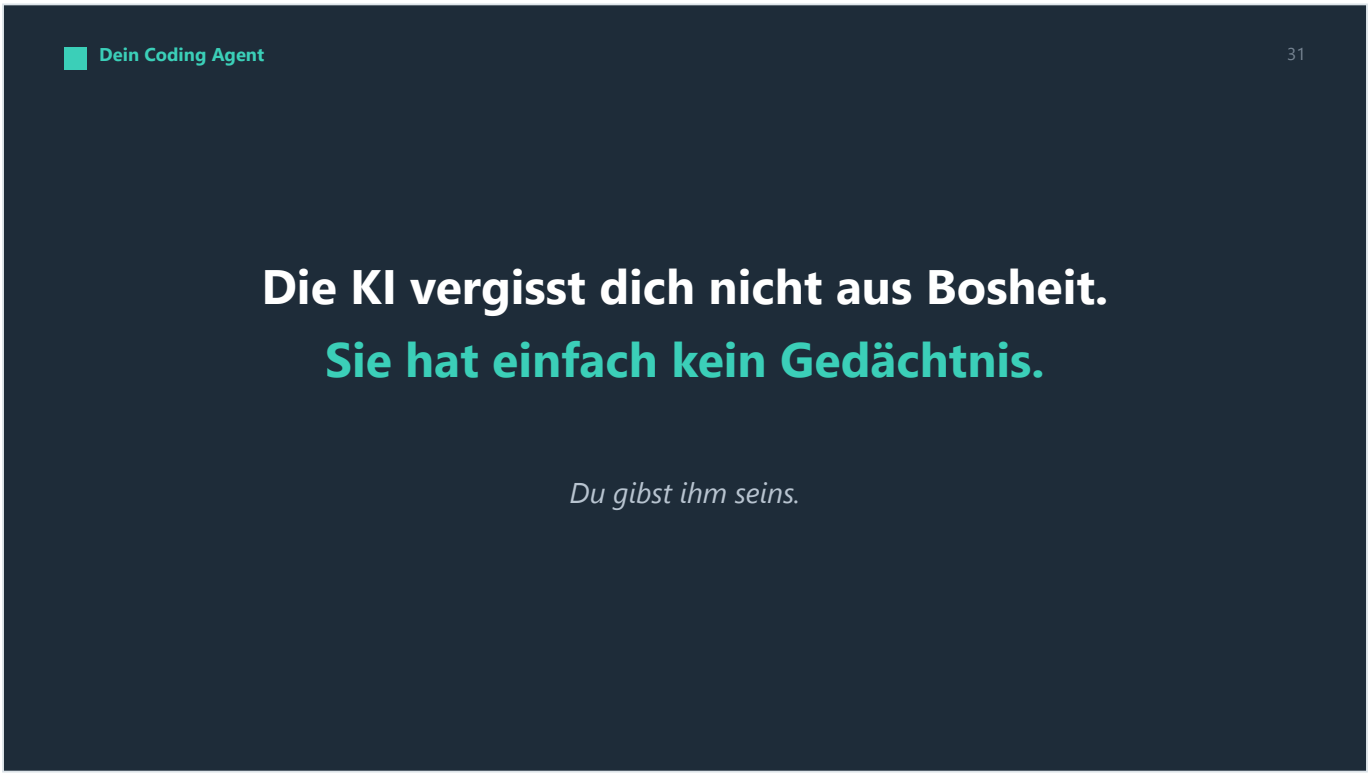
> **Fünf Mitnahmen, alle mit eigenen Messwerten belegt: Wer versteht, was sein Tool an das zustandslose Modell schickt, steuert Kontext, Kosten und Risiko selbst.**

Die Folie fasst den Vortrag in fünf Mitnahmen, jede mit eigenen Messwerten belegt. Bleibt nur eine hängen, dann diese: Das Modell ist zustandslos — jedes „Gedächtnis“ ist Leistung des Tools, nicht des Modells. Wer versteht, was das eigene Tool an das Modell schickt, kann Kontext steuern, Kosten kontrollieren und Risiken einschätzen.

Die vier weiteren Punkte konkretisieren das: Ein Proxy zeigt, was wirklich gesendet wird — schnell selbst gebaut oder fertig genutzt (XaresAICoder). Context-Management ist die wichtigste Fähigkeit: öfter bei null anfangen, Subagents, Markdown-Gedächtnis (AGENTS.md). Caching entscheidet die Rechnung — gemessen wurden Hit-Raten von 45–90 % (Cache-Mechanik: Seite 21). Und beim Token-Verbrauch liegt Faktor 4 zwischen den Tools bei identischer Aufgabe, sodass Architektur- und Modellwahl Kosten-Entscheidungen sind.

Der rote Faden: Der Agent lügt nicht — er vergisst. Sein Verhalten ist erklärbar, sobald man die Mechanik dahinter kennt. Genau dafür ist der Proxy da — er macht die Blackbox nachprüfbar, statt ihr blind zu vertrauen.

Messdaten: eigene Captures 07.07.2026 (WORKSHOP-CAPTURES.md)



> **Die KI vergisst dich nicht aus Bosheit — sie hat schlicht kein Gedächtnis; welchen Kontext der Agent bei jedem Aufruf bekommt, entscheidest du.**

Der Schlusssatz schließt den Bogen zum Titel „Dein Coding Agent“. Was auf der Folie steht, ist die Kernthese des ganzen Vortrags in drei Zeilen: Ein Sprachmodell ist auf API-Ebene zustandslos — jeder Aufruf beginnt bei null, ohne Erinnerung an den vorigen. Das ist keine Absicht und kein Fehler, sondern die Bauart.

„Du gibst ihm seins“ heißt: Das Gedächtnis, mit dem der Agent arbeitet, ist der Kontext, den du bei jedem Call mitschickst. Genau darum drehte sich der Vortrag — von der Harness (Seite 15), die den Kontext zusammenbaut, über Prefill und KV-Cache (Seite 21) bis zu den Token-Messungen, die zeigen, wie unterschiedlich Werkzeuge diesen Kontext füllen.

Wer das versteht, hört auf, dem Modell Boshaftigkeit zu unterstellen, und fängt an, seinen Kontext bewusst zu gestalten: knapp, relevant, cache-freundlich. Das ist der Hebel, der in der Hand des Nutzers liegt.

Danke!

32

Fragen?
Und: schaut euch gerne mal den XaresAICoder an — Proxy inklusive.

github.com/DG1001/XaresAICoder

Frederik Wystup · MeiLuft · Java Forum Stuttgart 2026

> **Der XaresAICoder mit eingebautem Proxy liegt offen auf GitHub — ein Werkzeug zum Verstehen dessen, was ein Coding Agent wirklich sendet, kein täglicher Begleiter.**

Danke für das Interesse. Wer die Mechanik aus diesem Vortrag selbst nachvollziehen will, probiert den XaresAICoder aus: Der eingebaute Proxy macht sichtbar, was sonst im Verborgenen über die Leitung geht — jeder Request, die Cache-Marker (Seite 21), die versteckten Guard-Calls im Auto-Modus.

Gedacht ist er zum Verstehen, nicht als täglicher Begleiter — das wird er erst, wenn er zum Protokoll- und Governance-Werkzeug ausgebaut ist. Fragen zum Code gern per Mail.

Repo: github.com/DG1001/XaresAICoder · Frederik Wystup, MeiLuft, Java Forum Stuttgart 2026