Das Projekt, das deine KI nie vergisst

Warum deine KI fünfmal mehr kostet, als sie müsste — und wie eine Handvoll einfacher Textdateien das behebt

Ein Verzeichnis.
Eine Datei.
Fertig.

Die Grundidee stammt von **Jake Van Clief** und seiner Interpretable Context Methodology.
Kostenlos. Weitergeben ausdrücklich erwünscht. · ebook.quizsparkhub.com

Diese Seite zuerst

Du musst nichts bauen.

Hinten im Buch steht ein Prompt. Du fügst ihn in Claude, Codex oder was du sonst benutzt ein, beantwortest drei Fragen, und zwanzig Minuten später existiert das ganze System. Bis dahin musst du keinen einzigen Befehl verstehen.

Warum dann ein Buch?

Weil ein System, das du nicht verstehst, ein System ist, das du irgendwann still kaputtmachst. Du verschiebst eine Datei, löschst etwas, das nutzlos aussieht, ignorierst eine Warnung — und sechs Wochen später funktioniert nichts mehr und du weißt nicht, warum.

Dieses Buch ist dafür da, dass du verstehst, was deine KI dir gebaut hat — und warum jedes Teil genau so aussieht, wie es aussieht.

Jeder Teil dieses Systems existiert, weil vorher etwas schiefging. Ich zeige dir, was. Das ist die ehrliche Art, einen Aufbau zu erklären — nicht „hier sind die Regeln", sondern „hier ist der Unfall, der diese Regel nötig gemacht hat".

Das Buch folgt der tatsächlichen Geschichte — woher die Idee kam, was passierte, als sie auf ein echtes System traf, und erst danach, wie das Ergebnis funktioniert.

Teil	Worum es geht
1	Das Problem, und was es dich wirklich kostet
2	Wo das angefangen hat — Jake Van Cliefs ICM
3	Was passierte, als es auf ein laufendes System traf
4	Eine Frage Schritt für Schritt durch das fertige System verfolgen
5	Warum jedes Teil genau so ist, wie es ist
6	Was passiert, wenn dein Projekt wächst
7	Von Hand aufsetzen — *optional, nur wenn du willst*
8	Die KI anschließen
Anhang	Die Prompts. Das ist, was die meisten benutzen werden

Teil 1 bis 6 sind das Buch. Teil 7 gibt es, damit nichts eine Blackbox bleibt — überspringen kostet dich nichts.

Alles hier stammt aus einem echten Projekt: einer Software namens QuizSpark Hub, fast vollständig von KI gebaut. Die Zahlen sind gemessen, nicht geschätzt. Die Fehler sind echt, mehrere davon meine.

Und die Grundidee stammt nicht von mir. Sie kommt von Jake Van Clief. Teil 2 handelt von ihm und seiner Methode, und er steht dort, weil man den Rest ohne ihn nicht versteht — alles Weitere ist entweder seine Idee oder etwas, das dazukommen musste, als seine Idee auf ein System traf, das sich unter einem bewegt.

Teil 1 — Das Problem

1. Der Assistent ohne Gedächtnis

Stell dir vor, du stellst einen großartigen Assistenten ein. Wirklich großartig — schnell, klug, nie müde.

Ein Haken: Jeden Morgen hat er alles vergessen.

Also erklärst du jeden Tag von vorn. Was das Projekt ist. Wie es funktioniert. Was du schon probiert hast. Was letztes Mal kaputtging. Warum diese eine seltsame Sache so ist, wie sie ist. Wenn du fertig bist, ist der halbe Tag weg. Dann macht er zwanzig Minuten hervorragende Arbeit und geht nach Hause.

Morgen erklärst du alles noch einmal.

So ist die Arbeit mit einer KI. Nicht weil sie dumm wäre — sondern weil sie jedes Gespräch mit leerem Kopf beginnt.

Also tun die Leute das Naheliegende: Sie fügen alles ein. Das ganze Projekt, alle Notizen, die komplette Geschichte. Und es funktioniert. Es kostet auch ein Vermögen und wird still schlechter, je größer das Projekt wird.

2. Was es tatsächlich kostet

Fast niemand misst das nach. Was dabei herauskommt, wenn man es tut:

Eine KI liest Text in kleinen Häppchen, sogenannten **Tokens**. Grob gesagt ist ein Token drei Viertel eines Wortes. Du zahlst pro Token, und es gibt eine Obergrenze — das *Kontextfenster* —, wie viel sie gleichzeitig im Kopf behalten kann.

In dem echten Projekt hinter diesem Buch gab es eine Regeldatei, die die KI lesen musste, bevor sie überhaupt irgendetwas tun konnte. Sie war 36 Kilobyte groß. Etwa **10.000 Tokens**. Jede Sitzung. Jede Frage. Bevor auch nur ein nützliches Wort herauskam.

Und was tat diese Datei?

Sie sagte der KI, **welche andere Datei** sie lesen soll. Das war ihre ganze Aufgabe. Ein Wegweiser.

Die eigentliche Antwort — die Datei, die erklärte, wie die Sache funktioniert — war 3 bis 4 Kilobyte groß. Etwa 1.000 Tokens.

> **Neun von zehn Tokens gingen an den Wegweiser, nicht an die Antwort.**

Neunzig Prozent der Kosten jeder Frage waren Wegfindung. Kein Nachdenken. Keine Antwort. Nur herausfinden, wo man nachschauen muss.

Wenn du diese Zahl einmal gesehen hast, folgt der gesamte Aufbau dieses Systems daraus.

3. Warum „einfach alles einfügen" scheitert

Drei Gründe, und alle drei werden schlimmer, je größer du wirst.

Es kostet jede Woche mehr. Dein Projekt wächst, der eingefügte Text wächst, deine Rechnung wächst. Was du davon hast, wächst überhaupt nicht.

Es geht irgendwann nicht mehr. Jede KI hat eine Obergrenze. Sobald dein Projekt größer ist als das Fenster, steht „alles einfügen" nicht mehr zur Verfügung. Jetzt musst du auswählen, was wegbleibt — und du hast keine Methode dafür, also wählst du schlecht.

Die Antworten werden schlechter. Das glauben die meisten am wenigsten. Vergraben in einem Textberg holt eine KI *eher* etwas Veraltetes hervor — etwas, das vor einem halben Jahr stimmte, etwas aus einer Notiz, die du zu löschen vergessen hast.

Mehr Kontext ist nicht mehr Genauigkeit. Ab einem Punkt ist es weniger. Du hilfst ihr nicht beim Denken; du gibst ihr mehr Gelegenheiten zu stolpern.

Teil 2 — Wo das angefangen hat

Nichts davon begann mit einem Plan. Es begann mit einem YouTube-Video.

Das Video

„Stop Building AI Agents. Use This Folder System Instead." von **Jake Van Clief**.

Der Titel ist das Argument. Damals baute jeder, der mit KI arbeitete, Agenten — Frameworks, Orchestrierungscode, Werkzeuge, die Werkzeuge aufrufen. Seine Behauptung: Das meiste davon ist überflüssig, und was man tatsächlich braucht, ist eine Ordnerstruktur.

Seine Methode heißt **ICM — Interpretable Context Methodology**. Der Kern passt in eine Zeile:

> **Ordnerstruktur ersetzt Orchestrierung.**

Statt Code zu schreiben, der einer KI sagt, was sie wann tun soll, ordnest du Ordner und Markdown-Dateien — und die Anordnung selbst ist die Anweisung. Kein Framework. Keine Orchestrierungsschicht. Das Dateisystem ist die Architektur.

Der Teil, der es funktionieren lässt

Sein Routing hat **drei Schichten**, und jede beantwortet genau eine Frage:

Schicht	Datei	Frage
L0	Wurzel-`CLAUDE.md`	Welcher Arbeitsbereich ist zuständig?
L1	Arbeitsbereichs-`CLAUDE.md`	Welche Dateien braucht dieser Bereich — und wann?
L2	`CONTEXT.md` der Stufe	Was genau soll die KI in diesem Schritt tun?

Jede Schicht **verengt** den Kontext.

L0 verhindert, dass die KI in dreißig Arbeitsbereiche schaut, wenn einer genügt. L1 verhindert, dass sie Dateien lädt, die sie erst in Schritt vier braucht — seine L1-Tabellen haben drei Spalten: Ressource, Wann und Warum. L2 gibt ihr eine begrenzte Aufgabe mit vier Feldern: Input, Prozess, Output, Fertigstellung.

Sein Grundsatz, in seinen Worten:

> *KI-Agenten arbeiten besser, wenn sie engen, passenden Kontext bekommen statt breitem, allgemeinem.*

Zielgröße pro Schritt: **200 bis 400 Zeilen statt Tausender.**

Zwei weitere Details, die wichtiger sind, als sie aussehen:

Namenskonventionen ersetzen eine Datenbank. Arbeitsbereiche klein geschrieben und mit Bindestrich. Stufen nummeriert als `NN-name`, mit führender Null. Du brauchst kein Verzeichnis darüber, was existiert, weil die Namen *das* Verzeichnis sind.

Ausdrückliche Ausschlusstabellen. Nicht nur „lies das", sondern „lies jenes nicht". Ohne sie stöbert eine KI herum, und Herumstöbern ist das, was deine Rechnung aufbläht.

Warum es eingeschlagen hat

Rund 30.000 Menschen arbeiten inzwischen so. Es gibt eine Community mit über 16.000 Mitgliedern und ein wissenschaftliches Papier.

Es hat eingeschlagen, weil es den üblichen Reflex umkehrt. Der übliche Reflex, wenn eine KI dein Projekt nicht versteht, ist: **gib ihr mehr**. Seine Antwort ist: gib ihr **weniger, aber geordnet**.

Auf dieser Idee ruht dieses ganze Buch. Alles nach diesem Kapitel ist das, was passierte, als jemand sie genommen und auf ein laufendes, kommerzielles System gerichtet hat.

Wenn du die Quelle willst, geh zu ihm. Er erklärt es besser, als eine Zusammenfassung kann — und er erklärt es für den Fall, für den er es entworfen hat.

Teil 3 — Was passierte, als es auf ein laufendes System traf

Die Methode war für Abläufe gebaut: recherchieren, dann entwerfen, dann prüfen. Nummerierte Stufen, eine KI, die sie durchläuft.

Worauf sie gerichtet wurde, war etwas anderes. Ein laufendes Produkt mit zahlenden Kunden — Datenbank, Hintergrundjobs, Zahlungsabwicklung, Videoerzeugung, E-Mail. Nichts in Stufen. Alles gleichzeitig, dauernd, in Bewegung.

Drei Dinge passierten, in dieser Reihenfolge.

Was unverändert übernommen wurde

Das Routing-Prinzip griff sofort — und zwar aus genau dem Grund, den er nennt.

Das Projekt hatte eine Regeldatei, die die KI vor jeder Aufgabe las. 36 Kilobyte. Sie zu teilen — ein winziger Router zum *Finden*, eine getrennte Datei für *Regeln zum Ändern* — senkte die Kosten einer Frage um den Faktor fünf. Gemessen, nicht geschätzt.

Das ist seine Idee, die genau das tut, was er angekündigt hat, an einem System, das er nie gesehen hat.

`CLAUDE.md` als Einstiegspunkt: von ihm. Die kleine Routing-Tabelle ganz vorn: von ihm. Der Grundsatz, dass eine KI eng liest und dann aufhört: von ihm.

Die Spuren sind im echten Projekt bis heute sichtbar. Vier Dateien namens `CONTEXT.md`, eine pro Bereich. Ein Ordner mit nummerierten Arbeitsanweisungen — `1-…`, `10-…`, `14-…`. Das ist sein Namensschema, unverändert.

Das Erste, das nicht passte

ICM routet nach **Arbeitsstufen**. Ein laufendes Produkt hat keine Stufen.

`01-discover`, `02-drafting`, `03-review` beschreibt einen Ablauf — etwas mit Anfang und Ende, wo Schritt zwei nach Schritt eins kommt.

Login, Abrechnung, Videoerzeugung, E-Mail: Das sind keine Schritte. Die existieren alle gleichzeitig. Niemand fragt „in welcher Stufe bin ich". Man fragt „welchen Teil des Produkts meinst du".

Also wurde aus dem Stufen-Router ein **Themen-Router**. Dasselbe Prinzip — eine kleine Datei, Kontext verengen, eine Sache lesen und aufhören — nur entlang einer anderen Achse.

Das war eine kleine Änderung. Die nächste war es nicht.

Das Zweite, das alles veränderte

Hier ist die Weggabelung, und sie lohnt sich langsam zu lesen.

ICM organisiert Anweisungen. Was soll die KI tun, womit, in welcher Reihenfolge. Anweisungen werden einmal geschrieben und bleiben richtig, bis du beschließt, sie zu ändern.

Ein laufendes System braucht zusätzlich Zustandsbeschreibungen. Wie funktioniert die Abrechnung *gerade*? Welche Tabellen gibt es *heute*? Was bedeutet dieser Fehler eigentlich?

Und Zustand tut etwas, das Anweisungen nie tun: **Er veraltet von selbst.**

Niemand bearbeitet eine Seite, um sie falsch zu machen. Jemand ändert das System, und die Seite — unangetastet, immer noch selbstbewusst, immer noch ausführlich — hört still auf zu stimmen. Nichts kündigt das an. Es gibt keine Fehlermeldung für ein Dokument, das eine Welt beschreibt, die weitergezogen ist.

Das ist das gesamte Problem, das der Rest dieses Buches löst — und es ist ein Problem, das ICM nie lösen musste, weil Anweisungen nicht von allein verrotten.

Was dazukommen musste, und was jede Erkenntnis gekostet hat

Jede Ergänzung unten entstand, weil etwas schiefging. Teil 5 erzählt jede Geschichte richtig; hier ist die Form davon.

Ergänzung	Was sie erzwungen hat
Datum auf jeder Seite	Seiten waren Monate alt, und nichts zeigte es an — weder einem Menschen noch der KI
Maschinenlesbares Inventar	der Prüfer braucht eine Liste zum Vergleichen; Fließtext lässt sich nicht mit der Wirklichkeit vergleichen
Prüfen in beide Richtungen	Seiten beschrieben Dinge, die es nicht mehr gab. Nichts fehlte, also sah nichts falsch aus
Zahlen im Fließtext prüfen	„27 Tabellen" stand an drei Stellen, als es 36 waren
Abschnitt „Fallstricke"	Stunden, die eine stille Fehlkonfiguration gekostet hat, stehen in keiner Quelle der Welt
Erzwingen im Sicherungsschritt	eine KI befolgte jede einzelne Regel und sicherte die Arbeit dann nie

Schau die Liste noch einmal an. Jeder Punkt dreht sich um dieselbe Sache: **Woher weißt du, dass die Seite noch stimmt?**

Das ist die Frage, die ein lebendes System dir aufzwingt. Und deshalb hat dieses System einen Prüfer, der als Programm läuft und Liste mit Liste vergleicht — statt einer regelmäßigen Durchsicht.

Das ist keine Kritik an ICM

Man könnte den letzten Abschnitt als „seine Methode war unvollständig" lesen. War sie nicht.

Er hat eine Methode gebaut, um Anweisungen an KIs zu ordnen, und dafür ist sie vollständig. Die Ergänzungen hier sind Antworten auf eine Frage, die sich in seinem Fall nicht stellt. Wenn deine Arbeit aussieht wie seine — Abläufe, Stufen, wiederholbare Vorgänge — brauchst du davon vielleicht nichts, und es hinzuzufügen wäre Aufwand ohne Gegenwert.

Nimm das Routing. Ergänze die Zustandsprüfung nur, wenn sich dein Boden bewegt.

Ein Dritter, der von woanders herkam

Im April 2026 veröffentlichte **Andrej Karpathy** ein Muster namens **LLM Wiki**. Anderer Ausgangspunkt: Er dachte über *Quellen* nach — Artikel, Papers, Forschungsmaterial — nicht über Agenten.

Seine Antwort: Statt bei jeder Frage in Rohquellen zu suchen, lässt man die KI sie **einmal** in ein verlinktes Markdown-Wiki kompilieren. Jede weitere Frage geht ans Wiki.

Seine Struktur wird dir inzwischen bekannt vorkommen. Eine `index.md` als Katalog. Eine `log.md`, rein chronologisch. Verlinkte Seiten. Eine Schema-Datei mit den Konventionen. Und ein regelmäßiger **Lint**-Durchgang gegen Widersprüche, überholte Aussagen und verwaiste Seiten.

Sein Unterschied zu diesem Buch ist derselbe, nur von der anderen Seite gesehen: **Seine Rohquellen ändern sich nicht.** Ein Papier, das im März erscheint, sagt im Dezember dasselbe. Also genügt ein Lint, den die KI selbst durchführt, indem sie liest und urteilt.

Für ein laufendes System genügt das nicht, weil sich der Boden bewegt — daher ein Programm statt eines Urteils.

Aber dieser Vergleich schneidet in beide Richtungen, und die ehrliche Fassung lautet: Die automatische Prüfung hier ist nur möglich, weil es etwas gibt, das man *fragen* kann. „Welche Tabellen existieren gerade?" hat eine abfragbare Antwort. In Karpathys Bereich gibt es diese Frage nicht. Er könnte diesen Prüfer nicht bauen, selbst wenn er einen wollte. Das ist kein Vorsprung. Es ist ein anderer Problemtyp.

Wo dich das hinstellt

Drei Menschen, aus drei Richtungen, mit derselben zugrunde liegenden Erkenntnis:

> **Der teure Teil ist nicht das Denken. Es ist das Suchen. Also kompiliere einmal und frage danach nur noch das Kompilat.**

Wenn eine Idee dreimal unabhängig auftaucht, ist sie keine Geschmacksfrage mehr.

Der Rest dieses Buches ist die dritte Fassung davon — die für ein System, das nicht stillhält. Teil 4 zeigt dir, wie es funktioniert. Teil 5 erklärt, warum jedes Teil so geformt ist, wie es ist, ein Unfall nach dem anderen.

Teil 4 — Eine Frage durchs System verfolgen

Bevor wir zu Regeln kommen, schauen wir einfach zu. Eine Frage, von vorn bis hinten.

Sagen wir, du fragst: **„Wie funktioniert der Login?"**

Ohne das System

Die KI hat keine Ahnung, wo etwas liegt. Also tut sie, was sie kann: Sie liest viel. Deine Regeldatei, mehrere vielversprechende Dateien, vielleicht einen Ordner mit alten Notizen. Irgendwo darin findet sie die Antwort und sagt sie dir.

Bezahlt hast du alles. Und weil sie unterwegs alte Notizen gelesen hat, beschreibt ein Teil der Antwort mit einiger Wahrscheinlichkeit, wie der Login im letzten Frühjahr funktionierte.

Mit dem System

Schritt eins. Die KI öffnet eine kleine Datei — das *Inhaltsverzeichnis*. Etwa 4 Kilobyte. Es sieht so aus:

```
| Deine Frage betrifft …          | Lies          |
|---------------------------------|---------------|
| Login, Konten, Sicherheit       | auth.md       |
| Zahlungen und Abos              | abrechnung.md |
| E-Mail-Versand                  | email.md      |
```

Dazu drei kurze Regeln, zu denen wir später kommen.

Schritt zwei. Sie findet die passende Zeile. Login → `auth.md`. Sie öffnet diese eine Datei.

Schritt drei. Oben in `auth.md` steht ein Block namens *Kurzantwort* — drei Zeilen zu den drei Fragen, die Leute tatsächlich zum Login stellen.

Ist deine Frage eine davon, **hört die KI hier auf**. Gesamtkosten: ein kleines Inhaltsverzeichnis plus zwanzig Zeilen.

Schritt vier. Brauchst du mehr, liest sie in derselben Datei weiter. Die Abschnitte stehen immer in derselben Reihenfolge: wie eine Anfrage durchs System läuft, was gespeichert wird, die Fallstricke, wo die eigentlichen Dateien liegen.

Schritt fünf. Berührt deine Frage einen Nachbarbereich, steht unten ein Abschnitt *Hängt zusammen mit*, der die nächste Datei beim Namen nennt. Ein Sprung, keine Suche.

Das ist der ganze Mechanismus.

Was gerade passiert ist

Die KI hat **eine kleine plus eine mittlere Datei** gelesen. Nicht dein Projekt. Nicht deine Chronik. Keine alten Notizen.

Dieselbe Antwort. Etwa ein Fünftel der Kosten. Und — das wiegt schwerer als das Geld — **sie konnte nichts Veraltetes versehentlich lesen**, weil sie das alte Material nie geöffnet hat.

Der eine Satz zum Merken

Alles andere in diesem Buch dient diesem Ablauf. Wenn du dir sonst nichts merkst:

> **Die KI liest ein winziges Inhaltsverzeichnis, dann genau eine Datei, und hört auf.**

Der ganze Rest des Systems existiert, damit diese eine Datei die *richtige* ist und noch *stimmt*.

Teil 5 — Warum jedes Teil genau so ist

Jetzt der interessante Teil. Jede Entscheidung unten existiert, weil ohne sie etwas schiefging. Ich gebe dir die Regel und dann den Unfall.

Regel 1 — Eine Datei pro Bereich — niemals zwei

Die Regel: Jeder Bereich deines Projekts bekommt genau eine Datei, die beschreibt, wie er *heute* funktioniert.

Warum nicht zwei? Weil zwei Dateien über dieselbe Sache sich widersprechen werden. Nicht „vielleicht" — werden. Jemand aktualisiert die eine und nicht die andere, und schon hast du zwei Quellen, die einander widersprechen.

Und hier ist der eigentliche Schaden: **In dem Moment, in dem zwei Quellen sich widersprechen, sind beide wertlos.** Du kannst nicht entscheiden, welche stimmt, also musst du im echten System nachsehen — genau die teure Sache, die die Dokumentation dir ersparen sollte.

Eine Datei kann falsch sein. Zwei Dateien sind garantiert nicht vertrauenswürdig.

Warum „heute"? Weil „wie es früher war" und „wie wir es gern hätten" verschiedene Dokumente mit verschiedenen Zwecken sind. Vermischt kann ein Leser nicht erkennen, welcher Satz welcher ist. Deine KI auch nicht.

Regel 2 — Das Inhaltsverzeichnis muss winzig bleiben

Die Regel: unter etwa fünfzig Zeilen. Nichts darin außer Frage → Datei.

Warum so streng? Weil diese Datei *jedes einzelne Mal* gelesen wird. Jede Frage, jede Sitzung, für immer.

Damit ist sie die eine Datei, bei der Größe eine dauerhafte Steuer ist. Zehn zusätzliche Zeilen sind nicht zehn Zeilen — sie sind zehn Zeilen mal jede Frage, die du jemals stellen wirst, solange das Projekt existiert.

Der Unfall: Im echten Projekt waren Inhaltsverzeichnis und Regeln dieselbe Datei. Sie wuchs vernünftig, ein nützlicher Absatz nach dem anderen, bis sie 36 Kilobyte groß war. Niemand hat je etwas Unvernünftiges hinzugefügt. Es hat sich einfach angesammelt.

So zahlt man am Ende 10.000 Tokens, um herauszufinden, welche Datei man öffnen soll.

Die Lösung war, sie zu zweiteilen: eine winzige Datei zum *Finden*, eine getrennte für *Regeln zum Ändern*. Denn den Finder brauchst du jedes Mal. Die Regeln nur, wenn du wirklich etwas änderst.

Die Kosten pro Frage fielen um den Faktor fünf. Gelöscht wurde nichts. Es wurde nur dorthin verschoben, wo es hingehört.

Regel 3 — Jede Datei trägt ein Datum

Die Regel: Jede Seite beginnt mit dem Datum, an dem sie zuletzt geprüft wurde.

Warum: Ohne Datum kannst du „das ist aktuell" nicht von „das ist seit acht Monaten falsch" unterscheiden. Deine KI auch nicht — sie liest eine veraltete Seite mit voller Überzeugung und erzählt dir mit voller Überzeugung etwas Falsches.

Ein Datum verwandelt eine unbeantwortbare Frage („stimmt das noch?") in eine Einschätzung, die du tatsächlich treffen kannst („vor drei Tagen geprüft, vermutlich in Ordnung" gegen „letztes Jahr geprüft, das sollte ich nachsehen").

Der Unfall: Mehrere Dateien trugen Datumsangaben, die Monate älter waren als ihre letzte Änderung. Jemand hatte den Inhalt geändert und das Datum stehen lassen. Der Prüfer meldet jetzt jede Seite, die älter als sechzig Tage ist — nicht weil sechzig eine magische Zahl wäre, sondern weil *irgendetwas* nörgeln muss.

Regel 4 — Jede Datei hat dieselbe Form

Die Regel: sechs Abschnitte, immer dieselben, immer in derselben Reihenfolge.

```
Kurzantwort           drei häufige Fragen, je eine Zeile
Der Weg durchs System wo es beginnt, was es berührt, wo es endet
Datenmodell           was gespeichert wird und was wichtig ist
Fallstricke           was tatsächlich Zeit gekostet hat
Wo was liegt          Datei, Funktion, Seite, Tabelle — die Adressen
Hängt zusammen mit    Nachbarbereiche, mit Dateinamen
```

Warum identische Form? Damit ein Leser guten Gewissens früh aufhören kann.

Ist jede Datei anders aufgebaut, musst du sie ganz lesen, um sicher zu sein, dass du nichts übersehen hast. Ändert sich die Form nie, weißt du, dass oben eine Kurzantwort steht und unten die Adressen — also kannst du aufhören, sobald du hast, was du brauchst.

Vorhersehbarkeit ist das, was frühes Aufhören ungefährlich macht. Und frühes Aufhören ist die Stelle, an der die tägliche Ersparnis entsteht.

Regel 5 — Die Kurzantwort ganz oben

Warum es sie gibt: Die meisten Besuche in einer Datei sind keine Tiefenrecherche. Sie sind „Moment, welche war das noch?"

Drei Zeilen oben beantworten die, und der Leser — Mensch oder KI — schließt die Datei nach zwanzig Zeilen wieder.

Hier eine echte:

```
- Heilige Regel: Leads werden nie verworfen. Über der Grenze werden
  sie erfasst und nur GESPERRT.
- Wo wirkt die Sperre? In der Datenbank-Rechteregel, nicht in der
  Oberfläche — deshalb ist sie nicht umgehbar.
- Welche Endpunkte sind öffentlich? Drei, absichtlich — deshalb ist
  ein Rate-Limit noch offen.
```

Wer nur das liest, weiß bereits mehr als die meisten, die die ganze Datei gelesen haben.

Regel 6 — Der Abschnitt „Fallstricke" — hier steckt der eigentliche Wert

Die Regel: Schreib auf, was jemanden tatsächlich Stunden gekostet hat. Keine Theorie, keine Best Practice. Narben.

Warum das der wertvollste Abschnitt im ganzen System ist: Alles andere lässt sich im Prinzip wiederherleiten. Jemand kann den Code lesen und herausfinden, was er tut.

Die drei Stunden, die du verloren hast, kann niemand wiederherleiten. Dieses Wissen existiert an genau einem Ort — in deinem Kopf — und es verdunstet.

Drei echte:

> **Eine Ausschlussliste für Dateien akzeptiert keine Kommentare am Zeilenende.** `ordner/ # zu groß` erzeugt eine Regel für einen Ordner, der wörtlich `ordner/ # zu groß` heißt. Sie schließt nichts aus und scheitert lautlos. Der erste Versuch sicherte 470 Megabyte statt 8.
> **Dieselbe Ausschlussliste wirkt nur auf noch nicht erfasste Dateien.** Was bereits erfasst ist, bleibt erfasst, egal was die Regel sagt. Es muss ausdrücklich von Hand entfernt werden.
> **Ein Fehlercode bedeutete „das Gateway hat das Warten aufgegeben", nicht „die Arbeit ist gescheitert".** Die Arbeit war in Wahrheit erfolgreich fertig. Tage gingen dafür drauf, am falschen Ende zu suchen.

Nichts davon steht in irgendeinem Handbuch. Genau deshalb gehört es in deins.

Regel 7 — „Hängt zusammen mit" unten — was aus einem Stapel eine Landkarte macht

Die Regel: Jede Datei endet damit, ihre Nachbarn zu nennen.

Warum: Ohne das hast du zwanzig Dateien und keine Ahnung, welche du brauchst. Mit dem Abschnitt ist jede Datei eine Kreuzung — du landest irgendwo und kannst zur richtigen Stelle laufen.

Die praktische Wirkung: **von jeder Frage aus höchstens zwei Sprünge.** Einer ins Inhaltsverzeichnis, einer zum Nachbarn, falls du knapp danebenlagst.

Lässt du das weg, funktioniert das System technisch weiter, aber die Leute hören auf, ihm zu vertrauen — denn falsch raten kostet sie eine Suche, und nach zwei erfolglosen Suchen lesen sie wieder alles.

Regel 8 — Die Chronik ist getrennt — und zum Lernen verboten

Die Regel: Du führst ein Tagebuch darüber, was sich wann geändert hat und warum. Du liest es nie, um zu lernen, wie etwas funktioniert.

Warum überhaupt führen? Weil es verhindert, dass Entscheidungen neu aufgerollt werden. „Warum machen wir nicht X?" — weil wir X im März probiert haben, und hier steht, was passiert ist. Das ist es, was ein Projekt davon abhält, sich im Kreis zu drehen.

Warum zum Lernen verbieten? Zwei Gründe.

Sie wächst unbegrenzt. Die Chronik des echten Projekts umfasst inzwischen 131 Dateien. Sie zu lesen, um das System zu verstehen, heißt, jahrelange Änderungsnotizen zu lesen, um einen Ist-Zustand zu rekonstruieren, der zwei Ordner weiter im Klartext steht.

Sie ist voll von Dingen, die nicht mehr stimmen. Ein Eintrag vom März beschreibt den März. Er war richtig, als er geschrieben wurde, und ist heute irreführend. Nichts markiert ihn als abgelaufen, denn er war nie als aktuell gemeint.

Denk an ein Auto. Die Chronik ist das Serviceheft: „Bremsen im März gewechselt, sie quietschten." Die Bedienungsanleitung erklärt, wie die Bremsen funktionieren. Fünf Jahre Serviceheft zu lesen, um Bremsen zu verstehen, wäre langsam, verwirrend — und du glaubst am Ende Dinge, die seit 2019 nicht mehr stimmen.

Diese Regel steht im Klartext in der ersten Datei, die die KI liest. Nicht vergraben in einem Stilhandbuch — ganz vorn, wo man sie nicht übersehen kann.

Regel 9 — Ein maschinenlesbares Inventar

Die Regel: Neben den Seiten für Menschen führst du eine Datei, die auflistet, was im laufenden System tatsächlich existiert — jede Datenbanktabelle, jeden Endpunkt, jeden geplanten Job — in einer Form, die ein Programm lesen kann.

Warum: weil der Prüfer etwas zum Vergleichen braucht.

Fließtext lässt sich nicht automatisch mit der Wirklichkeit vergleichen. Eine Liste mit einer Liste schon. Das Inventar ist die Brücke zwischen „was läuft wirklich" und „was behaupten unsere Seiten".

Es ist eine Momentaufnahme, kein Live-Bild. Deshalb trägt es seine eigene Auffrisch-Anleitung bei sich: die konkreten Befehle, die in deinem Aufbau die Frage „was existiert gerade?" beantworten. Auffrischen dauert zwei Minuten, und alles Weitere hängt daran, dass es ehrlich ist.

Regel 10 — Der Prüfer prüft in *beide* Richtungen

Das ist das Teil, das die meisten nie bauen — und das wichtigste.

Richtung eins — was existiert, aber nirgends dokumentiert ist. Naheliegend und nützlich. Jemand baut etwas Neues und vergisst es aufzuschreiben; der Prüfer bemerkt, dass etwas im Inventar auf keiner Seite vorkommt.

Richtung zwei — was dokumentiert ist, aber nicht mehr existiert. Nicht naheliegend. Und genau hier verrottet Dokumentation.

Denk kurz darüber nach. Dokumentation stirbt selten dadurch, dass etwas fehlt — Fehlendes fällt auf. Sie stirbt, indem sie **eine Welt beschreibt, die weitergezogen ist**. Die Seite ist noch da, immer noch selbstbewusst, immer noch ausführlich — und still falsch. Niemand merkt es, weil nichts fehlt.

Der Unfall: Die erste Fassung dieser Prüfung meldete Fehlalarme. Sie hielt alles mit Bindestrich für einen möglichen toten Verweis — Speichernamen, Kennnummern, Stilnamen. Sie auf Zeilen zu verengen, die tatsächlich von Endpunkten sprechen, löste es. Wichtig zu wissen: **Eine Prüfung, die zu oft falschen Alarm schlägt, wird abgeschaltet** — und dann hast du gar keine.

Später dazugekommen: Zahlen im Fließtext. Die Seiten sagten an drei Stellen „27 Tabellen", als es 36 waren. Keine Strukturprüfung konnte das sehen — es waren bloß Wörter. Jetzt vergleicht der Prüfer jede Zahl über zwanzig mit dem Inventar und meckert bei Abweichung.

Die dritte Richtung, mit der ich nicht gerechnet habe

Ich habe oben „beide Richtungen" geschrieben und es geglaubt. Es gibt eine dritte, und die ist die gefährliche.

Richtung drei — eine Kopie, die dem Original *voraus* ist.

Das Projekt hatte drei Kopien (siehe Regel 12). Eine davon lag im laufenden Projekt, wo auch eine zweite KI arbeitete. Diese KI schrieb dort einen Tagebucheintrag. Und sie korrigierte dort eine Sicherheitsangabe — die Anzahl der Tabellen, die für den Browser bewusst unsichtbar sind. Die Seite in der **maßgeblichen** Kopie sagte weiterhin drei. Richtig waren acht.

Die richtige Fassung lag also in der Kopie, die falsche im Original.

Mein Prüfer meldete die ganze Zeit **„Structure OK"**. Nicht weil er kaputt war — sondern weil er auf meinem Rechner läuft und **das laufende Projekt gar nicht lesen kann.** Er sieht immer nur „lokale Datei neuer als die Kopie". Eine Datei, die in der Kopie entsteht, ist für ihn konstruktionsbedingt unsichtbar.

Beinahe hätte ich es schlimmer gemacht. Mein Reflex war: „Das Repository ist maßgeblich, also die Kopie von dort überschreiben." Das hätte eine richtige Sicherheitsangabe durch eine falsche ersetzt.

Die Regel daraus: Bevor du zwei Kopien abgleichst, vergleiche beide und sieh dir die Abweichung an. Lass niemals „welche ist maßgeblich" darüber entscheiden, was inhaltlich stimmt. Maßgeblich sagt, wer **einen Konflikt gewinnt**. Es sagt nicht, wer **recht hat**.

Und wenn es eine Kopie gibt, die kein Skript von dir lesen kann, dann leg **in diese Kopie** einen Hinweis, woher sie stammt und dass Änderungen dort zurück in die Quelle müssen. Dieser Hinweis ist das Einzige, was zwischen dir und einem stillen Verlust steht.

Regel 11 — Was der Prüfer bewusst nicht kann

Er kann nicht feststellen, ob du recht hast.

Er prüft, dass eine Seite existiert, ein Datum trägt, ihre Abschnitte hat, erreichbar ist und nichts beschreibt, das verschwunden ist. Ob deine Erklärung des Logins stimmt, dazu hat er keine Meinung.

Grün heißt „strukturell fehlt nichts". Es heißt nicht „das stimmt".

Ich betone das, weil die Fehlerart heimtückisch ist: Ein grüner Prüfer *fühlt sich* wie eine Freigabe an. Wenn du anfängst zu glauben, er bestätige Inhalte, hörst du auf, Fakten am echten System zu prüfen — und das Ganze wird zu einer sehr ordentlichen Art, selbstbewusst falsch zu liegen.

Die ehrliche Haltung steht in der Ausgabe des Prüfers selbst, bei jedem Lauf: *Das prüft nur Struktur. Die Fakten gegen das laufende System verifizieren.*

Regel 12 — Drei Kopien

Die Regel: die Kopie, an der du arbeitest, eine online, eine noch woanders.

Warum drei und nicht zwei? Zwei Kopien schützen vor einem Ausfall — dein Laptop stirbt, die Online-Kopie überlebt. Sind das aber deine einzigen zwei, kann ein einziger falscher Abgleich den Fehler auf beide übertragen.

Die dritte liegt meist an einem anderen *Typ* von Ort — ein Server, ein anderer Anbieter — und fällt damit aus anderen Gründen aus.

Der Unfall: Im echten Projekt war das überhaupt nicht gemacht. Alles existierte genau einmal, auf einem Laptop. Keine Historie, kein Backup, kein Rückweg. Monate Arbeit, eine Festplatte.

Das war das größte Risiko im ganzen System, und es hatte nichts mit Dokumentationsqualität zu tun. Gefunden wurde es zufällig, beim Blick auf etwas ganz anderes.

Die andere Seite dieser Regel, später und auf die harte Tour gelernt: Drei Kopien sind auch drei Orte zum Auseinanderlaufen.

Zwei davon waren Git-Remotes — ein Befehl schiebt in beide, sie können gar nicht auseinandergehen. Die dritte lag im laufenden Projekt, und **kein Skript konnte sie erreichen.** Nur eine KI, von Hand, über die Oberfläche des Projekts.

Also fiel sie zurück. Leise, zwei Tage lang, während alles grün meldete.

Eine Kopie, die nicht automatisch geschrieben werden kann, **läuft auseinander.** Die Lösung ist nicht Disziplin. Sie ist ein **Stempel**: eine Markierung, aus welchem Stand die Kopie gemacht

wurde, direkt daneben, und im selben Atemzug mitgepflegt. Dann wird aus „ist das aktuell?" eine Frage, die du in einer Sekunde beantwortest, statt ein Gefühl.

Und leg den Stempel an **beide** Orte — einen in die Quelle, der sagt, wie weit die Kopie nachgezogen wurde; einen in die Kopie, der sagt, woher sie stammt. Der zweite ist für den, der die Kopie öffnet, ohne die Quelle je zu sehen. In einem Projekt mit mehr als einer KI gibt es diesen Menschen.

Regel 13 — Der Einstiegspunkt liegt *im* Projekt

Die Regel: Die Datei, die einer KI sagt, wo sie anfangen soll, gehört ins Projekt selbst — nicht in einen Ordner daneben.

Warum: Eine Datei außerhalb des Projekts ist für jeden unsichtbar, der nicht du bist, auf deinem Rechner, heute.

Der Unfall — und der ist meiner. Ich habe den Einstiegspunkt in einem Ordner auf dem Rechner des Nutzers abgelegt, außerhalb des Projekts. Es funktionierte einwandfrei. Es war auch nicht gesichert, hätte einen verlorenen Laptop nicht überlebt und wäre bei einem Verkauf oder einer Übergabe nicht mitgegangen.

Schlimmer: Eine *andere KI* arbeitete direkt im laufenden Projekt, wo diese Datei nicht existierte. Sie konnte unmöglich Regeln befolgen, die sie nicht sehen konnte. Ich hatte ein System gebaut, das nur für mich funktionierte, auf einem Rechner — und es nicht bemerkt.

Der Einstiegspunkt gehört dorthin, wo gearbeitet wird. Wird an zwei Orten gearbeitet, gehört er an beide.

Regel 14 — Jede Regel nennt ihren Grund

Die Regel: Schreib nie „tu X". Schreib „tu X, weil Y".

Warum: Weil der Leser eine KI ist, die Anweisungen gegen eine Lage abwägt, die niemand vorhergesehen hat. Eine Anweisung mit Grund übersteht das. Eine ohne Grund fällt in dem Moment weg, in dem sie unpraktisch wirkt.

„Lade nie den Chronik-Ordner" lädt zur Ausnahme ein, sobald etwas zu fehlen scheint.

„Lade nie den Chronik-Ordner, weil er aus 131 Dateien veralteter Änderungsnotizen besteht und du ein Vermögen verbrennst, um Dinge zu lernen, die nicht mehr stimmen" tut das nicht.

Begründungen sind keine Verzierung. Sie sind der Durchsetzungsmechanismus für alles, was ein Programm nicht prüfen kann.

Regel 15 — Regeln und Erzwingen sind zwei verschiedene Dinge

Die Regel: Häng den Prüfer an einen Schritt, der ohnehin passieren muss — Sichern oder Ausliefern. Nie als etwas Eigenes zum Daran-Denken.

Der Unfall, und es ist die beste Geschichte in diesem Buch: Eine andere KI baute ein komplettes neues Feature im echten Projekt. Sie las den Einstiegspunkt, fand die Regeln und befolgte jede einzelne. Sie schrieb die Dokumentation, aktualisierte das Inhaltsverzeichnis, frischte das Inventar auf, schrieb den Chronik-Eintrag. Wirklich gute Arbeit, ungefragt.

Den Sicherungsbefehl hat sie nie ausgeführt.

Eine Stunde hervorragender Arbeit lag ungesichert auf einem Laptop — kein Backup, nicht online, einen Unfall vom Verschwinden entfernt.

Die Regeln haben funktioniert. Das Sicherheitsnetz war ausgeschaltet.

Das ist der Unterschied. Regeln schützen dich davor, etwas zu vergessen. Erzwingen schützt dich vor dem Tag, an dem jemand die Regeln überspringt — und „jemand" schließt jede KI ein, die du je benutzen wirst.

Der Beweis kam eine Woche später, und er war schlimmer als beim ersten Mal.

Da stand die Regel längst geschrieben. Sie lag in der Arbeitsregeldatei, nummeriert, mit ihrem Grund daneben, dort wo beide KIs sie in jeder Sitzung lesen. Sie lautete: *Eine Sitzung endet mit dem Sicherungsbefehl. Fertig heißt gepusht, nicht gespeichert.*

An einem einzigen Tag haben **beide KIs sie übersprungen.** Codex ließ zwei fertige Commits auf dem Laptop liegen. Ich sechs Dateien, nicht einmal eingecheckt. Die Online-Kopien zeigten den Vortag. Wer sich das Projekt geholt hätte, hätte nichts von der Arbeit dieses Tages bekommen — ein neues Modul, ein funktionierendes E-Mail-System, eine korrigierte Sicherheitsseite.

Die Regel existierte. Sie war klar. Sie wurde gelesen. Sie wurde in acht Stunden zweimal übersprungen.

Repariert hat das kein schärferer Satz. Sondern **zwölf Zeilen Skript**, die nach jedem Speichern von selbst laufen und ungefragt ausgeben:

> *Diese Arbeit existiert nur auf diesem Rechner.*

Es blockiert nichts — eine Prüfung, die blockiert, wird umgangen. Es weigert sich nur, dich nicht wissen zu lassen.

Wenn du eine Regel schreibst und dich dabei ertappst, wie du hoffst, dass sich jemand daran erinnert, bist du nicht fertig. Die Regel ist der Entwurf. Was von selbst läuft, ist die

Umsetzung.

Regel 16 — Traue keinem Messgerät, das du nicht hast scheitern sehen

Die Regel: Bevor du einem grünen Licht glaubst, bring es einmal absichtlich zum Rotwerden. Wenn du es nicht rot bekommst, weißt du nicht, ob Grün etwas bedeutet.

Das ist die letzte Regel, die ich geschrieben habe, und die eine, die ich behalten würde, wenn ich die anderen fünfzehn hergeben müsste.

Die Unfälle — vier davon, an einem einzigen Arbeitstag:

1. **Eine Protokollansicht lieferte eine leere Liste** für Funktionsaufrufe, die nachweislich stattgefunden hatten. Kein Fehler. Keine Warnung. Eine leere Liste, die sich exakt liest wie „nichts ist schiefgegangen". Es wurden Stunden damit verbracht, daraus das Falsche zu schließen, und ein Tagebucheintrag musste am nächsten Tag korrigiert werden.
2. **Ein Suchindex meldete null Treffer** für einen Satz, den ich dreißig Sekunden zuvor in genau diese Datei geschrieben hatte. Hätte ich ihm geglaubt, hätte ich einen Abschnitt für fehlend gehalten und ein zweites Mal geschrieben.
3. **Ein Zeilenzähler zählte Leerzeilen als null.** Zwei Fassungen derselben Datei sahen dadurch siebzig Zeilen auseinander aus. Beinahe hätte ich einen Unterschied gemeldet, den es nicht gab.
4. **Eine Prüfung, die ich am selben Morgen gebaut hatte,** verglich die falschen zwei Dinge und schlug bei jeder Änderung Alarm, auch bei solchen, die sie gar nichts angingen. Ich hatte sie **genau dafür** geschrieben, veraltete Kopien zu finden — und binnen einer Stunde rief sie „Wolf".

Nummer vier ist die, bei der es sich zu verweilen lohnt. Ich hatte den ganzen Tag genau diese Lektion gelernt, ein Werkzeug dagegen gebaut — und denselben Defekt in das Werkzeug eingebaut.

Warum das die gefährlichste Fehlerart überhaupt ist: Ein kaputtes Messgerät, das einen Fehler anzeigt, wird am selben Vormittag repariert. Ein Messgerät, das leise „alles in Ordnung" meldet, tut zwei Dinge gleichzeitig — es sagt dir nichts, und es nimmt dir den Verdacht. Du hörst auf hinzusehen. Das ist schlechter als gar kein Messgerät, denn ohne würdest du wenigstens noch von Hand prüfen.

Was konkret zu tun ist:

- **Mach es einmal absichtlich kaputt.** Lösch eine Zeile, die der Prüfer finden muss. Benenn etwas um, das ihm auffallen soll. Sieh zu, wie er rot wird. Jetzt bedeutet Grün etwas.

- **Frag, was das Messgerät nicht sehen kann,** und schreib die Antwort daneben. Meines gibt bei jedem Lauf aus: *Das prüft nur Struktur; die Fakten gegen das laufende System verifizieren.* Ein Messgerät, das seine eigenen Grenzen nennt, ist das einzige, dem zu trauen sich lohnt.
- **Misstraue der Leere.** Ein leeres Ergebnis ist kein Beweis für Abwesenheit. Es passt genauso gut zu „die Frage ist nie angekommen".
- **Wenn etwas wichtig ist, miss zweimal auf verschiedene Weise.** Die zurückgefallene Kopie hat nicht der Prüfer gefunden, sondern ein Vergleich zweier Zeilenzahlen von Hand — und selbst der war beim ersten Versuch falsch.

Teil 6 — Was passiert, wenn dein Projekt wächst

Berechtigte Frage an dieser Stelle: Übersteht das den Kontakt mit einem wachsenden Projekt? Oder ist es eine hübsche Ordnung, die zerfällt, sobald echte Arbeit passiert?

Es wurde zufällig getestet, und der Test lohnt sich anzuschauen.

Die Ausgangslage

Eine KI (Claude) hatte das Doku-System am Vormittag gebaut. Am Nachmittag sollte eine **völlig andere KI** — Codex, von einer anderen Firma — ein neues Feature bauen. Ein großes: neun neue Datenbanktabellen, eine neue Schnittstelle, drei neue Datenbankfunktionen.

Niemand erwähnte Dokumentation. Niemand sagte „und vergiss die Regeln nicht". Die beiden KIs haben nie miteinander gesprochen.

Was passierte

Codex öffnete das Projekt, fand die Einstiegsdatei und las sie. Die Regeln standen darin, jede mit ihrem Grund.

Dann baute es das Feature — **und schrieb die Dokumentation als Teil der Arbeit.** Es aktualisierte die Seite des Bereichs, schrieb eine Warnung um, die nicht mehr stimmte, ergänzte eine Zeile im Inhaltsverzeichnis, frischte das Inventar auf und schrieb einen Chronik-Eintrag, der erklärte, warum das Feature so gebaut wurde.

Der Prüfer ging von 74 auf 87 dokumentierte Objekte. Alles grün.

Was es übersah

Vier Dinge, alle klein, alle an Stellen, die der Prüfer nicht sehen kann:

1. **Einen Querverweis.** Die Seite verlinkte den neuen Chronik-Eintrag nicht — ein Leser fand also das *Was*, aber nicht das *Warum*.
2. **Offene Aufgaben am falschen Ort.** Vier noch zu erledigende Dinge standen in der Chronik statt auf der Aufgabenliste. Wer nach „was ist noch offen" sucht, hätte sie nicht gefunden.
3. **Eine Statuszeile** auf der Übersichtsseite sagte noch das Alte.
4. **Zahlen im Fließtext.** An drei Stellen stand noch 27 Tabellen. Es waren 36.

Zwanzig Minuten zum Finden und Beheben. Punkt vier wird jetzt automatisch gefunden — diese Prüfung existiert *wegen* dieses Vorfalls.

Warum das ein gutes Ergebnis ist

Schau dir die Alternative an.

Ohne all das wäre dieses Feature gebaut und nirgends dokumentiert worden. Neun Tabellen, eine Schnittstelle, drei Funktionen — entdeckt drei Monate später von jemandem, der es aus dem Code zurückentwickeln muss und sich fragt, ob es überhaupt noch benutzt wird.

Stattdessen: Eine andere KI, von einer anderen Firma, ohne jede Einweisung, erzeugte korrekte Dokumentation als Nebenprodukt ihrer Arbeit.

Das ist das eigentliche Versprechen. Nicht Perfektion. Kontinuität — die Nächste macht dort weiter, wo die Letzte aufgehört hat, und du erklärst nichts.

Und ein fünftes Versäumnis, das schwerer wiegt

Codex hat den Sicherungsbefehl nie ausgeführt. Siehe Regel 15 in Teil 5. Die Arbeit lag ungesichert.

Ich schreibe das dazu, weil ein Buch, das nur die gute Hälfte erzählt, dir etwas verkauft. Das System hat gehalten. Das Netz war nicht eingeschaltet. Beides ist wahr, und das Zweite ist in etwa einer Minute behebbar.

Eine Woche später: beide KIs gleichzeitig

Die Geschichte oben handelt von zwei KIs, die sich abwechseln. Was dann passierte, waren zwei KIs am **selben Projekt am selben Tag** — und das zerbrach Dinge, die das Abwechseln nie berührt hat.

Was jede tat. Codex baute ein neues Modul, eine eigene Unterseite mit eigener Anmeldeübergabe. Ich verbrachte dieselben Stunden mit dem E-Mail-System: Zustellberichte, bestätigter Opt-in, ein rechtssicherer Fuß. Keiner von uns wusste vom anderen.

Was schiefging, der Reihe nach.

Nach zwei Stunden suchte ich aus einem ganz anderen Grund nach einem Wort und bekam einen Dateinamen zurück, den ich noch nie gesehen hatte: ein Tagebucheintrag von **heute**, über ein Modul, von dem ich nichts wusste. So habe ich es erfahren. Nicht durch ein Werkzeug. Durch einen Zufall.

Dann wurde es schlimmer. Dieser Eintrag existierte **nur im laufenden Projekt** — nicht im gesicherten Repository. Und eine Seite im Repository behauptete etwas über Sicherheit, das die Projektkopie **richtig** und das Repository **falsch** sagte. Die Kopie war dem Original voraus, ausgerechnet an der Stelle, wo Falschsein teuer ist.

Gleichzeitig hatten wir beide Arbeit auf dem Laptop liegen: zwei fertige Commits von Codex, sechs nicht eingecheckte Dateien von mir. Alles Online zeigte den Vortag.

Und der Prüfer sagte: **Structure OK.**

Was es gekostet hat: etwa drei Stunden eines Nachmittags, vollständig damit verbracht herauszufinden, was stimmt, statt irgendetwas zu bauen.

Was es tatsächlich behoben hat, und es ist peinlich klein.

Ein Ordner mit **einer Datei je KI**. Meine sagt, woran ich arbeite und wo. Codex hat seine eigene. Zwei Zeilen pro Stück.

Der Kniff ist *eine Datei je KI*, nicht eine gemeinsame. Zwei KIs, die dieselbe Datei bearbeiten, kollidieren; zwei KIs, die je ihre eigene bearbeiten, können es nicht. Die Konstruktion erledigt die Arbeit, nicht die Disziplin.

Zweites Stück: ein Startskript, das zehn Sekunden braucht und ausgibt, was man sonst nicht sehen kann — Commits, die nur hier liegen, nicht eingecheckte Dateien, **wer sonst gerade woran arbeitet**, die neuesten Tagebucheinträge, ob die Kopie zurückgefallen ist.

Hätte es das um neun Uhr morgens gegeben, hätte ich in der ersten Minute gewusst, was ich um elf durch Zufall herausfand.

Ein zu alter Eintrag ist schlimmer als keiner. Er beansprucht ein Gebiet, das vielleicht seit Stunden verlassen ist. Deshalb wird alles, was älter als ein halber Tag ist, als *möglicherweise vergessen* markiert statt geglaubt.

Was es ausdrücklich nicht tut: Es verhindert nichts. Zwei KIs können weiterhin im selben Moment dieselbe Datei ändern. Es macht die Kollision *sichtbar, bevor sie passiert* — mehr kann eine zweizeilige Textdatei ehrlicherweise nicht versprechen.

Der unangenehme Teil. Ich habe eine Reihe neuer Werkzeuge vorgeschlagen, um das zu beheben. Dann habe ich richtig hingesehen und festgestellt: Den Sicherungsbefehl gab es längst, und die Regel, die ihn verlangt, stand längst geschrieben. Es fehlte nichts. Es war schlicht übersprungen worden — von uns beiden, am selben Tag.

Gebaut habe ich deshalb fast nichts von dem, was ich vorgeschlagen hatte. Gebaut habe ich den Teil, der das Überspringen **sichtbar** macht.

Wenn du eine Sache aus diesem Kapitel mitnimmst: Wenn etwas wiederholt schiefgeht, prüfe erst, ob die Lösung schon existiert, bevor du eine zweite baust. Eine Regel, die übersprungen

wird, braucht keine bessere Fassung ihrer selbst. Sie braucht etwas, das es bemerkt.

Teil 7 — Von Hand aufsetzen

Diesen Teil kannst du komplett überspringen.

Im Anhang steht ein Prompt. Füg ihn in deine KI ein, beantworte drei Fragen, und alles aus diesem Buch wird für dich gebaut — samt Prüfer, getestet.

Dieser Teil existiert aus zwei Gründen. Manche wollen es selbst machen. Und wichtiger: **damit nichts im System eine Blackbox bleibt.** Selbst wenn du nie etwas davon tippst, ergibt der Ordner, den deine KI anlegt, nach dem Lesen Sinn.

Was angelegt wird

```
dein-projekt/
├── CLAUDE.md              Einstieg für Claude
├── AGENTS.md              Einstieg für andere KIs + die Regeln
├── ROUTING.md             Frage -> Datei  (das winzige Verzeichnis)
├── START-HIER.txt         dasselbe für Menschen
├── .gitignore             was nicht mitgesichert wird
├── referenzen/
│   ├── _VORLAGE.md        die Pflichtform einer Seite
│   └── ...                eine Datei pro Bereich — wächst mit
├── chronik/
│   └── README.md          „das ist Geschichte, nie zum Lernen laden"
├── system-inventar.json   was wirklich existiert, maschinenlesbar
├── NOTFALL.md             wo alles liegt, wem was gehört
└── ops/
    ├── pruefen.*          der Prüfer
    └── sichern.*          prüfen, sichern, überallhin kopieren
```

Am ersten Tag ist `referenzen/` leer und `ROUTING.md` hat keine Zeilen. **Das ist richtig so.** Das Gerüst ist gültig, wenn es leer ist. Bereiche entstehen, während du baust.

Die sechs Schritte

1. Bestand aufnehmen — und nicht raten. Welche Bereiche hat dein Projekt *heute*? Keine Pläne. Fünf bis zwanzig ist normal.

> Der teuerste Fehler beim Bau dieses Systems war, aus einer Zusammenfassung zu schließen, statt nachzusehen. Zweimal. Beide Male lag ich falsch, beide Male musste ich korrigiert werden. Sieh dir das echte Ding an.

2. Eine Seite pro Bereich, mit den sechs Abschnitten aus Regel 4 in Teil 5.

3. Das Inhaltsverzeichnis, in der Form aus Regel 2 in Teil 5. Unter fünfzig Zeilen. Wenn in deinem Projekt ein Wort zwei Dinge bedeutet, warne ganz oben davor — im echten Projekt bedeutete „Avatar" sowohl eine Käufer-Persona als auch einen sprechenden Presenter im Video, und diese eine Warnung hat Stunden gespart.

4. Versionskontrolle und Kopien.

```
git init
git add -A
git commit -m "Erste Fassung der Dokumentation"
```

Legt eine Historie an und sichert den aktuellen Stand. Ab hier siehst du, was sich geändert hat, und kannst zurück.

Dann ein **privates** Repository online:

```
git remote add origin https://github.com/DEINNAME/DEINPROJEKT.git
git push -u origin master
```

Sagt deinem Ordner, wo seine Online-Kopie liegt, und lädt sie hoch.

Privat, nicht öffentlich — diese Seiten beschreiben, wie dein System gebaut ist und was daran noch nicht sicher ist.

Schreib niemals ein Passwort oder einen Schlüssel in diese Dateien. Den Namen einer Einstellung, nie den Wert. Diese Dokumentation landet auf anderen Rechnern.

5. Der Prüfer, nach den Regeln 10 und 11 in Teil 5.

6. Mach ihn absichtlich kaputt. Trag eine falsche Zahl ein, lass ihn laufen, sieh nach, ob er meckert, mach es rückgängig.

> Eine ungeprüfte Prüfung ist schlimmer als keine, weil du ihr genau dann vertraust, wenn du es nicht solltest. Alles andere auf dieser Liste ist verhandelbar. Dieser Schritt nicht.

Unüberspringbar machen

Häng den Prüfer ans Sichern oder Ausliefern — siehe Regel 15 in Teil 5. Nicht als eigenen Befehl, den jemand vergessen kann, deine KI eingeschlossen.

Teil 8 — Die KI anschließen

Stand 2026-07-26. Dieses Kapitel veraltet am schnellsten, weil sich diese Konventionen ändern. Wenn hier etwas nicht zu dem passt, was du siehst, gilt die aktuelle Dokumentation des Werkzeugs.

Wie es funktioniert

Die meisten KI-Werkzeuge suchen beim Öffnen eines Projekts nach einer Anweisungsdatei. Leg deinen Einstiegspunkt dorthin, und du erklärst nie wieder etwas.

Zwei Dateinamen decken heute die meisten Werkzeuge ab:

- `CLAUDE.md` — wird von Claude Code gelesen
- `AGENTS.md` — eine herstellerneutrale Konvention, gelesen von Codex und anderen

Schreib beide. Sie sind kurz und dürfen fast dasselbe sagen. Zwei Dateien kosten nichts. Keine zu haben macht deine ganze Arbeit unsichtbar.

Was hineingehört

Kurz. Das wird jede Sitzung gelesen, jede Zeile ist also eine dauerhafte Kostenstelle — genau der Fehler aus Regel 2.

```
# Hier anfangen

Lies zuerst ROUTING.md. Dort steht, welche EINE Datei die Frage
beantwortet. Lies danach nur diese Datei.

Drei Regeln, bevor du irgendetwas lädst:

1. Der Chronik-Ordner ist Geschichte, nicht der Ist-Zustand. Niemals
   laden, um zu lernen, wie etwas funktioniert — er ist groß und
   voller Dinge, die nicht mehr stimmen.
2. Willst du etwas ÄNDERN, lies zusätzlich AGENTS.md.
3. <das eine verwirrende Wort deines Projekts und was es wo bedeutet>

Nach einer Änderung: die Seite dieses Bereichs in derselben Aufgabe
aktualisieren, Datum auf heute, dann den Prüfer ausführen.
```

Widersteh dem Drang, hier dein Projekt zu erklären. Genau dieser Reflex hat die 10.000-Token-Datei erzeugt.

Nachprüfen, ob es wirklich greift

Nimm es nicht an — teste es.

Starte eine völlig frische Sitzung und frag etwas, von dem du weißt, dass es dokumentiert ist. Kommt die Antwort, ohne dass du etwas erklärst, funktioniert es. Fragt die KI „was ist das für ein Projekt?", hat sie deine Datei nie gefunden.

Für ein Werkzeug, das nichts automatisch liest, fügst du das voran:

```
Projekt: <Name>, liegt unter <Pfad>.

Lies zuerst ROUTING.md — eine kurze Datei, die sagt, welche EINE Datei
meine Frage beantwortet. Lies danach nur diese.

Lade den Chronik-Ordner nicht, um zu lernen, wie etwas funktioniert.

Wenn du etwas änderst: die Seite dieses Bereichs in derselben Aufgabe
aktualisieren und vor dem Abschluss den Prüfer ausführen.

Meine Aufgabe heute: <Aufgabe>
```

Was du erwarten kannst

Keine Perfektion. Siehe Teil 6: Eine andere KI befolgte ungefragt jede Regel, übersah vier Kleinigkeiten und ließ den Sicherungsschritt aus. Eine Woche später ließen zwei gleichzeitig arbeitende KIs die Arbeit eines ganzen Tages auf einem Laptop liegen und legten einen Tagebucheintrag dorthin, wo die Sicherung ihn nicht sehen konnte.

Das ist das realistische Ergebnis, und es ist der Alternative weit überlegen. Der Maßstab ist nicht, ob etwas schiefgeht. Sondern wie lange du brauchst, um es zu merken.

Das ist das realistische Ergebnis — und es ist weit besser als die Alternative.

Was du davon hast

Billiger. Ein kleines Verzeichnis plus eine Seite statt allem, was du besitzt. Etwa fünfmal weniger, gemessen.

Nicht gebunden. Claude, Codex, was auch immer als Nächstes kommt — sie alle lesen einfachen Text. Kommt etwas Besseres, wechselst du und verlierst nichts. Dein Wissen liegt nicht im Produkt einer Firma.

Kein Erklären mehr. Neue Sitzung, neues Werkzeug, neuer Tag. Sie liest den Einstiegspunkt und weiß, wo sie ist.

Etwas, das du übergeben könntest. Verkaufst du das Projekt, holst du einen Partner dazu oder kommst nach einem Jahr zurück — es erklärt sich selbst. Der letzte Fall trifft dich mit Sicherheit.

Die acht Dinge zum Behalten

1. **Neun von zehn Tokens sind meist Wegfindung, nicht Antwort.** Miss nach, bevor du etwas annimmst.
2. **Eine Seite pro Bereich, die heute beschreibt.** Zwei Seiten über eine Sache machen beide unglaubwürdig.
3. **Die Chronik ist nicht die Bedienungsanleitung.** Lerne nie aus einem Änderungsprotokoll, wie etwas funktioniert.
4. **Schreib auf, was dich Zeit gekostet hat.** Alles andere lässt sich wiederherleiten. Das nicht.
5. **Prüfe in jede Richtung — auch rückwärts.** Dokumentation verrottet, indem sie Verschwundenes beschreibt. Und sie verrottet, wenn eine *Kopie* dem Original voraus ist und nichts, was dir gehört, es sehen kann.
6. **Traue keinem Messgerät, das du nicht hast scheitern sehen.** Bring es absichtlich zum Rotwerden. Und behandle ein leeres Ergebnis als Frage, nie als Antwort.
7. **Regeln sind kein Erzwingen.** Eine KI befolgte jede Regel und ließ die Sicherung trotzdem aus. Aufgeschrieben wurde dieselbe Regel dann von zwei KIs an einem Tag übersprungen. Behoben haben es zwölf Zeilen, die von selbst laufen.
8. **Wenn ihr zu zweit arbeitet, schreibt es hin.** Eine Datei je Bearbeiter, zwei Zeilen. Eine Kollision, die du kommen siehst, ist eine, der du ausweichen kannst.

Anhang — Fertige Prompts

Kopier sie. Ersetze alles in `<spitzen Klammern>`.

A. Ein ganz neues Projekt — das Gerüst anlegen

Für den allerersten Tag, wenn es noch **nichts** gibt. Er legt keine Inhalte an, sondern das leere Gerüst und die Regel, die es danach von selbst wachsen lässt.

```
Ich starte ein neues Projekt. Es existiert noch nichts — kein Code, keine
Datenbank, nichts. Bevor wir bauen, richte bitte das Gerüst ein, mit dem
jede KI später sofort weiß, worum es geht, ohne dass ich etwas erkläre.

Frag mich ZUERST diese drei Dinge und warte auf meine Antwort:
1. Wie heißt das Projekt, und was soll es in einem Satz tun?
2. Womit wird gebaut — Sprache, Datenbank, Hosting? Wenn etwas noch
   offen ist, sage ich "offen". Dann lässt du diese Stelle als
   AUSZUFÜLLEN stehen, statt zu raten.
3. Auf welchem Betriebssystem arbeite ich? Davon hängt ab, ob die
   Skripte .sh oder .ps1 werden.

Lege danach genau diese Struktur an — leer, aber vollständig:

  ROUTING.md           Frage -> Datei. Anfangs eine LEERE Tabelle plus
                       die drei Laderegeln. Unter 50 Zeilen halten.
  AGENTS.md            die Arbeitsregeln (unten). Wird nur beim ÄNDERN
                       gelesen, nicht beim Fragen.
  CLAUDE.md            kurzer Einstieg, zeigt auf ROUTING.md.
  START-HIER.txt       dasselbe für Menschen, mit Textbausteinen zum
                       Kopieren.
  referenzen/_VORLAGE.md   die Pflichtform einer Bereichsdatei.
  chronik/README.md    erklärt: das hier ist Geschichte und wird NIE
                       geladen, um zu lernen, wie etwas funktioniert.
  system-inventar.json leere Listen plus einen _refresh-Block mit den
                       konkreten Befehlen, mit denen man in DIESEM Stack
                       fragt: "was existiert gerade?"
  NOTFALL.md           wo alles liegt, wem was gehört, wie man nach
                       einem Rechnerverlust zurückkommt. Erst
                       stichwortartig — wächst mit.
  ops/pruefen.*        der Prüfer (unten).
  ops/sichern.*        prüft, committet, schiebt in alle Kopien.
  .gitignore           Kommentare NUR in eigenen Zeilen. Ein Kommentar
                       am Zeilenende wird Teil der Regel und macht sie
                       wirkungslos.

In AGENTS.md gehören diese Regeln, jede MIT Begründung — eine Regel ohne
Begründung wird von der nächsten KI ignoriert:

  1. Pro Bereich genau EINE Referenz, die den Ist-Zustand beschreibt.
```

```
     Warum: zwei Dateien über dieselbe Sache widersprechen sich binnen
     eines Monats, und dann glaubt man keiner mehr.
  2. Entsteht ein neuer Bereich, entsteht im SELBEN Arbeitsgang seine
     Referenz und seine Zeile in ROUTING.md. Warum: "später" heißt nie.
  3. Wird etwas geändert, wird die Referenz im selben Arbeitsgang
     aktualisiert und ihr Stand-Datum auf heute gesetzt.
  4. Die Chronik ist Geschichte, nicht der Ist-Zustand. Warum: sie
     wächst unbegrenzt und enthält Überholtes. Wer daraus lernt, lernt
     Falsches und zahlt dafür.
  5. Niemals Geheimniswerte in eine Datei, nur die NAMEN von
     Einstellungen. Warum: die Doku verlässt das Gerät.
  6. Was du nicht überprüft hast, kennzeichne als ungeprüft. Rate nicht
     und stell es als Tatsache dar.
  7. Vor "fertig": ops/pruefen ausführen, dann ops/sichern.

Der Prüfer muss fehlschlagen, wenn:
  - einer Referenz das Stand-Datum fehlt oder es älter als 60 Tage ist
  - ein vorgeschriebener Abschnitt fehlt
  - eine Referenz von ROUTING.md aus nicht erreichbar ist
  - ein interner Verweis ins Leere zeigt
  - etwas im system-inventar.json auf keiner Seite vorkommt
  - eine Seite etwas beschreibt, das im Inventar nicht mehr steht
  - eine Zahl im Fließtext nicht mehr zum Inventar passt

Bei NULL Bereichen muss er grün durchlaufen. Das Gerüst ist gültig,
auch wenn noch nichts drinsteht.

Zum Schluss, und das ist der wichtigste Teil:

  a) Führ den Prüfer aus und zeig mir, dass er grün ist.
  b) Mach absichtlich etwas kaputt — leg eine Referenz ohne Datum an —,
     lass ihn laufen, zeig mir, dass er meckert, und räum es wieder weg.
     Sag mir nicht, dass er funktioniert, ohne es vorzuführen.
  c) git init und erster Commit. Sag mir danach die zwei Befehle, mit
     denen ich eine Kopie außerhalb dieses Rechners anlege.

Und ab jetzt gilt für alles, was wir zusammen bauen: jeder neue Bereich
bekommt im selben Arbeitsgang seine Referenz und seine Routing-Zeile.
Ich möchte dich daran nicht erinnern müssen.
```

Der letzte Absatz ist der eigentliche Trick. Alles davor ist einmalige Einrichtung. Dieser Satz ist das, was das System danach ohne dich wachsen lässt — und er steht in `AGENTS.md`, also liest ihn auch jede KI, die morgen kommt.

Am Anfang hat `ROUTING.md` null Zeilen und der Prüfer null zu prüfen. Das ist richtig so. Der erste Bereich entsteht, sobald du das erste Stück gebaut hast — und ab da trägt die Regel.

B. Für ein Projekt, das schon existiert

```
Dieses Projekt existiert und läuft bereits. Ich will eine
Dokumentation, die die WIRKLICHKEIT abbildet, nicht das, was
irgendwann einmal beabsichtigt war.

Fang mit Messen an, nicht mit Schreiben:

1. Sieh dir das laufende System direkt an. Liste alles auf, was du
   findest, und sag dazu, wie du es herausgefunden hast. Was du nicht
   prüfen konntest, sag klar.

2. Vergleiche das mit vorhandener Dokumentation. Melde drei Listen
   getrennt:
   - Dinge, die existieren, aber nirgends dokumentiert sind
   - Dinge, die dokumentiert sind, aber nicht mehr existieren
   - Dinge, die anders dokumentiert sind, als sie tatsächlich sind

3. Zeig mir diese Listen, BEVOR du etwas schreibst. Manche Lücken
   sind Absicht, und das musst du von mir erfahren.

Danach: leg dasselbe Gerüst an wie in Prompt A (ROUTING.md, AGENTS.md,
CLAUDE.md, referenzen/, chronik/, system-inventar.json, ops/pruefen,
ops/sichern) — nur füllst du es diesmal sofort mit dem, was du in
Schritt 1 gefunden hast, statt es leer zu lassen. Dieselben Regeln,
derselbe Prüfer, derselbe Beweis am Ende: absichtlich kaputt machen und
vorführen, dass er meckert.

Zwei Dinge will ich besonders wissen:
- Alles, was tot aussieht, aber vielleicht nicht tot ist. Prüfe, ob
  etwas es tatsächlich benutzt, bevor du es für tot erklärst.
- Alles, was zu einem anderen Projekt oder Besitzer gehört, als sein
  Name vermuten lässt.
```

Die letzten zwei Punkte stammen aus echten Funden. In meinem eigenen Projekt gehörte ein Dienst, der nach dem Projekt benannt war, in Wahrheit zu einem **anderen Geschäft** — ihn abzuschalten hätte etwas Laufendes zerstört. Namen lügen. Prüf die Nutzung.

C. Gesund halten — das sagst du beim Arbeitsbeginn

```
Lies INHALTSVERZEICHNIS.md, danach nur die Datei, die es für meine
Aufgabe nennt. Lade den Chronik-Ordner nicht.

Wenn du etwas änderst: die Seite dieses Bereichs in DERSELBEN Aufgabe
aktualisieren, Datum auf heute setzen, einen Chronik-Eintrag schreiben
und den Prüfer ausführen, bevor du mir sagst, dass du fertig bist.

Meine Aufgabe: <Aufgabe>
```

D. Die monatliche Durchsicht

```
Führ den Prüfer aus und berichte, was er sagt.

Prüf danach drei Dinge, die er nicht sehen kann:
1. Zahlen im Fließtext, die nicht mehr stimmen (Anzahl Tabellen,
   Endpunkte, Preise, Grenzen).
2. Bereiche, in denen sich das laufende System geändert hat, die Seite
   aber nicht — vergleich die Datumsangaben mit den jüngsten
   Chronik-Einträgen.
3. Seiten, deren "Hängt zusammen mit" auf etwas zeigt, das inzwischen
   umbenannt oder zusammengelegt wurde.

Berichte die Funde. Repariere nichts, bevor ich die Liste gesehen habe.
```

Eine letzte Sache

Das Schwierigste daran ist nicht das Aufsetzen. Es ist der erste ehrliche Blick.

Als ich das für ein echtes Projekt gemacht habe, brachte die Bestandsaufnahme Dinge zutage, die niemand finden wollte: einen Dienst, der seit Monaten lief und von dem niemand sicher wusste, was er tut. Zwei getrennte Geschäfte auf einer Maschine, sodass keines ohne das andere verkauft werden konnte. Einen Einstiegspunkt, der mit einem kaputten Laptop verschwunden wäre. Und 470 Megabyte Müll, die wegen eines falsch gesetzten Kommentars für immer mitgesichert worden wären.

Nichts davon stand in irgendeiner Dokumentation, weil nie jemand nachgesehen hatte.

Geh nachsehen. Schreib auf, was du findest. Und lass die KI es so halten.

Geschrieben mit dem System, das es beschreibt. Jede Zahl wurde an einem echten Projekt gemessen. Jeder Fehler ist wirklich passiert.