

HANDBUCH

# Software mit KI entwickeln

Das kanban-kit und das claude-workflow-kit in der Praxis

*Idee, Grundsätze und der tägliche Arbeitsfluss. Der Mensch bleibt am Steuer.*

**Manfred Wolff**

Version 1.5 • August 2026

KI-gestützte Entwicklung

kanban-kit

claude-workflow-kit

Agile

[mwolff.org](https://mwolff.org)

## INHALT

# Inhaltsverzeichnis

|                                                                   |    |
|-------------------------------------------------------------------|----|
| <b>Einleitung</b> .....                                           | 4  |
| Warum dieses Buch .....                                           | 4  |
| Wie dieses Buch zu lesen ist .....                                | 9  |
| Die zwei Kits auf einen Blick .....                               | 11 |
| <b>Teil I • Die Idee und die Grundsätze</b> .....                 | 14 |
| 1. Es geht nur agil .....                                         | 14 |
| 2. Was ein Sprachmodell tut .....                                 | 19 |
| 3. Der Mensch bleibt am Steuer .....                              | 24 |
| 4. Rollen im KI-augmentierten Team .....                          | 29 |
| <b>Teil II • Die zwei Werkzeuge verstehen</b> .....               | 36 |
| 5. Das kanban-kit: das Board als geteilter Statusträger .....     | 36 |
| 6. Das claude-workflow-kit: der Prozess, ausführbar gemacht ..... | 40 |
| <b>Teil III • Der Arbeitsfluss in der Praxis (fachlich)</b> ..... | 47 |
| 7. Das Issue als Single Source of Truth .....                     | 47 |
| 8. Der Neun-Schritt-Prozess im Überblick .....                    | 52 |
| 9. Drei Modelle für ein Issue .....                               | 59 |
| 10. Ein Tag im KI-augmentierten Team .....                        | 68 |
| 11. Die PO-Schleife: fachliche und technische Issues .....        | 77 |
| 12. Takt und Events .....                                         | 82 |
| 13. Iterationsmodelle .....                                       | 86 |
| <b>Teil IV • Einrichtung und Betrieb (technisch)</b> .....        | 91 |
| 14. kanban-kit betreiben .....                                    | 91 |

|                                                           |            |
|-----------------------------------------------------------|------------|
| 15. claude-workflow-kit installieren .....                | 95         |
| 16. Die Skills im Detail .....                            | 100        |
| 17. Persistente Kontext-Dateien und der Vault .....       | 105        |
| 18. GitHub, GitLab oder lokal .....                       | 109        |
| 19. Nachts arbeiten lassen: der Nacht-Runner .....        | 113        |
| 20. Qualität und Security-Gates im Build verdrahten ..... | 125        |
| <b>Teil V • Den Prozess am Leben halten .....</b>         | <b>134</b> |
| 21. Erste Schritte für neue Teams .....                   | 134        |
| 22. Anti-Patterns und wie man sie erkennt .....           | 137        |
| 23. Die KI-Retrospektive .....                            | 141        |
| 24. Ausblick .....                                        | 144        |
| <b>Anhang .....</b>                                       | <b>152</b> |
| A. Skill-Referenz .....                                   | 152        |
| B. Config-Referenz .....                                  | 156        |
| C. Glossar .....                                          | 164        |
| D. Quellen und weiterführende Links .....                 | 169        |
| E. Über die Entstehung dieses Buches .....                | 173        |
| F. Dank .....                                             | 175        |

## EINLEITUNG

# Warum dieses Buch

2024 habe ich bei der fuseki GmbH angefangen, einer Firma, die vollständig auf KI spezialisiert ist. Keine Softwarebude, die auch etwas mit KI macht. Ein Haus, das eigene Modelle für industrielle Anwendungen baut. Mein Auftrag bestand aus einer einzigen Frage: Wie setze ich KI in der Softwareentwicklung ein?

Die ersten Schritte waren ernüchternd, vor allem bei der Qualität. Was zurückkam, sah aus wie Code, und zu oft war es keiner. Plausible Struktur, plausible Benennung, und dann eine Methodensignatur, die es nicht gab, oder eine Logik, die im zweiten Randfall auseinanderfiel. Ich habe damals mehr Zeit mit Korrigieren verbracht als mit Entwickeln.

Aus heutiger Sicht weiß ich, warum. 2024 haben die Modelle gerade das Laufen gelernt. Ich habe mit den ersten Gehversuchen einer Technik gearbeitet und daraus geschlossen, dass die Technik nichts taugt. Diesen Fehlschluss höre ich bis heute regelmäßig, nur dass die Leute inzwischen mit Werkzeugen von 2024 argumentieren und über 2026 urteilen.

Ich spiele Schach, deshalb liegt mir ein Beispiel nahe. Wer in den Achtzigern gegen ein Schachprogramm gewonnen hat und daraus geschlossen hat, Computer könnten kein Schach, lag damals richtig und ist seit dreißig Jahren widerlegt. Heute schlägt eine Engine auf einem Smartphone jeden Menschen, der je gelebt hat. Das Bild sagt etwas über das Tempo, in dem sich eine Technik verändert, und keinen Schritt weiter. Beim Schach sitzt mir ein Gegner gegenüber, und am Ende verliert einer. In der Softwareentwicklung verliert niemand dadurch, dass ich gewinne, und das Modell ist nicht mein Gegner.

Der Wendepunkt war kein Werkzeug. Er kam beim Lesen.

Ich habe damals sehr viele Artikel zum Thema gelesen, und viele davon handelten von Scheiterquoten. Ein großer Teil der KI-Vorhaben komme nie in den Produktivbetrieb, und manchmal werde die Arbeit sogar langsamer statt schneller. RAND kam 2024 auf mehr als achtzig Prozent ge-

scheiterter KI-Vorhaben, doppelt so viele wie bei IT-Projekten ohne KI. Gartner nennt für abgebrochene Generative-AI-Projekte dreißig Prozent. Die Größenordnung war damals überall zu lesen und ist es bis heute. Je nachdem, was als Scheitern gezählt wird, liegt der Wert irgendwo dazwischen, und für mein Argument reicht das.

Beim Lesen ist mir aufgefallen, dass ich auf die falsche Hälfte schaue.

Wenn achtzig Prozent scheitern, dann gelingen zwanzig. Warum sollte ich mich mit den achtzig beschäftigen? Seitdem sehe ich mir die zwanzig an und gucke, was sie anders machen.

Wie berechtigt die Skepsis war, hat später eine saubere Studie gezeigt. METR hat 2025 gemessen, was passiert, wenn der Zufall entscheidet, welche Aufgaben mit und welche ohne KI bearbeitet werden: Erfahrene Entwicklerinnen und Entwickler brauchten mit KI-Werkzeugen neunzehn Prozent **länger** für ihre Aufgaben. Interessanter als diese Zahl ist die zweite: Dieselben Personen schätzten hinterher, zwanzig Prozent schneller gewesen zu sein. Sie waren langsamer und hielten sich für schneller.

Das Problem liegt nicht bei den Modellen. Ein flüssiger Arbeitstag fühlt sich an wie ein produktiver. Das ist aber nur eine Empfindung, kein Messwert. Genau deshalb steht in diesem Buch so viel über Tests, Gates und Stop-Punkte: Sie sind die Stellen, an denen aus dem Gefühl eine Aussage wird.

Ich habe aufgehört zu fragen, ob KI in der Entwicklung funktioniert, und angefangen zu fragen, unter welchen Bedingungen sie es tut. Daran hängt alles, was in diesem Buch steht. Die Antwort war unbequem und unspektakulär: Es sind die Bedingungen, die die Softwareentwicklung in vierzig Jahren für sich erarbeitet hat. Test-First. Architekturregeln, die scheitern können. Kleine Einheiten. Ein zweiter Blick vor dem Merge. Und ein Mensch, der entscheidet.

Danach kamen zwei Jahre Recherche und Praxis. Die Grundsatzdebatten haben mir in dieser Zeit nicht geholfen. Geholfen haben die Fälle, in denen jemand etwas erfolgreich umgesetzt hatte.

Einer davon war Anfang Mai 2026. Eduard Andrae hat mir gezeigt, wie er trusted blogs gebaut hat, eine Plattform, die Unternehmen und Blogs für Kooperationen zusammenbringt. Das ist kein Webauftritt mit ein paar Unterseiten. Es ist ein zweiseitiger Marktplatz: zwei Nutzergruppen mit eigenen Rollen und Rechten, ein durchsuchbarer Katalog mit Filtern, Buchungs- und Freigabestrecken, Honorarlogik, Zahlung erst nach Veröffentlichung, dazu ein redaktioneller Bereich.

Die Plattform gibt es seit 2015, aber das, was heute läuft, ist kein weiterentwickelter Altbestand. Er hat sie 2025 und 2026 vollständig neu gebaut, und zwar ausschließlich mit KI.

Die Technik war dabei nicht mein Thema. Ich wollte eine Frage beantworten, die ich mir zwei Jahre lang immer wieder gestellt hatte: Geht das überhaupt, jenseits von Demos und Tutorials, in einer Anwendung mit echter fachlicher Komplexität?

Es geht. Das klingt banal. Für mich wurde damit aus einer Vermutung eine Gewissheit. Ich hatte bis dahin viel gelesen und vieles selbst probiert, aber ein fertiges, komplexes Ergebnis von jemand anderem zu sehen, wiegt schwerer als jede Studie. Studien sagen, wie oft etwas schiefgeht. Ein laufendes System sagt, dass es gehen kann.

Im Mai 2026 habe ich dann angefangen, alles, was ich bis dahin wusste, in ein System zu packen, statt es in jedem Projekt neu zusammenzusuchen. Daraus ist das *claude-workflow-kit* entstanden, und später das Board dazu.

## **Der Einwand, und was aus ihm wurde**

Ob die fuseki GmbH KI einführen sollte, war umstritten. Mein damaliger Chef Heiko Dietz hielt KI-gestützte Entwicklung für nicht praktikabel, und wir sind darin nicht zusammengekommen.

Während ich an diesem Buch schrieb, habe ich ihn gefragt, ob ich ihn namentlich nennen darf. Er hat zugestimmt und mir seine Gründe noch einmal aufgeschrieben. Es waren zwei, und ich halte beide bis heute für berechtigt.

Der erste: Die Modelle waren damals nicht reif genug. Das sehe ich heute genauso. Wer 2024 mit dem arbeitete, was es 2024 gab, kam zu genau diesem Ergebnis. Der Unterschied zwischen uns lag nicht in der Beobachtung. Er lag darin, welchen Schluss wir daraus gezogen haben.

Der zweite Grund wiegt schwerer: Diese Werkzeuge Menschen ohne Erfahrung in die Hand zu geben, ist gefährlich. Ein Modell produziert plausiblen Code schneller, als eine unerfahrene Person ihn prüfen kann, und das Prüfen ist genau der Teil, der Erfahrung verlangt. Dieser Einwand galt 2024, er gilt 2026, und er ist einer der Gründe, warum in diesem Buch so viel über Leitplanken und so wenig über Prompts steht.

Dann kam der Teil seiner Antwort, mit dem ich nicht gerechnet hatte. Die fuseki entwickelt inzwischen produktiv mit Claude, nach einer Testphase, die im Jahr zuvor begonnen hatte. Mich interessiert das Wie: erst den erfahrenen Entwicklerinnen und Entwicklern in die Hand geben, dann testen, dann verbessern, dann verbreitern. Das ist ziemlich genau der Weg, den Kapitel 21 empfiehlt, und wir haben uns darüber nie abgestimmt.

Beide Positionen waren zu ihrer Zeit richtig. Getrennt hat uns keine Haltung, getrennt haben uns der Zeitpunkt und der Weg der Einführung. Wer heute ablehnt, was 2024 abzulehnen war, argumentiert mit einem Werkzeugstand, den es nicht mehr gibt. Wer heute alles freigibt, weil die Modelle besser geworden sind, übersieht den zweiten Einwand, der nie erledigt war.

## Warum dieses Buch

Ich habe zunächst ein Whitepaper geschrieben, das den Prozess beschreibt: Werte, Rollen, Artefakte, neun Schritte, drei menschliche Stop-Punkte. Die häufigste Rückmeldung darauf war nicht Widerspruch. Sie war eine Frage: Wie mache ich das konkret?

Dieses Buch entwickelt das Whitepaper weiter. Es beschreibt denselben Prozess, aber mit den Werkzeugen daneben, mit den Befehlen, die ich tippe, mit den Dateien, die im Repo liegen, und mit den Stellen, an denen ich hingefallen bin.

## Warum zwei Werkzeuge und keine Plattform

Es wäre naheliegend gewesen, eine Plattform zu bauen. Eine Oberfläche, ein Login, alles integriert. Ich habe es nicht getan.

Eine Plattform bindet. Sie bestimmt, wo die Issues liegen, wie das Board aussieht, welche Modelle laufen. Wer sie einmal benutzt, kann den Prozess nicht mehr vom Werkzeug trennen. Und weil sich die Werkzeuglage in diesem Feld monatlich verschiebt, wäre eine Plattform die teuerste Art, sich festzulegen.

Deshalb sind es zwei getrennte Dinge. Das eine ist eine dünne Schicht aus Skills, die den Prozess in Claude Code ausführbar macht. Das andere ist ein Board, auf dem der Status sichtbar wird. Beide sind Open Source, beide sind einzeln benutzbar, und wer den Prozess fahren will, braucht keines von beiden.

Der Name `claude-workflow-kit` klingt nach fester Bindung an Claude Code. So fest ist sie nicht. Skills gibt es dort ebenso wie bei Codex und bei Cursor, nur unter anderem Namen und an anderer Stelle im Repo. Der Prozess, den dieses Buch beschreibt, hängt an keinem dieser Werkzeuge. Er hängt an kleinteiligen Issues, an Tests, an einem zweiten Blick und an drei menschlichen Freigaben. Nichts davon ist herstellergelbunden.

Der Prozess trägt. Die Werkzeuge sind austauschbar. Wer an den Werkzeugen hängt statt am Prozess, hat mich missverstanden.

## Was dieses Buch nicht ist

Es ist keine Einführung in Claude Code und keine in git. Es setzt voraus, dass du Software entwickelst und dass du schon einmal mit einem Sprachmodell gearbeitet hast.

Es ist auch kein Versprechen. Ich verkaufe hier kein Framework, und ich misstraue allen, die das gerade tun. Was ich beschreibe, ist eine Momentaufnahme aus meiner Praxis im Jahr 2026. Die Werkzeuge werden sich ändern. Die drei Stop-Punkte werden bleiben.

## EINLEITUNG

# Wie dieses Buch zu lesen ist

Dieses Buch beantwortet zwei Fragen, und sie richten sich an unterschiedliche Leserinnen und Leser. Die erste: Warum sieht der Prozess so aus? Das ist Teil I. Die zweite: Wie fahre ich ihn? Das sind die Teile II bis V. Wer beide Antworten liest, bekommt das ganze Bild. Wer nur eine liest, sollte wissen, welche.

**Teil I ist der Konzeptteil.** Warum es nur agil geht, warum der Mensch am Steuer bleibt, was aus den zwölf Prinzipien im KI-Zeitalter wird, welche Rollen es braucht. Das ist der Teil für alle, die den Ansatz bewerten, bevor sie ihn einführen. Er kommt ohne Werkzeugnamen aus.

**Teil II stellt die beiden Werkzeuge vor,** das Board und die Skill-Bibliothek, und erklärt, wie sie zusammenspielen. Ich habe ihn kurz gehalten.

**Teil III ist die Praxis.** Das Issue als Quelle der Wahrheit, der Neun-Schritt-Prozess, ein kompletter Tag von morgens bis abends, die Zusammenarbeit mit einem Product Owner, Takt und Events, Iterationsmodelle. In diesem Teil kommen Board und Skills gemeinsam vor, weil ich sie im Alltag auch gemeinsam einsetze.

**Teil IV ist die Technik.** Installation, Konfiguration, Betrieb, die Skills im Einzelnen, Nachtbetrieb, Qualitäts-Gates. Das ist der Teil zum Nachschlagen, nicht zum Durchlesen.

**Teil V handelt davon, den Prozess am Leben zu halten.** Einführung in einem neuen Team, Anti-Patterns, die KI-Retrospektive, Ausblick.

Wer entscheidet, ob das etwas fürs eigene Team ist, liest Teil I und Kapitel 10. Wer morgen anfangen will, liest Kapitel 10, dann Kapitel 21, dann Teil IV nach Bedarf. Wer den Prozess schon fährt und etwas nachschlagen will, benutzt Teil IV und den Anhang.

## Ein Hinweis zur Stimme

Ich schreibe in der ersten Person. Was hier steht, ist nicht der Stand der Forschung, es ist meine Praxis. Ich schreibe dazu, wie sicher ich mir bin, auch dann, wenn die Antwort lautet: nicht sehr. Und wo sich eine Aussage absehbar überholen wird, steht das dabei.

Fachbegriffe benutze ich auf Englisch, wo die Branche sie so verwendet: Issue, Push, Pull Request, Backlog, Commit.

## EINLEITUNG

## Die zwei Kits auf einen Blick

Zwei Werkzeuge tauchen in diesem Buch immer wieder auf. Beide sind von mir, beide sind Open Source, und beide sind optional. Dieser Abschnitt sagt in Kürze, was sie tun, damit die Begriffe ab hier sitzen.

**Das `claude-workflow-kit`** ist eine Bibliothek aus Skills für Claude Code. Jeder Skill entspricht einem Schritt im Prozess: planen, Issues anlegen, implementieren, prüfen, reviewen, pushen, mergen. Dazu kommen Skills, die nicht zu dieser Reihe gehören: für Session-Start und Session-Ende, für die Retrospektive, für die fachliche Vorstufe und für das Prüfen der Dokumente, bevor ich gegen sie implementiere. Das Kit ist keine Plattform und kein Agent. Es ist eine dünne Schicht aus Textdateien und einem Adapter. Ich installiere es mit einem Node-Skript, das ein paar Fragen stellt.

Mit den Skills steht Claude Code nicht allein. Codex kennt dasselbe Konzept, Cursor kennt es, und die anderen Werkzeuge ziehen nach. Format und Ort unterscheiden sich: Claude Code liest `.claude/skills/`, Codex liest `AGENTS.md`, Cursor liest `.cursor/rules`. Der Inhalt einer Anweisung ist derselbe.

Ich habe die Skills deshalb von Anfang an so geschrieben, dass keine Annahme über das Werkzeug darin steckt. Alles Werkzeugspezifische liegt in der Config und im Adapter. Wer das Kit auf eine andere Engine übertragen will, übersetzt das Format und behält den Inhalt. Kapitel 18 sagt, warum das Kit diese Übersetzung nicht mitliefert.

**Das `kanban-kit`** ist ein selbst-hostbares Board. Projekte, Boards, Spalten, Karten, Epics, Labels, Kommentare, Anhänge, ein Kennzahlen-Dashboard und eine rollenbasierte Rechteverwaltung. Es läuft in Docker auf einem eigenen Server.

### Was wirklich nötig ist

Das `kanban-kit` ist für den Prozess nicht nötig.

Das `claude-workflow-kit` arbeitet genauso gut gegen GitHub Projects, gegen GitLab oder in einem vollständig lokalen Modus, in dem die Issues als Markdown-Dateien im Repo liegen und es gar kein Board gibt. Ein Board-Adapter schirmt die Skills von der Plattform ab. In der Konfiguration lege ich fest, welche es ist, und dort ändere ich sie auch wieder.

Und genau genommen ist auch das `claude-workflow-kit` nicht nötig. Ich habe den Prozess ein halbes Jahr lang ohne Kit gefahren, mit Konventionsdateien und Trigger-Phrasen. Das Kit macht es zuverlässiger: Eine Anweisung, die als Skill im Dateisystem liegt, greift im richtigen Moment und geht nicht in einem langen Chatverlauf unter. Der Prozess funktioniert auch ohne Kit. Mit Kit hält er im Alltag durch.

Wenn du nur eine Sache aus diesem Buch mitnimmst, dann den Prozess: kleinteilige Issues, das Issue als Quelle der Wahrheit, Test-First, ein zweites Modell im Review, und drei Stellen, an denen ein Mensch entscheidet.

|                       | <code>claude-workflow-kit</code>                | <code>kanban-kit</code>                |
|-----------------------|-------------------------------------------------|----------------------------------------|
| Was es ist            | Skills für Claude Code                          | Selbst-hostbares Board                 |
| Wo es läuft           | Im Projekt-Repo                                 | Docker auf eigenem Server              |
| Nötig für den Prozess | Nein, aber hilfreich                            | Nein                                   |
| Alternativen          | Konventionsdateien und Trigger-Phrasen von Hand | GitHub Projects, GitLab, lokaler Modus |
| Lizenz                | Open Source                                     | Open Source                            |

## TEIL I

# Die Idee und die Grundsätze

Dieser Teil beantwortet in vier Kapiteln die Frage, warum der Prozess so aussieht, wie er aussieht. Werkzeugnamen kommen dabei noch keine vor.

Kapitel 1 begründet, warum es im KI-Zeitalter aus meiner Erfahrung nur noch agil geht und warum ich Scrum nicht mehr für die richtige Herangehensweise dafür halte. Dort stehen auch die zwölf agilen Prinzipien und was im KI-Modus aus jedem einzelnen wird. Kapitel 2 beschreibt, was ein Sprachmodell tatsächlich tut, und leitet aus fünf Eigenschaften der Architektur die Regeln ab, die im Rest des Buches als Praxis auftauchen. Kapitel 3 zeigt, weshalb der Mensch am Steuer bleibt, und zwar als Folge dieser Eigenschaften. Kapitel 4 beschreibt die Rollen, die es dafür braucht, und eine, die kaum jemand nennt.

Dieser Teil steht vor allem wegen Kapitel 2 am Anfang. Wer den Prozess ohne die Architektur liest, hält ihn für eine Ansammlung von Vorsichtsmaßnahmen. Wer die Architektur kennt, kann die Regeln selbst herleiten und im Zweifel begründet abweichen.

Wer entscheiden will, ob das im eigenen Team taugt, findet hier die Begründung. Wer es schon entschieden hat, kann direkt zu Kapitel 10 springen und einen Arbeitstag ansehen.

## KAPITEL 1

# Es geht nur agil

Wer heute Software entwickelt und dabei mit einem Sprachmodell arbeitet, hat bei der Methode keine echte Wahl. Schwergewichtige Vorgehensmodelle scheitern in der KI-augmentierten Welt schneller und gründlicher, als sie es je in der rein menschlichen Welt getan haben. Wasserfall, V-Modell mit langen Implementierungsphasen, Phasen-Gate-Modelle mit halbjährlichen Release-Zyklen: Diese Ansätze waren schon vor der KI brüchig. Im KI-Zeitalter halte ich sie für unhaltbar.

Der Grund ist praktisch. Schwergewichtige Methoden setzen auf Plan-Stabilität. Wer so arbeitet, entscheidet einmal, dokumentiert ausführlich und implementiert lange. Diese Annahme funktioniert nur, wenn sich die Welt zwischen Plan und Auslieferung nicht zu schnell ändert. Sprachmodelle erweitern ihre Fähigkeiten aber monatlich, und eine sinnvolle Aufgabe ist oft in Stunden erledigt statt in Tagen. Da ist Plan-Stabilität eine Illusion.

## Drei Verschiebungen erzwingen Agilität

**Die Implementierung wird billig.** Ich habe im vergangenen Jahr eine Keycloak-Integration gebaut. Null Erfahrung. Nach vier Stunden lief sie. Vor der KI hätte ich dafür zwei bis vier Wochen eingeplant, mit Einarbeitung eher mehr. Wenn vier Stunden reichen, ist Implementierung nicht mehr die knappe Ressource. Knapp sind nun das Verstehen, das Reviewen, das Entscheiden. Diese Tätigkeiten sind iterativ. Sie funktionieren in kurzen Schleifen mit Feedback, nicht in langen Phasen mit Übergaben.

**Die Werkzeuglage ist dynamisch.** Was heute mit dem einen Modell schwer ist, ist morgen mit einem neuen trivial. Wer einen Sechs-Monats-Plan aufstellt, plant gegen eine Werkzeug-Generation, die zur Hälfte der Laufzeit überholt sein wird. Auf Veränderung zu reagieren ist hier nichts, was man sich aussuchen kann. Die Veränderung ist der Normalzustand.

**Die Erwartungen haben sich angepasst.** Dass ein sichtbarer Prototyp innerhalb von Tagen steht, ist heute normal. Die Kundschaft hat einen Prototyp ohnehin schon immer für fast fertig gehalten. Heute liegt sie damit nicht mehr weit daneben: Aus Wochen oder Monaten Wartezeit sind Tage geworden.

## Was vom Manifest 2001 trägt

Im Juni 2026 habe ich auf meinem Blog den Beitrag “Scrum ist tot. Was kommt danach?” veröffentlicht. Es gab viele Kommentare, und etliche lasen darin, ich hätte die Agilität gleich mitbeerdigt. Dabei steht das Gegenteil schon im Beitrag, und ich sage es auch hier: Agilität ist wichtiger denn je. Das Agile Manifest von 2001 formuliert vier Wertepaare und zwölf Prinzipien, und beide haben in der KI-Ära nichts an Gültigkeit verloren.

Individuen und Interaktionen über Prozesse und Werkzeuge: gerade weil das Werkzeug jetzt so stark geworden ist, zählt umso mehr, wie Menschen miteinander und mit ihm umgehen. Funktionierende Software über ausführliche Dokumentation: gerade weil die KI Dokumentation auf Knopfdruck produziert, darf sie nicht zum Selbstzweck werden. Zusammenarbeit mit der Kundschaft über Vertragsverhandlung: gerade weil schneller Output das Risiko erhöht, an der Kundschaft vorbei zu liefern. Reagieren auf Veränderung über das Befolgen eines Plans: gerade weil sich die Möglichkeiten monatlich verschieben.

Anforderungsänderungen sind willkommen, auch spät? Das gilt heute noch mehr, weil eine Änderung nur noch einen Bruchteil des früheren Aufwands kostet. Ständige Aufmerksamkeit auf technische Exzellenz? Die ist heute Pflicht, gerade weil KI schnell schlechten Code produziert, wenn niemand am Steuer sitzt.

Das Manifest gilt heute noch. Scrum war keine schlechte Implementierung. Sie passt aus meiner Sicht nur nicht mehr.

## Die zwölf Prinzipien im KI-Modus

Am Anfang habe ich überlegt, ob die zwölf Prinzipien im KI-Zeitalter neu geschrieben werden müssen, und im ersten Entwurf dieses Buches hatte ich das tatsächlich getan. Bei genauem Hinsehen brauchen sie keine neue Fassung. Sie brauchen, wie immer, den Blick darauf, wie ein Team sie umsetzt. Ich habe deshalb die zwölf originalen Sätze genommen und an jeden dieselbe Frage gestellt: Was verlangt er weiterhin, was wird leichter, und wo muss ich neu aufpassen?

Das Ergebnis ist eindeutiger, als ich erwartet hatte. Kein Prinzip ist im KI-Modus schwerer umzusetzen. Die meisten sind leichter. Bei einigen verschiebt sich die Skala oder die Form.

| Prinzip                               | Was sich im KI-Modus verschiebt                                                           |
|---------------------------------------|-------------------------------------------------------------------------------------------|
| 1. Früh und kontinuierlich ausliefern | Aus zweiwöchigen Releases wird tägliche Auslieferung. Die Skala, nicht das Ziel.          |
| 2. Späte Änderungen willkommen heißen | Wird substantiell billiger. Bedingung: Die Änderung wandert ins Issue, nicht in den Chat. |
| 3. Häufig ausliefern                  | Tages-Iterationen sind der neue Default, Vier-Stunden-Takte sind möglich.                 |
| 4. Fachseite und Entwicklung täglich  | Wird konkreter: Die Diskussion geht vom Abstrakten ins Sichtbare, oft am selben Tag.      |
| 5. Motivierte Individuen, Vertrauen   | Der Anspruch an die Menschen steigt, der Bedarf an Mikromanagement nicht.                 |
| 6. Direkte Kommunikation              | Zwischen Menschen weiter überlegen. Der Mensch-Modell-Anteil ist notwendig schriftlich.   |
| 7. Funktionierende Software als Maß   | Wird dringlicher als je zuvor, weil die Menge explodiert und nichts mehr aussagt.         |
| 8. Nachhaltige Geschwindigkeit        | Leichter und gefährdeter zugleich. Das Werkzeug wird nie müde, der Mensch schon.          |

| Prinzip                                | Was sich im KI-Modus verschiebt                                                        |
|----------------------------------------|----------------------------------------------------------------------------------------|
| 9. Technische Exzellenz                | Wird in beide Richtungen verstärkt. Ohne Leitplanken entstehen sehr schnell Stapel.    |
| 10. Einfachheit, die Kunst wegzulassen | Der Aufwand sinkt, die nötige Disziplin steigt. Das einzige Prinzip mit dieser Schere. |
| 11. Selbstorganisierende Teams         | Selbstorganisation bleibt zwischen Menschen. Das Modell ist nicht Teil davon.          |
| 12. Regelmäßige Reflexion              | Bekommt eine zweite Ebene: die KI-Retrospektive als eigenes Format.                    |

Wenn ich mit Teams Agilität eingeführt habe, habe ich immer dasselbe gesagt: Schaut euch die zwölf Prinzipien an und überlegt, was ihr konkret tun müsst, damit sie tragen. Genau das steht auch jetzt an. Neu ist nur die Frage, was jedes Prinzip im KI-Zeitalter verlangt. Wer die Prinzipien 2001 ernst genommen hat, ist heute gut gerüstet. Wer sie als Lippenbekenntnis behandelt hat, wird seine Versäumnisse jetzt deutlicher merken. Die KI verstärkt, was sie vorfindet.

## Scrum passt im KI-Zeitalter nicht mehr, Agilität schon

Im Blogbeitrag habe ich diese Unterscheidung unscharf gelassen und das Missverständnis damit provoziert. Scrum und Agilität sind nicht dasselbe. Scrum ist eine Implementierung agiler Prinzipien für eine bestimmte Welt, und für diese Welt hat sie funktioniert. Seine Events setzen ein vollständig menschliches Team voraus. Seine Sprints sind in Wochen gerechnet, nicht in Stunden. Sein Backlog ist statisch, gemessen an dem, was ein Sprachmodell an einem Vormittag an Vorschlägen liefert.

Scrum geht von Sprints zwischen einer und vier Wochen aus, in der Praxis sind es meist zwei. Diese Annahme stimmt aus meiner Sicht nicht mehr. Ein zweiwöchiger Sprint ist damit nicht falsch, er ist grobkörnig geworden. Mir hat 2025 jemand aus einem Scrum-Team berichtet, dass ein Feature bereits fertig war, bevor es als Ticket erstellt, im Sprint Planning erklärt, geschätzt und zugewiesen war. Nicht halb fertig. Fertig, getestet, dokumentiert, gemergt. Ich setze eine Anforderung heute schnell

ler um, als ich sie planen kann. Damit bremst der Sprint-Plan genau die Anpassungsfähigkeit aus, die er herstellen soll.

Auch Boris Gloger sagt, dass die alte, oberflächliche Art, Scrum zu betreiben, am Ende ist. Mich stört etwas anderes: Wer jeden Einwand mit “das ist dann kein richtiges Scrum” beantwortet, beendet die Diskussion, bevor sie anfängt. Ein Framework, das sich jeder Kritik entzieht, indem es sich selbst neu definiert, ist kein Framework mehr. Es ist eine Haltung.

## Der Takt ist das Issue

Der Takt ist jetzt das Issue, nicht mehr der Sprint. Deshalb arbeite ich mit einem Board als zentralem Artefakt und mit dem Issue als Quelle der Wahrheit. Prinzipien statt Prozesse. Flow statt Sprints.

Ich benutze ein Taskboard, kein Kanban-Board im strengen Sinn. Kanban ist ein Pull-System. Die nachgelagerte Station zieht sich Arbeit, wenn Kapazität frei wird, und WIP-Limits erzwingen genau das. Bei mir läuft es gemischt: Ich schiebe Issues nach Ready, das ist ein Push, und ich entscheide dabei über die Menge. Erst danach zieht sich das Modell das oberste Issue, das ist ein Pull. Die Spalten sehen aus wie Kanban, die Mechanik ist es nur zur Hälfte.

Das Werkzeug heißt trotzdem kanban-kit. Namen setzen sich nicht nach Lehrbuch durch. Wo ich im Buch die Mechanik meine, spreche ich vom Board oder vom Taskboard, und wo eine Kanban-Regel wirklich gilt, sage ich es dazu.

Drei Dinge bleiben: Stakeholder-Reviews, allerdings regelmäßig statt sprintgebunden. Retrospektiven, allerdings ergänzt um eine eigene KI-Retrospektive. Issue-Grooming, allerdings im Dialog mit dem Modell statt im Meeting. Kapitel 12 beschreibt diese Events im Einzelnen.

Das ist keine Absage an Planung. Der Plan ist Schritt 2 des Prozesses und der meistunterschätzte Schritt überhaupt.

## KAPITEL 2

# Was ein Sprachmodell tut

Ich habe lange die falsche Frage gestellt. Ich habe gefragt, was das Modell weiß. Die brauchbare Frage ist, was es tut.

Als ich 2024 anfang, mit Sprachmodellen zu arbeiten, hatte ich eine unausgesprochene Vorstellung im Kopf: Da ist etwas, das Programmieren gelernt hat und mir hilft. Diese Vorstellung erklärt nicht, warum die Ergebnisse so ausfielen, wie sie ausfielen. Erst als ich mich damit beschäftigt habe, wie ein Transformermodell tatsächlich arbeitet, ergaben die Fehler ein Muster. Und aus diesem Muster folgt fast alles, was in diesem Buch an Regeln steht.

Dieses Kapitel ist keine Einführung in maschinelles Lernen. Es enthält keine Formeln und erklärt Attention nicht im Detail, dafür gibt es bessere Bücher. Es beschreibt fünf Eigenschaften der Architektur und sagt, welche Regel aus jeder folgt.

## Das nächste Token

Ein Transformermodell tut genau eine Sache: Es sagt das nächste Token voraus.

Der Ablauf ist unspektakulär. Der Text wird in Tokens zerlegt, also in Wortstücke. Das Modell berechnet über den gesamten bisherigen Text eine Wahrscheinlichkeitsverteilung für das nächste Token. Aus dieser Verteilung wird eines gezogen, nicht zwangsläufig das wahrscheinlichste. Das gezogene Token wird angehängt, und die Rechnung beginnt von vorn. So entsteht Satz für Satz, Zeile für Zeile, Datei für Datei.

Das ist alles. Es gibt keinen Plan, der vor dem ersten Token existiert, und keine Absicht, die über das nächste Token hinausreicht. Was wie ein Gedankengang aussieht, ist eine sehr lange Folge einzelner Vorhersagen. Jede bekommt den bisherigen Text als Eingabe.

**Das Modell setzt ein Gespräch nicht fort, es liest es bei jedem Zug komplett neu.** Bis ich das verstanden hatte, hat es gedauert, und im Alltag wiegt es schwerer als der erste Punkt. Für mich fühlt sich der Chat wie eine Fortsetzung an. Für das Modell ist jeder Zug das erste Mal: Ich gebe den gesamten bisherigen Verlauf noch einmal mit, und das Modell verarbeitet ihn von vorn. Der Zustand liegt nicht im Modell, er liegt im Text, den ich ihm schicke. Wenn sich ein Werkzeug an frühere Sessions zu erinnern scheint, reicht eine Schicht darüber sie als Text wieder hinein.

Daraus folgt die Lösung, die in diesem Buch immer wiederkehrt: Was das Modell wissen soll, schreibe ich auf, halte es aktuell und übergebe es im richtigen Moment.

## Fünf Eigenschaften, fünf Regeln

**Erstens: Es optimiert auf Plausibilität, nicht auf Korrektheit.** Es lernt im Training, welches Token als Nächstes wahrscheinlich ist, nicht ob eine Aussage stimmt. Beides fällt oft zusammen, weil korrekter Text im Trainingsmaterial häufig vorkommt, aber es ist nicht dasselbe. Deshalb schlägt ein Modell SQL-String-Konkatenation vor. Es missachtet damit keine Sicherheit. Diese Zeile sieht in Millionen Trainingsbeispielen einfach so aus. Aus dieser Eigenschaft folgt Test-First. Der Test bindet die Ausgabe an etwas, das nicht aus dem Modell selbst kommt.

**Zweitens: Zwischen zwei Aufrufen gibt es keinen Zustand.** Nichts bleibt im Modell. Was ich gestern besprochen habe, existiert heute nur, wenn es in dem Text steht, den ich mitschicke. Aus dieser Eigenschaft folgt der ganze Kontext-Apparat: das Issue als Quelle der Wahrheit, die Konventionsdateien, der Vault. Das Gedächtnis liegt in Dateien. Kapitel 7 und 17 beschreiben das im Einzelnen.

**Drittens: Das Fenster ist begrenzt, und darin gewichtet das Modell um.** Solange der Text hineinpasst, ist alles noch da. Aber je länger er wird, desto weniger Gewicht bekommt der Anfang, weil sich die Aufmerksamkeit auf mehr Text verteilt. Wird es zu viel, fasst die Werkzeugschicht Älteres zusammen, und hier geht tatsächlich etwas verloren. Aus dieser

Eigenschaft folgen die Skills: Ich zerlege den Prozess in einzelne Stücke und lade sie im richtigen Moment, statt alles an den Anfang in eine lange Datei zu schreiben. Und sie erklärt, warum der Nacht-Runner pro Issue eine frische Session startet.

**Viertens: Das Modell markiert nicht, wo sein Muster aufhört.** Nichts darin trennt “das habe ich tausendmal gesehen” von “das rate ich gerade”, und nichts schreibt den Unterschied dazu. Beides kommt im selben Ton, mit derselben Flüssigkeit. Aus dieser Eigenschaft folgt das Zwei-Modelle-Review und die menschliche Verifikation. Es gibt kein Signal, auf das ich warten könnte. Also sehe ich selbst hin.

**Fünftens: Es ist nicht deterministisch.** Weil bei jedem Token der Zufall mitentscheidet, ergibt dieselbe Eingabe nicht zwangsläufig dieselbe Ausgabe. Aus dieser Eigenschaft folgt, dass ein Modell kein Gate sein kann. Ein Gate, das beim zweiten Versuch durchwinkt, ist keines. Qualitäts- und Sicherheitsprüfungen gehören deshalb in deterministische Werkzeuge, und Kapitel 20 handelt von nichts anderem.

| Eigenschaft der Architektur                                | Was daraus im Prozess folgt                                |
|------------------------------------------------------------|------------------------------------------------------------|
| Sagt das nächste Token voraus, optimiert auf Plausibilität | Test-First, Akzeptanzkriterien, manuelle Verifikation      |
| Kein Zustand zwischen Aufrufen                             | Issue als Quelle der Wahrheit, Konventionsdateien, Vault   |
| Begrenztes Fenster, Umgewichtung, Kompaktierung            | Skills statt einer langen Datei, frische Session pro Issue |
| Markiert die eigene Wissensgrenze nicht                    | Review durch ein zweites Modell, menschliche Freigabe      |
| Nicht deterministisch                                      | Gates in deterministischen Werkzeugen, nicht im Modell     |

## Warum die alten Leitplanken passen

Erst neulich hat mir jemand aus einem Entwicklungsteam gesagt: “KI funktioniert nicht.” Ich habe nicht widersprochen, ich habe gesagt: Zeig

mal. Gezeigt wurden mir Trivialitäten, ein Komma, das die KI in ein Semikolon geändert hatte. Ein ernsthafter Versuch war das nicht.

Dahinter steckt ein Prozessproblem. Niemand käme auf die Idee, Entwicklerinnen und Entwickler ohne Architekturkonzept, ohne Konventionen und ohne Tests loszuschicken und danach zu urteilen, Softwareentwicklung produziere grundsätzlich Schrott. Bei KI tun das fast alle.

Gegen die fünf Eigenschaften oben helfen Testpyramide, Architekturregeln, Konventionen, Reviews und statische Analyse. Diese Konzepte sind keine Bürokratie. In ihnen steckt die Erfahrung aus Projekten, die ohne sie gescheitert sind. Sie sind nicht richtig, weil sie alt sind. Sie sind richtig, weil sie genau die Fehlerklasse adressieren, die ein Sprachmodell produziert. Wäre die Fehlerklasse eine andere, würden andere Leitplanken passen.

Eine Regel im Kopf ist ein Wunsch. Eine Regel im Build wirkt. Wer ArchUnit einsetzt und Architekturregeln als automatische Tests formuliert, bekommt vom Modell keinen Code mehr, der die Schichtengrenzen verletzt. Der Build schlägt fehl. Das Modell muss es richtig machen. Leitplanken müssen scheitern können, nicht argumentieren. Regeln gehören in Dateien, nicht in Köpfe.

## Was ich dabei korrigieren musste

Ich habe für das fehlende Gedächtnis lange ein Bild benutzt, das nur halb stimmt: die anterograde Amnesie. Der bekannteste Fall ist Henry Molaison, der nach einer Hirnoperation jahrzehntelang keine neuen Erinnerungen mehr bilden konnte. Intelligent, freundlich, hilfsbereit, und doch kein Gestern.

Zwischen zwei Sessions trifft das Bild. Innerhalb einer Session ist es falsch. Wer glaubt, das Modell habe dort etwas vergessen, wiederholt sich. Wer weiß, dass es umgewichtet hat, räumt stattdessen den Kontext auf oder startet neu. Die erste Reaktion hilft wenig, die zweite zuverlässig.

Diese Korrektur zeigt, worauf es hinausläuft: Eine ungenaue Vorstellung davon, wie ein Modell arbeitet, führt zu Vorkehrungen, die plausibel wirken und nicht wirken.

## **Was hier bewusst fehlt**

Diese Beschreibung ist vereinfacht. Sie lässt aus, was für die Praxis dieses Buches nichts ändert: wie Attention rechnet, wie Post-Training und Verstärkungslernen das Verhalten formen, wie Mixture-of-Experts oder Reasoning-Modelle den Ablauf verändern. Manches davon verschiebt die fünf Eigenschaften ein Stück. Reasoning-Modelle etwa erzeugen vor der Antwort eine längere interne Ableitung, was die vierte Eigenschaft abschwächt, aber nicht aufhebt.

Die Details unterscheiden sich zwischen Anbietern und ändern sich schneller, als ein Buch nachziehen kann. Die Ebene darüber, die fünf Eigenschaften und die Regeln daraus, ändert sich langsamer. Deshalb steht dieses Kapitel im Konzeptteil.

Die KI verstärkt die Arbeitsweise, die sie vorfindet. In keiner Architekturbeschreibung steht dieser Punkt, und trotzdem folgt er aus ihr. Sie setzt plausibel fort, was ich ihr gebe, und meldet nicht, wenn es schlecht ist. Wer sauber arbeitet, wird schneller sauber. Wer schlampig, wird schneller schlampig. Die KI ist kein Korrektiv, und deshalb handelt das nächste Kapitel davon, wer hier das Steuer hält.

## KAPITEL 3

# Der Mensch bleibt am Steuer

Ich spiele Schach. Wenn ich am Brett sitze und am Zug bin, ist die wichtigste Frage nie, wie gut mein Gegner spielt. Die wichtigste Frage ist, wer die Initiative hat. Solange ich sie habe, bestimme ich Tempo und Richtung der Partie. Gebe ich sie ab, ist jeder Zug nur noch Reaktion, und Reaktion kostet Zeit, kostet Optionen, kostet am Ende oft die Partie.

Genau dieses Muster begegnet mir seit Monaten in der Arbeit mit KI-Werkzeugen, und es sieht zunächst harmlos aus. Das Modell beginnt, eigene Annahmen zu treffen. Es schlägt Refactorings vor, die ich nicht angefragt habe. Es wählt Bibliotheken, die ich nicht evaluiert habe. Es erweitert den Scope, weil es das konsistenter findet. Jeder einzelne Vorschlag wirkt für sich plausibel, aber die Summe führt weg von meinem Ziel, dorthin, wohin das Modell den Code gedrängt hat. Das fühlt sich produktiv an, es passiert ja viel. Es ist der Moment, in dem die Partie kippt.

Ein Sprachmodell ist kein passives Werkzeug wie eine IDE. Es ist etwas, das Vorschläge in den Raum stellt, die plausibel klingen und sofort umsetzbar wirken. Wenn ich nicht aufpasse, sind sie auch sofort umgesetzt. Aus meiner Anfangszeit kenne ich das gut: Ich habe gesagt, mach mal einen Plan. Das Modell hat den Plan geliefert, gefragt, ob es loslegen soll, und schon implementiert, bevor ich geantwortet hatte. Deshalb reicht es nicht, sorgfältig zu sein. Ich brauche eine Struktur, die die Initiative dort hält, wo sie hingehört.

## Die KI als Implementations-Engine

Ich behandle die KI konsequent als Werkzeug. Das sagt, wie ich sie behandeln will, nicht, was sie ist. Ich behaupte nicht, ein Sprachmodell sei nur die nächste Stufe nach vi, Emacs und IntelliJ. Es ist ein Bruch, keine Fortsetzung. Behandeln will ich es trotzdem wie ein Werkzeug, weil nur so klar bleibt, wer die Verantwortung trägt.

Am besten trifft es der Begriff Implementations-Engine: ein starker Motor, der eine Hand am Steuer braucht. Das Bild hat eine Grenze. Ein Motor hat keine Meinung zur Route. Ein Sprachmodell hat eine, es schlägt eine Architektur vor. Genau deshalb ist die Hand am Steuer nötig.

Warum lässt sich Verantwortung nicht an ein Modell abgeben? Das übliche Argument lautet, Verantwortung sei grundsätzlich unteilbar. Das ist nicht meins: An Menschen delegiere ich Verantwortung ständig. Delegation ist ein Vertrag zwischen Personen, die Konsequenzen tragen können. Ein Modell kann keine tragen.

## **Zwei Modi des Denkens, und das Modell hat nur einen**

Zurück ans Brett. Ich bin ein fortgeschrittener Anfänger, und trotzdem kenne ich beide Modi. In der Eröffnung, in den ersten zehn Zügen, hatte ich die Stellung oft schon auf dem Brett. Das Mustergedächtnis springt an, und ich kann sofort ziehen. Im Mittelspiel springt es manchmal noch an. Aber irgendwann kommt der Moment, in dem ich wirklich rechnen muss, Zug für Zug, Variante für Variante.

Im ersten Modus ist das Modell stark. Den zweiten Modus kennt es nicht. Und es stockt nie. Es markiert die Stelle nicht, an der sein Muster aufhört. Es antwortet auf die bekannte und auf die neue Stellung im selben Ton, mit derselben Sicherheit. Es versteckt die Lücke nicht, es füllt sie.

Die Gefahr ist deshalb nicht das Tempo des Modells. Die Gefahr ist, dass mir ein plausibler Zug genügt, obwohl die Stellung neu ist. Ich sehe nur eine glatte Antwort und halte sie für eine geprüfte. Wer das Prüfen überspringt, spielt in einer neuen Stellung einen Zug aus einer alten Partie.

In genau diesen Stellungen übernehme ich. Schneller bin ich nicht. Aber ich bin der Einzige im Raum, der überhaupt in den zweiten Modus schalten kann. Ein Modell macht den ersten Modus billig. Den zweiten macht es wichtiger, weil es ihn nicht kann.

## Die Denkarbeit wandert nach vorn

Die Denkarbeit wandert nach vorn. Dieser Satz trägt meinen ganzen Prozess.

An einem Tag im Juli habe ich eine Stunde an einer Spezifikation gearbeitet. Eine Stunde für einen Text, den am Ende niemand sieht außer mir und dem Modell. Entstanden ist dabei kein Feature und keine Zeile, die läuft. Sofort in die Tasten zu hauen hätte sich produktiver angefühlt. Es war trotzdem die wichtigste Stunde des Tages.

Das Verfahren dabei ist unspektakulär. Ich lasse das Modell eine erste Variante schreiben. An ihren Annahmen sehe ich meine eigenen Lücken. Vor einem leeren Dokument sehe ich sie nicht, ich halte sie für Klarheit, weil ich sie im Kopf schon gefüllt habe. Die erste Variante ist falsch, aber sie ist konkret. Die nächste ist besser, weil ich die vorige gesehen habe. Nach einer Stunde steht ein Text, der trägt.

Das ist ausdrücklich nicht die Rückkehr zum Lastenheft. Ein Issue hat vier Abschnitte und passt auf eine Seite. Geklärt wird trotzdem mehr. Das Nachdenken wird nicht mehr, es rückt nach vorn.

Der verbreitete Reflex geht in die andere Richtung: Je komplexer die Aufgabe, desto mehr gebe ich ab. Das ist falsch herum. Wer bei Komplexität abgibt, gibt genau dort ab, wo die stillen Annahmen am teuersten werden. Ich kann nicht prüfen, was ich nicht verstehe. Das Bauen nimmt mir das Modell ab, das Verstehen nicht. Verstehen täuscht es mir nur vor, wenn ich es lasse.

Wenn ich nicht mehr durchsteige, hilft mir kein Modell. Dann fehlt mir keine Implementierung. Dann fehlt mir die Spezifikation.

## Die drei Stop-Punkte

Aus dieser Haltung folgen drei Stellen im Prozess, die ich nicht automatisiere. Ich habe mein Werkzeug in wenigen Wochen fünfmal umgebaut,

und diese drei Punkte haben sich in keiner der fünf Fassungen verändert.

**Das Ready.** Ich entscheide, welche Issues in den nächsten Durchlauf kommen. Am Board heißt das: Ich ziehe sie nach Ready. Darin liegt die Planung, wie viel Arbeit auf einmal, welche Priorität, welche Abhängigkeiten. Die KI zieht nie eigenmächtig etwas nach Ready.

**Der Push.** Ich gebe frei, bevor Code meinen Rechner verlässt. Jeder Batch braucht eine eigene Freigabe. Eine frühere Freigabe in derselben Session gilt nicht für neue Commits.

**Der Merge.** Ich bringe Code nach production, nachdem ich auf dem Test-Server geprüft habe.

Alles dazwischen kann die KI machen. Diese drei Punkte nicht. Dort liegt die Verantwortung.

Es gibt einen vierten, und er ist mir erst aufgefallen, als ich ihn schon gebaut hatte: den Git-Tag. Ein Tag markiert eine Veröffentlichung, und Veröffentlichungen gebe ich selbst frei, genau wie die drei Punkte davor. Beim Push auf main entsteht deshalb bewusst keiner, dort entstehen interne Patch-Stände, die niemand veröffentlicht. Ich zähle ihn nicht mit, wenn ich von den Stop-Punkten rede, weil er keine Trigger-Phrase hat und niemanden aufhält. Der Gedanke dahinter ist aber derselbe.

Dazu gehört eine Abgrenzung: Kontrolle heißt, dass die Freigabe ein bewusster Akt bleibt. Gerade dann, wenn der Output gut ist.

Die Trigger-Phrasen für Ready, Push und Merge sind Mikro-Verträge. Ohne sie passiert nichts, mit ihnen passiert genau das, was im Issue steht. Sie gehören deshalb in die Sicherheitsarchitektur, nicht in die Bedienungsanleitung.

## Je autonomer die KI, desto strenger die Stop-Punkte

Das ist die Grundregel, und sie wird in dem Maß wichtiger, in dem die Werkzeuge autonomer werden. Nachtläufe sind gut. Nachtläufe ohne Stop-Punkte sind es nicht. Ohne Stop-Punkte fallen Entscheidungen, die

kein Mensch getroffen hat, und je länger die KI allein arbeitet, desto mehr davon sammeln sich an.

Wie nötig die Regel ist, habe ich an einem Tag gelernt, an dem das Modell zweimal den Workflow gebrochen hat. Einmal hat es einen Fix an einer Drop-Zone ohne Plan und ohne Ready direkt implementiert und gepusht. Einmal ist es nach einem Bug in Bewegung geblieben, statt anzuhalten. Ich habe damals geschrieben: "Du bist wieder vollständig im Auto-Modus. Das ist nicht ok." Beide Male war der Code danach in Ordnung. Es war kein technischer Bug, es war ein Prozess-Bug. Also habe ich den Workflow zurückgerollt, nicht den Code.

Daraus folgt auch meine Rolle im Alltag. Ich muss das Modell aktiv einbremsen. Es produziert, es macht weiter, es nimmt die nächste Aufgabe, bevor die letzte verifiziert ist. Der Mensch ist nicht der Antreiber. Der Mensch ist die Bremse.

Schnelligkeit ohne Stop-Punkte produziert das Falsche zuverlässig. Ein Plan, der hängt, ist immer besser als ein Commit, der schon irgendwie passt.

Wo die Schwelle genau verläuft, entscheidet die Verantwortung, nicht der Aufwand. Ein Beispiel dafür steht in Kapitel 19: Nachts darf ein Modell feststellen, dass an einem Issue nichts zu ändern ist, und das protokollieren. Ändern darf es die Anforderung nicht. Wer diese Grenze sauber zieht, kann viel automatisieren, ohne einen Stop-Punkt aufzuweisen.

## KAPITEL 4

# Rollen im KI-augmentierten Team

In meinem Prozess haben sich drei zentrale menschliche Rollen herausgebildet. Das ist eine Beobachtung aus meiner Praxis, kein Gesetz: Wer gute Gründe für eine vierte Rolle hat, soll sie besetzen. Ich komme mit dreien aus. Daneben benutze ich die KI als Werkzeug. Sie hat ihre Eigenheiten, aber keine Rolle im klassischen Sinne.

### Product Owner: das Was

Das Was gehört der Product-Owner-Rolle: Anforderungen definieren, priorisieren, Issues formulieren oder nach Diktat formulieren lassen. Sie entscheidet, welches Issue als nächstes drankommt und welches warten muss. Sie gibt das Ready und die Trigger-Phrasen für Push und Merge. Arbeitet das Team mit einem Board, schiebt sie die Issues nach Ready, und die KI tut das nie.

Diese Rolle verschwindet im KI-Modus nicht, sie wird wichtiger. Spezifikationsqualität wird zum Engpass. Wer unscharf beschreibt, bekommt unscharfen Code, nur eben in höherem Tempo. Sprint Planning, Stakeholder-Management, Release-Strategie: Was diese Rolle im klassischen Scrum trug, bleibt. Es verteilt sich nur auf engere Takte. In diesem Modus halte ich mehrere Mikro-Plannings pro Tag ab und gebe an einem Tag mehr als ein Release frei, weil die KI mir die operative Last abnimmt.

Für die Arbeit mit einer PO-Rolle hat das Kit eine eigene Schleife mit fachlichen und technischen Issues. Kapitel 11 beschreibt sie.

### Senior Developer: das Wie

Die Senior-Developer-Rolle verantwortet das Wie. Sie prüft Architektur-Vorschläge, bevor sie in Code übergehen. Ein Veto hat sie auf jedes Issue, das technisch unausgegoren ist. Sie entscheidet, wann ein Refactoring nötig ist und wann ein vorhandenes Muster gerade ausreicht. Sie sorgt

dafür, dass die KI nicht in eine eigene Designwelt abdriftet, die zwar funktionierenden, aber inkonsistenten Code produziert.

Diese Rolle ist im KI-Zeitalter anspruchsvoller geworden, nicht entspannter. Wer zehn Jahre Erfahrung mit einem Tech-Stack hat, kann mit der KI in Stunden Dinge produzieren, für die früher Tage nötig gewesen wären. Wer drei Monate Erfahrung hat, kann mit demselben Werkzeug Dinge produzieren, die plausibel aussehen und subtil falsch sind. Senior ist heute, wer das Plausible vom Korrekten unterscheiden kann.

Meine Keycloak-Integration ist dafür das Beispiel, das ich am häufigsten erzähle, und regelmäßig versteht es jemand falsch. Vier Stunden für etwas, wofür ich früher zwei bis vier Wochen eingeplant hätte, klingt nach einem Beweis, dass das jetzt jeder kann. Es ist das Gegenteil. Ich habe geschwitzt. Die kryptische TLS-Fehlermeldung kam nicht von Keycloak, sondern von einem fehlenden Subject Alternative Name im Wildcard-Zertifikat. Ich musste die ganze Fehlerkette im Kopf haben: DNS, Reverse Proxy, Zertifikat, Keycloak-Konfiguration, Client-Konfiguration. Gelöst habe ich das nur, weil ich es beurteilen konnte, und weil ich die KI gebremst habe, wenn sie einem falschen Pfad folgte. Wer das nicht kann, läuft mit ihr gemeinsam in die Sackgasse, nur schneller.

Vier Stunden für Keycloak zeigen nur, dass jemand, der es kann, jetzt schneller ist als je zuvor. Wissen allein trägt das nicht. Ich muss beurteilen können, was ich damit anfangen kann.

## Qualitätsrolle: die Verifizierung

Die Qualitätsrolle verifiziert. Sie lässt die Code-Reviews von einem zweiten Modell anderer Bauart als dem implementierenden laufen, prüft Coverage- und Mutation-Scores, macht Security-Reviews und stellt sicher, dass keine Secrets ins Repository wandern, dass SQL-Queries parametrisiert sind, dass DTOs validiert werden.

Der Grund für das zweite Modell ist Statistik. Ein Modell ist gegenüber seiner eigenen Lösung systematisch unkritisch. Es hat sie soeben selbst formuliert und würde jetzt genauso vereinfachen wie eben. Ein anderes

Modell ist an anderen Stellen blind, und der Mensch dazwischen interpretiert die Diskrepanzen.

Daraus folgt eine Abgrenzung: Drei Modelle sind kein Vier-Augen-Prinzip. Mehr Modelle ersetzen den Menschen in dieser Rolle nicht. Sie liefern ihm Material. Nützlich werden sie, wenn jedes eine andere Frage stellt, und Kapitel 16 zeigt bei `/issue-review`, wie ich das organisiere.

Diese Rolle fällt in der Praxis am ehesten weg. Ich kenne Projekte, in denen es sie schlicht nicht gab, nach dem Motto: Wir haben doch Sonar, die Entwickler werden es schon richten, und was übrig bleibt, findet der Kunde. Im KI-Modus dreht sich das um. Wenn Code in Stunden statt Tagen entsteht, staut sich die Arbeit vor der Verifizierung, und diese Rolle wird wichtiger. Sie muss nicht hauptamtlich besetzt sein, in einem kleinen Team kann die Senior-Developer-Rolle sie mitübernehmen. Aber sie muss ausdrücklich existieren und darf nicht stillschweigend bei jemandem landen, der ohnehin schon alles macht.

## Die KI als Werkzeug

Die KI hat keine Stimme im Daily-Standup. In der Team-Retrospektive fragt sie niemand nach ihrer Meinung. Sie tritt nicht in Stakeholder-Reviews auf. Sie hat keinen Namen am Whiteboard.

Sie hat eine klare Aufgabenliste.

| Die KI darf                                  | Die KI darf NICHT                          |
|----------------------------------------------|--------------------------------------------|
| Code, Tests und Dokumentation schreiben      | Pushen ohne explizite Trigger-Phrase       |
| Issues nach Diktat formulieren               | PRs nach production ohne Trigger erstellen |
| Pläne erstellen und zur Diskussion stellen   | Eigenständig Issues nach Ready ziehen      |
| Refactorings durchführen, sobald freigegeben | Plan-Approval als Ready interpretieren     |

| Die KI darf                                      | Die KI darf NICHT                            |
|--------------------------------------------------|----------------------------------------------|
| Lokale Builds und Test-Suites ausführen          | Tests skippen oder Hooks umgehen             |
| Code-Reviews von zweiten Modellen einlesen       | Eigenmächtig Backlog-Issues priorisieren     |
| Memory-Dateien aktualisieren und konsolidieren   | Verantwortung für Releases tragen            |
| Konventionen aus den Konventionsdateien anwenden | Stakeholder-Kommunikation selbständig führen |

Dass die KI mitgearbeitet hat, schreibe ich in die Commit-Message, als Co-Authored-By-Zeile. Ein Teammitglied wird die KI dadurch nicht. Die Zeile funktioniert wie die Acknowledgement-Sektion in einem wissenschaftlichen Paper: Anerkennung dafür, dass das Werkzeug viel beigetragen hat, ohne Stimme, ohne Verantwortung, ohne Unterschrift. Der Gedanke bekommt inzwischen Rückenwind vom Gesetzgeber: Der EU AI Act verlangt ab August 2026 die Kennzeichnung KI-generierter Inhalte. Commit-Messages meint er nicht. Aber die Richtung ist dieselbe: Transparenz darüber, wo eine Maschine mitgeschrieben hat.

Der Einwand ist berechtigt: Eine Werkzeug-Zeile im Commit wirkt widersprüchlich, wenn das Werkzeug kein Mitglied ist. Ich halte die Zeile trotzdem für richtig, und die Frage gehört nicht weggewischt.

Selbstorganisation, das elfte Prinzip, funktioniert zwischen Menschen. Die KI ist nicht Teil davon. Sie wählt nicht aus, an welchem Issue sie arbeitet, sie definiert nicht die Architektur, sie verhandelt nicht mit Stakeholdern. Die Form verschiebt sich: weniger Stand-ups, mehr asynchrone Koordination über Issue und Board. Die Senior-Developer-Rolle hält dabei die Architektur zusammen, weil ein Modell allein keinen kohärenten Entwurf hinbekommt.

## Die Rolle, über die zu wenig gesprochen wird

Eine Rolle fehlt in dieser Aufzählung, und das wird gerade zum Problem: die der Berufseinsteigerinnen und Berufseinsteiger.

Hosseini und Lichtinger haben in Harvard Lebenslauf- und Stellendaten von 62 Millionen Beschäftigten in 285.000 Unternehmen über zehn Jahre ausgewertet. Ergebnis: In Unternehmen, die generative KI einführen, sinkt die Junior-Beschäftigung innerhalb von sechs Quartalen um sieben bis zwölf Prozent, während die Senior-Beschäftigung praktisch unverändert bleibt. Der Rückgang kommt nicht durch Entlassungen, sondern durch ausbleibende Einstellungen. Eine Stanford-Auswertung von Brynjolfsson, Chandar und Chen auf Basis von Gehaltsabrechnungsdaten zeigt dasselbe Muster zugespitzt auf unsere Branche: Die Beschäftigung in der Softwareentwicklung zwischen 22 und 25 Jahren lag im Juli 2025 knapp zwanzig Prozent unter dem Höchststand von Ende 2022, während sie in allen anderen Altersgruppen weiter wuchs.

Das Problem daran ist betriebswirtschaftlich, nicht sozialpolitisch. Senior-Erfahrung fällt nicht vom Himmel. Sie entsteht, wenn jemand um zwei Uhr nachts ein Produktivsystem wiederbelebt und sich dabei einen Mentalkomplex zu Caching, Locking und Netzwerk-Timeouts aufbaut, der ein Berufsleben lang trägt. Gelernt habe ich immer dann, wenn ich mich wirklich mit einem Problem herumschlagen musste. Wenn die KI diese Phase überspringt, bleibt es bei Oberflächenwissen. Fragen, Antwort bekommen, einbauen, fertig. Gerungen habe ich dabei mit nichts.

Wer die KI bremsen können soll, muss genau die Situationen erleben, die die KI gerade wegrationalisiert. Eine schmale Basis und eine breite Spitze ist keine Pyramide, das ist eine umgedrehte. Sie kippt. Wer heute keine Juniors mehr einstellt, hat in fünf Jahren niemanden, der die Reviews macht.

Das Bild von der KI als Junior ist in der Branche das beliebteste, und es ist das falscheste. Ein Junior lernt aus dem Review: Er macht denselben Fehler kein zweites Mal, weil er verstanden hat, warum er ihn gemacht hat. Das Modell macht ihn beim nächsten Prompt wieder, egal wie ausführlich ich ihn im letzten erklärt habe. Wer das Bild trotzdem benutzt, baut eine Mentoring-Erwartung auf, die niemand einlöst. Genau daher kommt die Frustration, die ich in vielen Teams sehe: Sie erziehen etwas, das nicht erziehbar ist. Ich erziehe nicht. Ich schreibe eine Kontextdatei und pflege sie wie Code.

Meine praktische Antwort auf die Nachwuchsfrage hat zwei Teile. Erstens: Review-first statt Code-first. Wenn die KI ohnehin den ersten Wurf schreibt, sollten Einsteigerinnen und Einsteiger reviewen, nicht schreiben. Das ist anstrengender und schult genau das, was später zählt. Zweitens: Reibung bewusst wieder einbauen. In Diskussionen kursiert dafür der Vorschlag, zwei Stunden pro Woche ohne KI zu arbeiten, damit die einsteigende Person an einer Sache ehrlich verzweifelt. Erprobt habe ich das nicht. Ich halte die Idee für plausibel, belegen kann ich sie nicht.

Im Code-Review stellt sich heute eine andere Frage. Früher: “Warum hast du es so gemacht?” Heute: “Verstehst du, was die KI hier gemacht hat?” Ehrlich beantwortet sie selten jemand. Fremden Code zu reviewen ist die neue Kernkompetenz. Sie baut auf der alten auf. Wer nie selbst geschrieben hat, kann nicht reviewen.

## TEIL II

# Die zwei Werkzeuge verstehen

■ Zwei Kapitel, zwei Werkzeuge, und für beide gilt: Nötig sind sie nicht.

Der Prozess aus Teil I läuft mit GitHub Projects, mit GitLab und in einem vollständig lokalen Modus, in dem die Issues als Dateien im Repo liegen. Ich beschreibe hier die Werkzeuge, mit denen ich selbst arbeite.

Kapitel 5 beschreibt das kanban-kit, ein selbst gehostetes Board mit Projekten, Spalten, Karten, Epics und einem Kennzahlen-Dashboard. Kapitel 6 stellt das claude-workflow-kit vor, eine Bibliothek aus Skills, die den Neun-Schritt-Prozess ausführbar macht, und begründet, warum eine einzelne Workflow-Datei dafür nicht reicht.

Am Ende von Kapitel 6 steht, wie beide zusammenspielen. Es ist unspektakulär, und das ist Absicht.

## KAPITEL 5

# Das kanban-kit: das Board als geteilter Statusträger

Das kanban-kit ist keine Voraussetzung. Der Prozess in diesem Buch läuft gegen GitHub Projects, gegen GitLab und in einem lokalen Modus ohne jedes Board. Das kanban-kit ist eine von vier Möglichkeiten, und ich habe es gebaut, weil ich eine bestimmte Kombination haben wollte. Die anderen Optionen sind darum nicht schlechter. Ganz ohne Statusträger geht es allerdings nicht: Das Ready ist der Moment, in dem ein Mensch ein Issue aus dem Backlog in die Spalte Ready schiebt. Irgendetwas muss diesen Status tragen, ein Board, Labels oder Dateien im Repo. Austauschbar ist das Werkzeug, nicht der Handgriff.

Was ich wollte: ein Board, das mir gehört. Keine Anmeldung bei einem Dienst, keine Frage, wer die Daten sieht, keine Abhängigkeit von einer Preisliste. Dazu ein paar Dinge, die die großen Board-Werkzeuge entweder nicht haben oder hinter Bezahlschranken legen, allen voran die Durchlaufzeiten. Und ich brauchte, gerade am Anfang meiner Arbeit, ein Vorhaben mit echter Komplexität. Wer zeigen will, dass mit KI Software entsteht, die bei Sonar hundert Prozent Testabdeckung, keine Duplikate und keine offenen Befunde hat, muss so etwas wirklich bauen. Sonst kommt verlässlich der Einwand: schöne Theorie, aber zeig mal, dass es geht. Das kanban-kit ist deshalb beides, mein Werkzeug und mein Beleg.

## Die Struktur

Vier Ebenen, mehr nicht.

**Projekt.** Die oberste Klammer. Ein Projekt hat Mitglieder und kann mehrere Boards enthalten.

**Board.** Innerhalb eines Projekts. Ein neues Board bekommt automatisch die fünf Spalten, die der Prozess braucht: Backlog, Ready, In progress, In review, Done. Spalten kann ich anlegen, umbenennen, umsortieren und, wenn sie leer sind, löschen.

**Spalte.** Der Statusträger. Wo eine Karte steht, ist ihr Status, für Menschen wie für das Modell sichtbar.

**Karte.** Das Issue. Titel und Markdown-Beschreibung, dazu Zuständige, Fälligkeit, Labels, Abhängigkeiten auf andere Kartennummern, Anhänge, Kommentare und ein Aktivitätsverlauf.

Der Nacht-Runner liest genau dieses Abhängigkeitsfeld und stellt Karten zurück, deren Vorgänger nicht fertig sind. Kapitel 19 beschreibt das im Detail.

## Was ich im Alltag wirklich benutze

Von den Funktionen benutze ich vier ständig.

**Den Ideen-Pool.** Eine zweite Zone unterhalb der aktiven Karten für alles, was noch nicht auf dem Board sichtbar sein soll. Rohe Anforderungen, halbe Gedanken, Dinge aus einem Stakeholder-Gespräch. Eine Idee erscheint nicht in der Spaltenansicht und belegt keine Position. Per Knopfdruck wandert sie ins Backlog, und per Menüeintrag wandert eine Karte zurück in den Pool. Beide Richtungen brauchen nur das Recht zum Verschieben, nicht zum Löschen. So sieht mein Grooming aus: Ich muss nichts wegwerfen, um mein Board sauber zu halten. In der PO-Schleife aus Kapitel 11 ist der Ideen-Pool die Stufe vor dem Backlog, also die ungesichtete Rohanforderung.

**Labels.** Pro Board definiert, farbig, filterbar in der Listen-Ansicht. Ich benutze sie sparsam, aber an einer Stelle sind sie tragend: Der Nacht-Runner arbeitet standardmäßig nur Karten mit dem Label `kit:nightrun` ab. So markiere ich auf einem einzigen Board, was nachts laufen darf, und behalte den Rest für die interaktive Arbeit am Tag.

**Epics.** Eine Klammer über mehrere Karten, mit Fortschrittsbalken und Board-Filter. Ich benutze sie für alles, was in mehr als fünf Issues zerfällt.

**Das Dashboard.** Das Board misst bei jedem Spaltenwechsel die Verweildauer, unabhängig davon, ob ein Mensch die Karte gezogen hat oder ein Skill sie über die API bewegt. Daraus rechnet es durchschnittliche Lead

Time und Cycle Time aus, die Verweildauer je Spalte, den Durchsatz pro Woche und eine Liste der Ausreißer, also der Karten, die über sieben Tage in einer Spalte lagen.

Diese Ausreißer-Liste ist die nützlichste Ansicht im ganzen Werkzeug. Sie beantwortet die Frage, die ich mir sonst nicht stelle: Wo bleibt die Arbeit eigentlich liegen? Bei mir liegt sie fast nie in In progress. Sie liegt in In review, und das ist genau der erwartete Befund, wenn der Takt vom langsamsten menschlichen Schritt bestimmt wird.

## Rechte, Papierkorb, Editiermodus

Drei Dinge, die im Alltag kaum auffallen und deshalb richtig gebaut sein müssen.

Rollen gibt es auf zwei Ebenen. Plattformweit `USER` und `ADMIN`, pro Projekt `OWNER`, `ADMIN`, `MEMBER` und `VIEWER`. Den letzten `OWNER` eines Projekts kann niemand entfernen oder herabstufen. Das ist eine kleine, aber wichtige Sicherung. Weitere Rollen mit individuell zugeschnittenen Rechten habe ich mir schon überlegt. Gebraucht habe ich sie bisher nicht, also habe ich sie nicht gebaut. YAGNI gilt auch für das eigene Werkzeug.

Gelöschte Karten landen im Papierkorb, und das Board entfernt sie erst nach einer Aufbewahrungsfrist endgültig. Karten in Done bekommen einen Countdown und wandern danach ins Archiv.

Der Editiermodus trennt den Alltag von strukturellen Änderungen. Standardmäßig ist er aus, und nach einem Neustart ist er es wieder. Erst wenn er an ist, erscheinen die Symbole zum Umbenennen von Projekten, Boards und Spalten. Karten anlegen, verschieben und archivieren funktioniert unabhängig davon. Ich habe das eingebaut, nachdem ich mir zweimal beim Arbeiten eine Spalte umbenannt hatte.

## Eine Unstimmigkeit, die auffallen wird

Wer in die Konfiguration des `claude-workflow-kit` schaut, findet für dieses Board nicht den Wert `kanban-kit`, sondern `toolbox`, und der zu-

gehörige Adapter heißt `tbx.mjs`. Das ist ein historischer Name: Das Board war ursprünglich Teil meiner Anwendung Toolbox, bevor ich es als eigenes Werkzeug herausgelöst habe. Die Toolbox gibt es nach wie vor, und ich sehe bis heute keinen Grund, die übrigen Namen umzubenennen.

## KAPITEL 6

# Das claude-workflow-kit: der Prozess, ausführbar gemacht

Mein Workflow stand in einer Datei. Das Modell hat ihn trotzdem vergessen.

Am Anfang jeder Session lief alles nach Plan, aber zwei Stunden später fehlten auf einmal Schritte. Ich saß da und dachte: Das habe ich doch gerade erst gesagt. Also wiederholte ich mich, immer und immer wieder.

Der Grund ist technisch, und ich gucke genau hin, weil davon abhängt, welche Vorkehrungen ich treffe. Ein Sprachmodell erinnert sich nicht, es verarbeitet ein Kontextfenster. Innerhalb einer Session vergisst es nichts, aber bei Nachricht hundert steht Nachricht eins gegen neunundneunzig neuere. Die Architektur bevorzugt das zuletzt Gesagte. Das ist kein Bug, so ist das Modell gebaut. Eine Workflow-Datei, die das Modell einmal am Anfang liest, verliert dadurch an Wirkung. Sie wird nicht gelöscht. Sie wird begraben.

Dazu kommt die Kompaktierung. Aus “Schritt 4: Vor der Implementierung Testfälle definieren, weil sie die Akzeptanzkriterien absichern” wird “Tests schreiben”. Die Regel ist noch da, aber verschwommen.

Wer glaubt, das Modell habe vergessen, wiederholt sich. Wer weiß, dass es umgewichtet hat, räumt den Kontext auf. Die praktische Folge sieht in beiden Fällen ähnlich aus. Was ich daraus mache, ist verschieden.

## Die Idee: eine Anweisung pro Schritt, im richtigen Moment

Das Kit ist meine Antwort darauf. Wenn es nicht reicht, am Sessionanfang sehr ausführlich alle Schritte zu beschreiben, dann zerlege ich sie in kleine Anweisungen und lade jede genau dann, wenn ihr Schritt dran ist.

Ein Skill muss keine zwei Stunden Konversation überleben. Er muss keine Kompaktierung überstehen. Jeder Schritt bekommt seine Anweisung

just in time. Jemand, der sich an eine Einarbeitung von vor drei Monaten erinnern soll, gegen jemanden, der vor jedem Arbeitsschritt die passende Checkliste aufschlägt. Der zweite produziert weniger Fehler. Klüger ist er nicht, das System stützt ihn.

Ein zweiter Punkt: Ich rufe die Skills selbst auf. Wenn der Plan steht und die Issues drankommen, tippe ich `/issues`. Kein Raten und kein Hoffen, dass die Maschine mich versteht. Wer sich darauf verlässt, dass das Modell schon von selbst den richtigen Skill wählt, spielt Lotterie. Der Skill läuft, weil ich es entschieden habe.

Skills lösen das Problem, das ich lange falsch beschrieben habe. Sie schaffen keine Automatik, sie schaffen Struktur. Sie verlagern Anweisungen dorthin, wo Drift sie nicht erreicht: ins Dateisystem, abrufbar im richtigen Moment.

## Die Skills im Überblick

Der Kernprozess hat neun Schritte. Das Modell übernimmt davon fünf, vier bleiben menschlich. Dazu kommen Skills, die außerhalb der Nummerierung stehen und den Arbeitsrhythmus strukturieren.

| Schritt | Was                                 | Wer    | Skill                         |
|---------|-------------------------------------|--------|-------------------------------|
| 1       | Anforderung formulieren             | Mensch | kein Skill                    |
| —       | Fachliches Issue anlegen (optional) | Modell | <code>/fachplan</code>        |
| 2       | Anforderung planen                  | Modell | <code>/plan</code>            |
| 3       | Issues anlegen                      | Modell | <code>/issues</code>          |
| 4       | Ready: Issues freigeben             | Mensch | kein Skill                    |
| 5       | Ready-Issues implementieren         | Modell | <code>/implement-ready</code> |
| 6       | Lokale Checks ausführen             | Modell | <code>/local-check</code>     |
| 7       | Review durchführen                  | Modell | <code>/review</code>          |

| Schritt | Was                   | Wer    | Skill             |
|---------|-----------------------|--------|-------------------|
| 8       | Push auf main         | Mensch | /push-main        |
| 9       | Merge nach production | Mensch | /merge-production |

Zwischen Schritt 8 und 9 prüfe ich den Test-Server im Browser. Dafür gibt es keinen Skill, Pflicht ist es trotzdem.

Quer dazu stehen einige weitere Skills. `/kontext` lädt zum Session-Start den Lageüberblick, `/document` schreibt zum Session-Ende den Log, `/retro` ist der Wartungsschritt für den Prozess selbst. `/issue-review` prüft die Dokumente selbst, bevor das Modell gegen sie implementiert; Kapitel 9 beschreibt das ausführlich. `/implement-test` und `/implement-done` sind der kleinteilige Einstieg in Schritt 5 mit sichtbarem Rot-Grün-Übergang, `/implement-next` nimmt genau ein Issue und ist zugleich der Baustein des Nachtbetriebs.

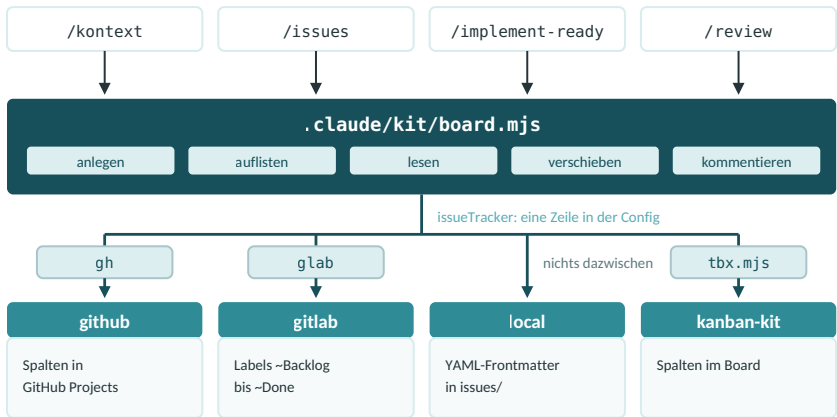
Kapitel 16 beschreibt jeden Skill einzeln mit Zweck, Eingabe, Ausgabe und Grenzen.

## Der Board-Adapter

Die Skills wissen nichts von `gh` oder `glab`. Alle Issue- und Board-Operationen laufen über einen Adapter, `.claude/kit/board.mjs`. Er kennt eine kleine Schnittstelle, im Kern: Issue anlegen, auflisten, lesen, verschieben, kommentieren. Anhang A listet sie vollständig.

Der Adapter ist selbst noch einmal geschichtet. Für GitHub und GitLab spricht `board.mjs` die jeweilige CLI an, im lokalen Modus schreibt er Dateien. Wer das kanban-kit als Board benutzt, braucht zusätzlich `tbx.mjs`. Diese Datei ist der Adapter von `board.mjs` zum kanban-kit und übersetzt die Schnittstelle auf dessen API. Für die Skills ändert sich dadurch nichts, sie rufen weiterhin nur `board.mjs` auf.

Die Skills wissen nichts von gh oder glab. Sie rufen immer nur den Adapter.



Die mittlere Reihe ist das Werkzeug, das der Adapter bedient: zwei CLIs, ein zweiter Adapter, im lokalen Modus keines. `codeHost` ist die zweite, unabhängige Achse. Nützlichste Kombination: `codeHost github`, `issueTracker local`.

Die Frage in der Softwareentwicklung ist ja immer: Wann baue ich eine Abstraktion? Als ich mit GitHub gearbeitet habe, wusste ich schon, dass GitLab irgendwann dazukommt, aber ich habe eine bewusst unsaubere Lösung in Kauf genommen. Die Fallunterscheidung stand im Skill. Das war architektonisch falsch. Ich hatte die technische Schuld notiert. Erst als ich wirklich mit der GitLab-Anbindung begann, brauchte ich die Abstraktionsschicht.

Ich konfiguriere über zwei unabhängige Achsen. `codeHost` bestimmt, wo Repository und Pull Requests liegen. `issueTracker` bestimmt, wo Issues verwaltet und Board-Karten bewegt werden. Beide dürfen auf verschiedene Plattformen zeigen, etwa Code auf GitHub und Issues lokal.

| Ziel                | Voraussetzung                                                                            |
|---------------------|------------------------------------------------------------------------------------------|
| <code>github</code> | GitHub CLI ( <code>gh</code> ), authentifiziert, plus ein Projekt-Board mit fünf Spalten |
| <code>gitlab</code> | GitLab CLI ( <code>glab</code> ), authentifiziert, Spalten werden über Labels abgebildet |

| Ziel                 | Voraussetzung                                                           |
|----------------------|-------------------------------------------------------------------------|
| <code>local</code>   | Nichts. Issues liegen als Markdown-Dateien mit YAML-Frontmatter im Repo |
| kanban-kit als Board | Laufende Instanz, ein projekt-gebundenes Token und <code>tbx.mjs</code> |

Im lokalen Modus gibt es kein Board, keine externe CLI, keine Anmeldung. Die Issues liegen als Dateien im Repository, sind damit versionierbar und laufen durch dieselben Reviews wie der Code. Für Einzelarbeit und für den ersten Versuch mit dem Prozess ist das der geradlinigste Weg.

## Warum Jira fehlt

Das Kit unterstützt Jira nicht. Die Frage kommt in jedem Gespräch, deshalb steht die Antwort hier und nicht im Anhang.

Das liegt nicht an Jira. Ein Adapter dafür wäre möglich, die Schnittstelle ist schmal, und Jira hat alles, was der Prozess braucht. Ich habe es bisher nicht gebraucht. Keiner meiner Kunden und keines meiner Projekte hat es verlangt.

Das ist YAGNI, angewandt auf mein eigenes Werkzeug. Ich habe die Abstraktionsschicht gebaut, als der zweite Fall auftrat, nicht als ich ihn für denkbar hielt. Denselben Maßstab lege ich an den dritten an. Wenn ein Projekt Jira verlangt, entsteht der Adapter an dem Tag, an dem ich ihn brauche, und vermutlich an einem Nachmittag. Bis dahin wäre er Code, den niemand aufruft.

Wer heute mit Jira arbeitet und den Prozess trotzdem fahren will, hat zwei Wege. Der eine ist, den Adapter zu schreiben, die Schnittstelle ist dokumentiert und klein. Der andere ist, die Achsen zu trennen: Jira bleibt das Werkzeug, in dem das Management den Stand sieht, und der Prozess läuft darunter im lokalen Modus. Der zweite Weg klingt nach Kompromiss. In der Praxis ist er oft ehrlicher, weil ein Management-Jira

und ein Board für die tägliche Entwicklung selten gleich fein geschnitten sind.

## **Wie beide Werkzeuge zusammenspielen**

Das Zusammenspiel ist unspektakulär. Die Skills schreiben und bewegen Issues über den Adapter auf dem Board. Der Mensch schiebt genau an drei Stellen etwas.

Das Board ist dabei kein Kontrollinstrument für das Modell. Es ist ein gemeinsames Gedächtnis. Das Modell liest von dort, was zu tun ist, und schreibt dorthin, was es getan hat. Ich lese dasselbe. Weil der Status an einem Ort steht und nicht in einem Chatverlauf, überlebt er den Sessionwechsel, den Rechnerwechsel und die Nacht.

## TEIL III

# Der Arbeitsfluss in der Praxis (fachlich)

Hier steht die tägliche Arbeit. Board und Skills laufen in diesem Teil nebeneinander, so wie sie es im Alltag auch tun.

Kapitel 7 beschreibt das Issue: Es ist die Quelle der Wahrheit, nicht der Chat. Wenn aus diesem Buch eine einzige Regel bleibt, dann diese. Kapitel 8 ist die Landkarte mit den neun Schritten, jeder mit einem eindeutigen Akteur und einem klaren Ergebnis. Kapitel 9 lässt das fertige Issue von zwei Modellen gegenlesen, bevor ich es freigebe. Kapitel 10 zeigt einen kompletten Tag von halb neun bis Feierabend. Er fängt mit der Ernte an und hört mit der Saat auf.

Danach drei Kapitel, die die Kommunikation bestimmen:

- die Zusammenarbeit mit einem Product Owner
- die Events, die ein Team braucht
- die Frage, wie lang eine Iteration eigentlich sein sollte

Wer nur ein Kapitel aus diesem Buch liest, sollte Kapitel 10 nehmen.

## KAPITEL 7

# Das Issue als Single Source of Truth

Das Issue ist die Quelle der Wahrheit. Nicht der Chat, nicht der Plan, nicht mein Gedächtnis. Wenn ich aus diesem Buch eine einzige Regel retten müsste, wäre es diese.

Der Grund ist rein technisch. Ein Sprachmodell beginnt jede Session bei null, und innerhalb einer Session verliert früh Gesagtes an Gewicht, ohne dass ich es merke. Das Modell meldet nicht, wenn ihm etwas entgleitet. Was im Issue steht, liest das Modell zu Beginn jeder Bearbeitung frisch.

Ich habe die Erinnerung aus dem Modell herausgezogen und in Dateien gelegt. Alles andere in diesem Kapitel folgt daraus.

## Die vier Abschnitte

Ein Issue in meinem Format hat vier Abschnitte, und es passt auf eine Seite.

**Kontext.** Warum diese Aufgabe ansteht, was vorher fehlte, welche Vorgeschichte dazugehört. Hier stehen auch die Rückverweise auf das Plandokument und auf ein fachliches Issue, falls die PO-Schleife aus Kapitel 11 im Spiel ist.

Dazu kommt beim Anlegen eine Zeile `Autor-Modell:`. Sie hält fest, welches Modell das Issue geschrieben hat, und `/issues` füllt sie aus der Umgebungsvariablen `KIT_AGENT_MODEL`. In einer interaktiven Session steht dort `unbekannt`. `/issue-review` braucht die Zeile später, damit kein Modell sein eigenes Issue prüft.

Im selben Abschnitt können `Pruefung:` und `Pruefung-Stand:` stehen. Damit lege ich fest, wie viele Prüfrunden dieses Issue braucht, oder dass es keine braucht. Kapitel 16 beschreibt das.

**Aufgabe.** Was konkret zu tun ist, welche Dateien betroffen sind, welche Tests vorher geschrieben werden müssen.

**Akzeptanzkriterium.** Wie verifiziert wird, dass die Aufgabe erledigt ist. Konkret, messbar oder ausführbar. An diesem Abschnitt entscheidet sich, ob ein Issue etwas taugt.

**Abhängigkeiten.** Welche anderen Issues zuerst fertig sein müssen, als Nummern, oder “Keine”.

Dazu kommen zwei Angaben, die ich mir angewöhnt habe: eine grobe Aufwandsschätzung und ein ausdrückliches Out-of-Scope. Im Out-of-Scope sage ich, was in diesem Issue **nicht** passiert. Ohne diese Zeile fängt das Modell an, selber zu denken, eigenständig zu formulieren und den Scope zu erweitern, weil es das konsistenter findet und jede einzelne Erweiterung für sich plausibel ist. Je enger ich die Grenzen ziehe, in denen sich das Modell bewegen darf, desto besser wird das Ergebnis.

## Warum die Abhängigkeiten Nummern sind

Wozu der Aufwand gut ist, zeigt sich spätestens beim ersten Nachlauf. Der Nacht-Runner liest genau dieses Feld und stellt Issues zurück, deren Vorgänger noch nicht fertig sind. Fehlt der Eintrag, kann er nicht entscheiden, ob das Fundament noch trägt, und baut auf Sand weiter.

Eine Regel dazu habe ich auf die harte Tour gelernt: Der Rückverweis auf ein fachliches Issue gehört in den Kontext, nie in die Abhängigkeiten. Ein fachliches Issue wird erst fertig, wenn seine technischen Kinder fertig sind. Steht es als Abhängigkeit in den Kindern, wartet jedes Kind auf den Elternteil, der auf die Kinder wartet. Eine Henne-Ei-Falle, die einen ganzen Nachlauf kostet und morgens wie ein Fehler des Runners aussieht.

## Kleinteilig heißt: eigenständig testbar

Wer mich kennt, weiß, dass ich schon immer ein Freund davon war, kleinteilig zu schneiden, und das gilt für die KI genauso. Ein Issue ist ein logischer Schritt, der für sich getestet werden kann. Je kleiner ich schneide, desto genauer ist das Ergebnis.

Wenn ich beim Lesen denke, das dauert drei Tage, ist es kein Issue, sondern ein Epic, und ich schneide weiter. Ein Time-Series-Feature habe ich so in acht Issues zerlegt, jedes mit eigenen Akzeptanzkriterien und expliziten Abhängigkeiten. Kleinteilige Issues sind leichter umzusetzen, leichter zu reviewen und im Fehlerfall leichter zurückzurollen.

Vor einem Nachtlaf verschärfe ich das Schnittkriterium noch einmal. Tagsüber kann ich eine Unklarheit im Gespräch auflösen. Nachts wird sie geraten. Ein Issue, das nachts laufen soll, muss ohne Rückfrage umsetzbar sein.

Ein Modell erzeugt zu jeder Unklarheit sofort eine plausible Variante. Es fragt nicht nach. Wenn etwas offen ist, trifft es eine Annahme und formuliert sie als Tatsache.

## Nichts auf Vorrat

Kleinteilig schneiden heißt auch, nicht auf Vorrat zu bauen. Ich hatte einmal einen Chef, der von “You Ain’t Gonna Need It” überhaupt nichts hielt. Sein Argument: Eine Funktion soll nur einmal angefasst werden, also beim ersten Mal vollständig ausspezifiziert und vollständig implementiert. Das Ergebnis waren vollständige CRUD-Implementierungen, bei denen niemand fragte, ob das Löschen jemals jemand aufruft. Code, der existierte, ohne je aufgerufen zu werden, und der trotzdem getestet, gewartet und mitgeschleppt werden musste.

Das Gegenargument gegen YAGNI war immer dasselbe: Nacharbeiten kostet. Das stimmte, solange Implementierung teuer war. Heute stimmt es nicht mehr. Ein Beispiel aus einer Vereinswebseite: Der News-Bereich erzeugt Meldungen automatisch aus dem Kontext, das war Anforderung eins. Später wollte der Kunde News anpassen können. Ein Issue, fünf Minuten Implementierung, deployed. Geplant war ein Ringpuffer mit zehn Einträgen ohne explizites Löschen, weil das niemand brauchte. Im Betrieb zeigte sich, dass manuelles Löschen sinnvoll wäre. Wieder ein Issue, wieder fünf Minuten, wieder deployed. Drei Schritte, drei Implementierungen, keine davon verschwendet.

Mit der KI kostet das Hinzufügen fast nichts mehr. Ein zusätzliches Feature, eine zusätzliche Option, eine zusätzliche Validierung: alles nur ein Issue. Genau das macht Einfachheit schwerer, denn früher hat der Aufwand von selbst gebremst, wer alles einbauen wollte. Diese Bremse ist weg. Billig geworden ist nur die Implementierung. Ungenutzten Code muss ich weiterhin ausspezifizieren, testen und warten.

## **Der Chat ist kein Speicher**

Der typische Verstoß gegen diese Regel sieht harmlos aus. Mir fällt beim Reviewen etwas auf, ich sage es im Chat, das Modell setzt es um, alles gut. In derselben Session funktioniert das. Zwei Tage später existiert diese Absprache nicht mehr, und niemand weiß, warum der Code so aussieht.

Deshalb: Fällt mir etwas auf, ändere ich das Issue. Kommt eine späte Anforderungsänderung, wandert sie ins Issue, nicht in den Chatverlauf. Ergänze ich ein Akzeptanzkriterium, steht es danach im Issue. Späte Änderungen sind willkommen, das ist das zweite agile Prinzip. Späte Änderungen, die im Chat liegen bleiben, sind im KI-Modus so gefährlich wie eh und je.

Dieselbe Regel gilt, wenn ich mit einem Product Owner verhandle. Das läuft im Body des Issues, nicht in Kommentaren darunter. Sonst steht die Anforderung an zwei Orten, ein Teil im Body, ein Teil darunter, und eine spätere Session muss sie sich zusammensuchen. Der Body trägt den verhandelten Stand, die Kommentare den Verlauf dorthin. Was nicht im Body steht, existiert für die Implementierung nicht.

## **Wer das Issue schreibt**

Das Modell schreibt Issues, ich diktiere sie. Diese Arbeitsteilung ist produktiv, weil das Modell gut darin ist, aus “der Abstand hier wirkt komisch” die Frage zu machen, welcher Abstand, im Verhältnis zu welchem Element, mit welcher CSS-Eigenschaft, in welchem Breakpoint. Das ist Übersetzungsarbeit.

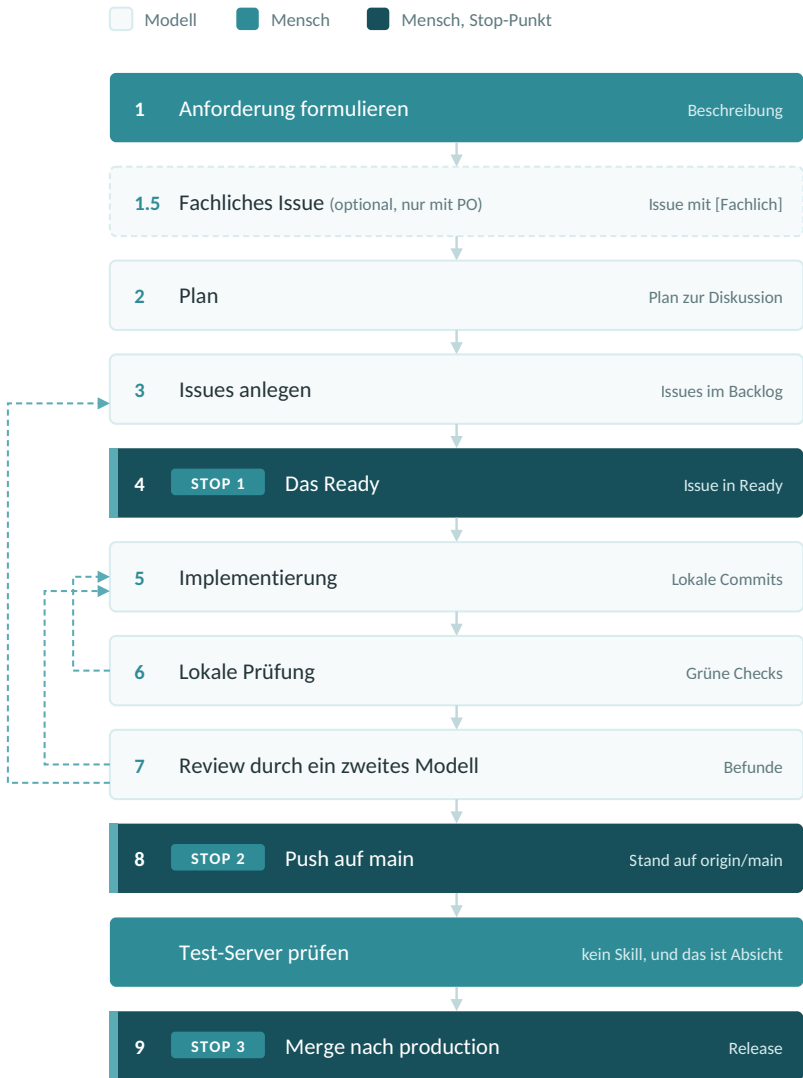
Nach Ready bewegt das Modell kein Issue. Ein Issue zu formulieren und ein Issue freizugeben sind zwei verschiedene Handlungen, und nur die zweite ist eine Entscheidung.

## KAPITEL 8

# Der Neun-Schritt-Prozess im Überblick

Was ich in Teil I als Werte, Prinzipien und Rollen beschrieben habe, schreibe ich hier als Anleitung auf. Neun Schritte, jeder mit einem klaren Akteur und einem klaren Output. Dieses Kapitel ist die Landkarte. Kapitel 10 zeigt dieselben Schritte an einem echten Tag, Kapitel 16 beschreibt die zugehörigen Skills im Einzelnen.

Fünf Schritte macht das Modell. Vier bleiben beim Menschen, und drei davon sind die Stop-Punkte, an denen der ganze Prozess hängt.



Gestrichelt: die regulären Rückwege. Aus 6 und 7 zurück auf 5, aus 7 auf 3, wenn ein Befund keine Korrektur ist, sondern eine Designfrage.

Fünf Schritte macht das Modell. Vier bleiben beim Menschen, drei davon sind Stop-Punkte.

## 01 Anforderung formulieren

*Mensch.* Die auftraggebende Person formuliert oder diktiert eine Anforderung. Wenn ein Satz abbricht oder mehrdeutig bleibt, wird nachgefragt, nicht geraten.

*Output:* eine knappe, verständliche Beschreibung dessen, was gebaut werden soll.

## Ohne Nummer: Das fachliche Issue

*Modell.* Nur in Projekten mit Product Owner. Das Modell schreibt die rohe Anforderung in ein fachliches Issue um, technikfrei, im Story-Format. Kapitel 11 beschreibt diese Schleife.

*Output:* ein Issue mit `[Fachlich]`-Präfix, das die PO-Rolle abnehmen kann.

## 02 Plan

*Modell.* Der Plan benennt Ziel und Wirkung, betroffene Bereiche und Dateien, architektonische Entscheidungen mit Begründung, offene Fragen und die geplante Verifizierung. Annahmen stehen ausdrücklich drin. Der Plan geht in die Diskussion, nicht in die Implementierung.

Diesen Schritt halte ich für den meistunterschätzten im ganzen Prozess. Hier wandert die Denkarbeit nach vorn, und was hier offenbleibt, klärt sich später im Code, wo es das Zehnfache kostet.

*Output:* ein lesbarer Plan, an dem ein Mensch entscheiden kann, ob er sinnvoll ist.

## 03 Issues anlegen

*Modell.* Der freigegebene Plan wird in kleinteilige Issues geschnitten, jedes mit den vier Abschnitten aus Kapitel 7. Ab jetzt ist das Issue die Quelle der Wahrheit.

*Output:* ein oder mehrere Issues im Backlog.

## 04 Das Ready

*Mensch. Erster Stop-Punkt.* Die PO-Rolle schiebt die Issues für den aktuellen Durchlauf nach Ready. Darin liegt die Planung: wie viel Arbeit auf einmal, in welcher Reihenfolge, mit welchen Abhängigkeiten.

Ein Plan-Approval aus Schritt 2 ist nicht das Ready. Diese Verwechslung sehe ich häufiger als jede andere, und sie ist der Grund, warum die beiden Schritte getrennt sind.

*Output:* ein freigegebenes Issue in Ready.

## 05 Implementierung

*Modell.* Implementiert wird gegen das Issue, nicht gegen den Chat. Pro Issue: Karte nach In progress, Issue vollständig lesen, Tests zuerst schreiben, dann implementieren bis grün, lokal committen, Karte nach In review.

Test-First ist hier keine Stilfrage. Wer sagt “schreib mir Funktion X”, bekommt eine plausibel aussehende Funktion X, die in den meisten Fällen ihren Job tut. Wer sagt “schreib einen Test, der das Verhalten X spezifiziert, und dann eine Funktion, die diesen Test grün macht”, bekommt eine Funktion mit ausdrücklichem Vertrag.

*Output:* lokale Commits mit Bezug auf das Issue. Nichts ist gepusht.

## 06 Lokale Prüfung

*Modell.* Alle Pflicht-Checks laufen: Build, Tests, Coverage, Mutation Score, statische Analyse, Architekturregeln. Bei Frontend-Änderungen kommt die Verifikation im Browser dazu. Die neueren Modelle können den Browser selbst bedienen, urteilen muss darüber weiterhin ein Mensch.

Ein roter Check blockiert den weiteren Prozess. Ohne Ausnahme.

*Output:* eine grüne Pflicht-Check-Liste oder ein Stopp.

## 07 Review durch ein zweites Modell

*Modell.* Ein Review in frischer Session, ohne den Entstehungskontext, möglichst durch ein Modell anderer Bauart. Ein Modell ist gegenüber seiner eigenen Lösung systematisch unkritisch. Die Befunde landen im Issue oder im Pull Request.

Das Review deckt nicht alles ab. Secrets-Scan, SQL-Konkatenation und fehlende Validierung gehören in deterministische Werkzeuge im CI, weil ein solches Werkzeug mit keinem Sprachmodell einen blinden Fleck teilt.

*Output:* ein dokumentierter zweiter Blick.

## 08 Push

*Mensch.* Zweiter Stop-Punkt. Auf explizite Trigger-Phrase wird der aktuelle Commit-Batch auf main gepusht, nicht mehr. Eine frühere Freigabe in derselben Session gilt nicht für neue Commits.

*Output:* ein neuer Stand auf origin/main.

## Zwischenschritt: Test-Server prüfen

*Mensch.* Der Test-Server zieht main. Geprüft wird im Browser: Golden Path, kritische Randfälle, keine sichtbaren Regressionen. Dafür gibt es keinen Skill, und das ist Absicht.

## 09 Merge nach production

*Mensch.* Dritter Stop-Punkt. Ein Pull Request von main nach production wird erstellt. Den Merge führt die Person durch, die auf dem Test-Server geprüft hat.

Mit dem Merge wandern die betroffenen Karten auf Done. Auch das macht ein Mensch, kein Skill.

*Output:* ein Release in Produktion.

## Die Zählung, und warum sie starr ist

| Schritt                       | Akteur        | Ergebnis              |
|-------------------------------|---------------|-----------------------|
| 1 Anforderung                 | Mensch        | Beschreibung          |
| — Fachliches Issue (optional) | Modell        | [Fachlich] -Issue     |
| 2 Plan                        | Modell        | Plan zur Diskussion   |
| 3 Issues                      | Modell        | Issues im Backlog     |
| <b>4 Ready</b>                | <b>Mensch</b> | <b>Issue in Ready</b> |
| 5 Implementierung             | Modell        | Lokale Commits        |
| 6 Lokale Prüfung              | Modell        | Grüne Checks          |
| 7 Review                      | Modell        | Befunde               |
| <b>8 Push</b>                 | <b>Mensch</b> | <b>Stand auf main</b> |
| <b>9 Merge</b>                | <b>Mensch</b> | <b>Release</b>        |

Die Nummerierung ist pedantisch. Sie steht so in den Skills, in der Workflow-Datei und in diesem Buch, weil eine gemeinsame Zählung eine gemeinsame Sprache ergibt. “Ich bin zurück auf fünf” ist eine vollständige Statusmeldung.

Zwischennummern gibt es deshalb nicht. Eine Nummer würde eine Reihenfolge und eine Pflicht behaupten, die es beim fachlichen Issue nicht gibt.

Aus demselben Grund trägt auch die Prüfung der Dokumente keine Nummer. Zwischen Schritt 3 und dem Ready prüfe ich, ob das Issue selbst taugt, bevor jemand dagegen implementiert. Das ist optional und lohnt sich nicht bei jedem Einzeiler. Kapitel 16 beschreibt den Skill dazu, `/issue-review`.

Der Weg durch die Schritte ist nicht immer geradeaus. Aus Schritt 6 und 7 geht es regelmäßig zurück auf 5. Aus Schritt 7 wird gelegentlich ein neues Issue in Schritt 3, wenn ein Befund keine Korrektur ist, sondern

eine Designfrage. Einen Stop-Punkt überspringe ich dabei nie, auch dann nicht, wenn das Ergebnis ohnehin gut aussieht.

## **Wann der volle Prozess nicht nötig ist**

Neun Schritte für eine Textkorrektur wären albern, und wer das Verfahren so liest, legt es zu Recht weg. Es gibt zwei Bahnen.

Bahn 1 ist die kleine Änderung. Eine Datei, ein Asset oder ein Config-Wert, und sie bleibt im Haus: kein Datenmodell, keine Migration, kein Endpoint, keine Sicherheitslogik, nicht mehr als ein Modul. Ein Favicon tauschen. Einen Tippfehler beheben. Einen Default in der Config ändern. Das setze ich direkt um, ein Commit, fertig. Kein Plan, kein Issue, kein Ready, kein Plandokument.

Bahn 2 ist alles andere. Sobald Datenmodell, API, Migration, Sicherheit oder ein zweites Modul im Spiel sind, läuft der volle Weg aus diesem Kapitel. Das gilt auch, wenn der Aufwand über einen Commit hinausgeht. Eine neue Tabelle, ein neuer Endpoint, ein neues Feature in der Oberfläche.

Trifft eines der Bahn-1-Merkmale nicht zu, ist es Bahn 2. Im Zweifel auch. Der Fehler in diese Richtung kostet einen Plan, den ich nicht gebraucht hätte. Der Fehler in die andere Richtung kostet einen Nachmittag.

Ein Stop-Punkt fällt auch in Bahn 1 nicht weg: Gepusht wird nur auf Trigger-Phrase. Die kleine Änderung spart den Plan, nicht die Verantwortung.

Meistens sage ich selbst, welche Bahn es ist. Sage ich nichts, benennt das Modell sie zu Beginn laut, "das ist Bahn 1, ich ändere direkt". Es wartet dabei nicht auf Antwort. Ich kann eingreifen, aber nur, wenn ich dabei bin und mitlese. Sichtbarkeit also, mehr nicht. Bei einem Favicon-Tausch auf ein Ready zu warten wäre genau die Zeremonie, gegen die Bahn 1 gebaut ist.

## KAPITEL 9

## Drei Modelle für ein Issue

Ich schreibe ein Issue. Es ist gut. Sauber geschnitten, die Akzeptanzkriterien sind konkret, die Abhängigkeiten stehen drin. Dann gebe ich es einem anderen Modell zu lesen, und das findet Fehler.

Das passiert nicht gelegentlich, das passiert verlässlich.

Das war zuerst ärgerlich, weil ich glaubte, fertig zu sein. Dann wurde es interessant, weil es sich wiederholte. Was sich wiederholt, liegt am Verfahren.

### Warum ich die Lücke nicht sehe

Wenn ich ein Issue schreibe, habe ich den Kontext im Kopf, aus dem es entstanden ist. Das Gespräch, das vorausging. Die Datei, die dabei offen war. Die Entscheidung von vorletzter Woche. Nichts davon steht im Issue, und das muss es auch nicht, es steht ja in meinem Kopf.

Beim Wiederlesen ergänze ich diesen Kontext, ohne es zu merken. Ich lese nicht den Text. Ich lese, was ich gemeint habe. Solange derselbe Kopf liest, bleibt die Lücke zwischen beidem unsichtbar.

Ein fremdes Modell hat diesen Kopf nicht. Es hat den Text. Es stolpert genau dort, wo später die Implementierung stolpern wird.

Das ist derselbe Grund, aus dem Code-Review funktioniert, eine Stufe früher und mit höherem Einsatz. Ein Fehler im Code kostet den Code. Ein Fehler im Issue pflanzt sich in alles fort, was daraus entsteht, und das Issue ist nach Kapitel 7 die Quelle der Wahrheit.

### Ein Beleg, der mir nicht gefällt

Am 6. August 2026 habe ich siebzehn Issues geschrieben, formuliert vom stärksten Modell, das ich zur Verfügung hatte, im Vier-Abschnitt-Format, mit sorgfältig ausformulierten Akzeptanzkriterien.

Drei davon trugen ein Kriterium der Form “Manueller Durchlauf gegen ein Wegwerf-Verzeichnis”.

Nachts lief der Runner. Er startet Sessions ohne Menschen daneben. Ein Kriterium mit dem Wort manuell kann er nie abhaken, weil niemand da ist, der es tut.

Die Sessions standen vor einer Wahl, die sie nicht gewinnen konnten. Eine hielt an und meldete, sie könne nicht abschließen; für den Runner sah das aus wie ein Fehlschlag. Die zweite schloss ab und vermerkte das offene Kriterium im Bericht; für das Board sah das aus wie ein Erfolg. Sie stand wochenlang auf Done, obwohl sie nicht fertig war.

Beides an einem Abend, beides aus derselben Ursache: ein Wort im Issue, das ich für harmlos gehalten hatte.

Das Modell war stark. Wer den Kontext hat, sieht die Lücke trotzdem nicht.

Drei Stufen, fünf Aufträge

`/issue-review` sitzt zwischen `/issues` und dem Ready. Modelle, die das Dokument nicht geschrieben haben, lesen es und sagen, wo es schärfer werden muss.

Das fertige Arbeitspaket ist dabei nur die unterste Stufe. Dieselbe Überlegung gilt eine Stufe höher und noch eine:

| Stufe                 | Prüft                                                | Nachweis im Issue |
|-----------------------|------------------------------------------------------|-------------------|
| <code>fachlich</code> | das [Fachlich]-Issue aus <code>/fachplan</code>      | Fachplan-Review:  |
| <code>plan</code>     | das Plandokument aus <code>/plan</code>              | Plan-Review:      |
| <code>issue</code>    | das technische Arbeitspaket aus <code>/issues</code> | Issue-Review:     |

Der Aufruf ist immer derselbe: `/issue-review #N`. Es gibt bewusst kein `/fachplan-review` und kein `/plan-review`. Welche Stufe greift, liest der Skill am Titel-Präfix ab. Drei Kommandos wären drei Wege, die Stufe

falsch zu wählen. Das Dokument weiß selbst, was es ist. Ohne Argument nimmt der Skill das ganze Backlog.

Die Prüfung wandert nach oben, weil die Reichweite eines Fehlers nach unten wächst. Ein Fehler in der fachlichen Anforderung pflanzt sich in den Plan fort, von dort in jedes Arbeitspaket und schließlich in allen Code. Derselbe Fehler, im Arbeitspaket gefunden, kostet ein Issue. In der Anforderung gefunden, kostet er einen Satz.

Ein Autor und ein Prüfer sind besser als ein Autor allein. Aber ein Prüfer sieht eine Sorte Fehler: die, nach der er sucht. Deshalb hat jede Stufe eigene Rollen:

| Stufe    | Rolle                      | Fragt                                                                                                                                      |
|----------|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| fachlich | Form und Beobachtbarkeit   | Trägt die Anforderung alle vier Story-Abschnitte? Ist jedes Kriterium aus Nutzersicht beobachtbar? Steht Technik drin, wo keine hingehört? |
| fachlich | Abgrenzung                 | Widersprechen sich Ziele und Nicht-Ziele? Fehlt eine Scope-Grenze? Ist eine offene Frage längst entschieden?                               |
| plan     | Architektur und Bestand    | Stimmt jede Behauptung über den Bestand? Trägt jede Entscheidung eine Begründung? Was bricht, das der Plan nicht nennt?                    |
| plan     | Schnitt und Abhängigkeiten | Lässt sich der Plan zerlegen? Welche Reihenfolge erzwingt er? Sind die offenen Fragen wirklich Stopp-Fragen?                               |
| issue    | Prüfbarkeit                | Ist jedes Akzeptanzkriterium maschinell prüfbar? Steht Manuelles im dafür vorgesehenen Block? Fehlen Randfälle?                            |

Zwei Modelle mit identischem Auftrag sind kein zweiter Blick. Sie sind derselbe Blick zweimal, und er bringt entsprechend wenig. Der Gewinn liegt im Blickwinkel.

## Warum das Arbeitspaket nur noch einen Prüfer hat

Auf den oberen beiden Stufen prüfen zwei Modelle, beim Arbeitspaket eines. Das war nicht immer so. Dort gab es eine zweite Rolle, die nach Scope, Abhängigkeiten und Kollateralschäden im Bestand fragte.

Am 8. August waren drei von vier Scope-Befunden eines Laufs Fehlalarme, jedes Mal an einer Abhängigkeitsgrenze, weil der Prüfer das Nachbar-Issue nicht sah. Diese Fragen entscheiden sich im Plan, nicht im einzelnen Paket. Die Rolle ist deshalb eine Stufe nach oben gewandert, statt zu verschwinden.

Beim Arbeitspaket bleibt die Frage, ob sich jedes Akzeptanzkriterium maschinell prüfen lässt. Sie hat auf den oberen Stufen kein Gegenstück, weil Akzeptanzkriterien erst dort entstehen. Und genau daran ist die Nacht vom 6. August gescheitert.

## Warum ein fremdes Modell dazugehört

Kapitel 20 beschreibt, warum zwei Modelle eine große Schnittmenge an blinden Flecken haben: Sie sind auf ähnlichem Material trainiert. Für Issues kommt dazu, dass Modelle einer Familie dieselbe Vorstellung davon haben, wie ein gutes Issue überhaupt aussieht.

Ein Modell aus einem anderen Haus liegt anders daneben. Wo zwei Verwandte übereinstimmend nichts sehen, sieht ein Fremder manchmal etwas.

Ein Prüfer muss deshalb ein Adapter sein. Ein Eintrag mit `kind: claude` läuft in Claude Code selbst, mit dem Modell, das dabeisteht. Ein Eintrag mit `kind: command` ruft ein beliebiges Kommandozeilenprogramm auf: Der Prompt geht über die Standardeingabe hinein, die Antwort kommt über die Standardausgabe zurück.

```
"issueReview": {
  "rounds": 1,
  "requiredBeforeReady": false,
  "reviewers": [
    { "name": "opus", "kind": "claude", "model": "claude-
opus-5" },
    { "name": "sonnet", "kind": "claude", "model": "claude-
sonnet-5" },
    { "name": "codex", "kind": "command", "cmd": "codex exec" }
  ]
}
```

Damit nimmt jedes Werkzeug teil, das Text liest und Text schreibt. Das Kit kennt das fremde Werkzeug nicht und muss es nicht kennen. Es prüft vor dem Lauf, ob das erste Wort der Kommandozeile im PATH liegt, mehr nicht. Die Liste veraltet dadurch nicht, wenn eine neue Modellgeneration erscheint.

Ein Issue-Body enthält Backticks, Anführungszeichen und Zeilenumbrüche. Ihn durch eine Kommandozeile zu quoten ist genau die Fehlerklasse, die ich aus dem Board-Adapter entfernt habe. Deshalb geht der Prompt über die Standardeingabe.

Die Vorlage bringt nur die eigene Modellfamilie mit. Wer ein fremdes Modell will, trägt es selbst ein, und das ist Absicht: Nicht jeder hat Codex installiert, manche wollen Gemini, und eine Vorlage, die beim ersten Lauf rot meldet, schreckt ab.

## Wer wen prüft, gehört abgelesen

Die Zuordnung steht in `pairs`. Steht der Autor dort, gewinnt sein Eintrag. Sonst greift eine Regel: die vordersten Prüfer, die nicht der Autor sind, so viele wie die Stufe vorsieht. Welches Modell der Autor war, steht als Zeile `Autor-Modell`: im Kontext-Abschnitt, geschrieben von `/issues`.

Meine erste Fassung hatte nur diese Regel. Das ist elegant, spart Pflege und funktioniert, bis ich nachrechne. Bei vier Prüfern greifen immer die vorderen. Der vierte kommt in keinem einzigen Fall zum Zug.

Der vierte war das fremde Modell. Ausgerechnet den Prüfer, der die blinden Flecken der anderen nicht teilt, hatte meine Sortierung stillgelegt. Nichts daran war kaputt. Es lief, es lieferte Befunde, es sah vollständig aus.

Deshalb gibt es `pairs`, und deshalb gibt ein Befehl die fertige Zuordnung aus:

```
node .claude/kit/board.mjs issue-review matrix
```

Wer so etwas aufsetzt, sollte sich die fertige Tabelle ansehen, statt die Regel im Kopf durchzurechnen. Eine Regel, die ich ausrechnen muss, verbirgt genau die Fälle, an die ich nicht gedacht habe.

## Was kann raus

Prüfer schlagen Ergänzungen vor. Immer. Ein fehlender Randfall, ein zusätzliches Kriterium, ein Hinweis auf etwas, das ich bedenken sollte. Das ist ihre Aufgabe, und sie machen sie gut.

Nur ist Ergänzen leichter als Streichen. Nach drei Modellen ist das Issue doppelt so lang, und ein doppelt so langes Issue ist nicht doppelt so gut implementierbar. Es ist oft schlechter, weil die Zusätze das Wesentliche zudecken.

Deshalb steht in beiden Prüfaufträgen eine Frage, die sonst niemand stellt: Was kann raus? Ohne diese Gegenkraft kippt das Verfahren. Wer sie weglässt, bekommt längere Issues statt besserer.

## Wo der Mensch bleibt

Die Befunde gehen als Kommentar ans Issue, mit Schweregrad `BLOCKER` oder `WICHTIG`. Den Body überschreibt der Skill nie automatisch, er zeigt einen Vorschlag und fragt einmal nach.

Die Regel aus Kapitel 11 gilt dabei weiter: Die Verhandlung gehört in den Body, weil dort der Stand steht. Ein Prüfbefund ist kein Stand, er ist ein

Vorschlag, über den ich noch entscheide. Erst was ich übernehme, wandert in den Body.

Zwei Modelle können sich einig und trotzdem falsch sein. Wenn beide dasselbe sagen, heißt das nur, dass beide dasselbe gelesen haben.

Wer entscheidet, was im Issue steht, entscheidet über die Anforderung. Wer über die Anforderung entscheidet, entscheidet über das Produkt. Das ist keine Modellfrage.

Stimme ich zu, trägt das Dokument die Nachweiszeile seiner Stufe, mit allen Prüfern, die tatsächlich gelaufen sind:

```
Fachplan-Review: codex, sonnet (2026-08-11)
Plan-Review: codex, sonnet (2026-08-11)
Issue-Review: codex (2026-08-11)
```

Lehne ich ab, entsteht kein Marker. Ein Review, dessen Ergebnis ich verwurfe, hat das Issue nicht geschärft.

Von den dreien gibt nur `Issue-Review` frei. Die beiden anderen belegen eine frühere Stufe, nicht das Arbeitspaket. Wer sie verwechselt, zieht ein ungeprüftes Arbeitspaket nach Ready.

Das Verfahren fügt dem Prozess keinen vierten Stop-Punkt hinzu. Es schärft den ersten: Ich weiß danach, was ich freigebe.

## Das Gate vor Ready

Mit `requiredBeforeReady: true` stellt der Nacht-Runner Ready-Issues ohne Marker kommentiert ins Backlog zurück und fährt mit dem nächsten fort. Tagsüber weisen die Implementierungs-Skills nur darauf hin und fragen. Nachts antwortet niemand, tagsüber steht ein Mensch daneben.

Der Default ist `false`. Ein Kit-Update darf keinem laufenden Projekt über Nacht den Runner anhalten. Wer das Verfahren einführt, schaltet es bewusst ein.

Nicht jedes Issue braucht die Prüfung, und wer das Gate scharf schaltet, muss eine Tür offen lassen. Sie ist eine Zeile im Kontext-Abschnitt, der Schlüssel ohne Umlaut geschrieben:

Pruefung: Verzicht

Erlaubt sind 1, 2, 3 und Verzicht. Die Ziffer steht für die Zahl der Runden, Verzicht heißt: ohne Prüfung freigegeben. Der Nacht-Runner nimmt das als zweiten Freibegrund, gleichwertig zum Marker.

Zwei Sperren hängen daran. Setzen darf das nur ein Mensch: Sobald der Adapter merkt, dass er unbeaufsichtigt läuft, weist er jede Verringerung ab. Und der Verzicht verfällt, wenn sich das Issue ändert, denn darunter liegt eine Prüfsumme über den Body. Eine Freigabe “das braucht keine Prüfung” gilt für die Aufgabe, die ich gelesen habe, nicht für die, die später daraus geworden ist.

Denselben Weg kann der Nacht-Runner auch von sich aus gehen. Kapitel 19 beschreibt die Review-Nacht: Sie prüft Dokumente aus dem Backlog, statt zu bauen, und setzt den Marker nur, wenn nichts zu ändern war.

## Was es kostet, und was es nicht leistet

Jeder Prüfer ist ein zusätzlicher Lauf. Weil die Prüfung nach oben wandert, sind es weniger, als ich zuerst gerechnet hatte. Ein Plan mit dreizehn Arbeitspaketen kostet siebzehn Läufe, nicht sechsundzwanzig. Zweimal die Anforderung, zweimal der Plan, dann je einmal pro Paket.

Nach meiner Erfahrung lohnt sich das trotzdem nicht bei jedem Einzeiler. Es lohnt sich bei Issues, die etwas kosten, wenn sie falsch sind, und die erkenne ich meistens daran, dass ich beim Schreiben gezögert habe. Deshalb muss ich das Verfahren bewusst einschalten: Ohne den `issueReview`-Block in der Config tut `/issue-review` nichts.

`rounds` steht bei mir auf 1. Weitere Runden finden vor allem Geschmacksfragen. Wenn eine zweite Runde nichts mehr mit Schweregrad BLOCKER oder WICHTIG liefert, sagt der Skill das.

Das Verfahren ersetzt kein fachliches Gespräch. Wenn unklar ist, was gebaut werden soll, hilft kein Modell beim Formulieren. Dafür braucht es jemanden, der die Domäne kennt.

Und es macht aus einer falschen Anforderung keine richtige. Ein Issue, das präzise das Falsche verlangt, ist nach drei Modellen präzise falsch. Das Verfahren macht eine Anforderung präzise, nicht wahr. Das ist weniger, als ich mir wünsche, und mehr, als ich vorher hatte.

KAPITEL 10

# Ein Tag im KI-augmentierten Team

Ich beschreibe einen Tag. Keinen Sprint, kein Release, einen Tag.

Er beginnt mit dem Ernten und endet mit dem Säen. Auf den ersten Blick ist der Tag damit verkehrt herum. Morgens prüfe ich, was in der Nacht entstanden ist, und bringe es nach production. Nachmittags denke ich nach: neue Anforderungen, Grooming, Plan, Issues. Abends gebe ich frei, was die kommende Nacht bauen soll.

Der Nachtlauf ist bei mir der Normalfall. Abends ziehe ich die Karten, sortiere sie, setze die Leitplanken, schneide die Issues sauber. Nachts wird nur gebaut. Die Implementierung ist so billig geworden, dass sie in den Schlaf passt.

| Uhrzeit | Was                                                      | Wer                            |
|---------|----------------------------------------------------------|--------------------------------|
| 8:30    | <code>/kontext</code> , Lageüberblick und Nachtprotokoll | Ich lese                       |
| 8:45    | Ergebnisse der Nacht sichten, In review durchgehen       | Ich                            |
| 9:00    | <code>/local-check</code>                                | Modell, ich lese den Bericht   |
| 9:30    | <code>/review</code>                                     | Zweites Modell, ich entscheide |
| 10:30   | <code>/push-main</code>                                  | <b>Nur ich</b>                 |
| 10:45   | Test-Server im Browser prüfen                            | Nur ich                        |
| 11:15   | <code>/merge-production</code>                           | <b>Nur ich</b>                 |
| 11:30   | <code>/document</code>                                   | Modell                         |
| 13:00   | <code>/fachplan</code> für die neue Anforderung          | Modell, PO groomt              |
| 14:30   | <code>/plan #N</code> aus dem fachlichen Issue           | Modell, ich prüfe              |

| Uhrzeit | Was                                              | Wer                          |
|---------|--------------------------------------------------|------------------------------|
| 15:30   | <code>/issues</code>                             | Modell                       |
| 16:15   | <code>/issue-review</code> über die neuen Issues | Modell prüft, ich entscheide |
| 17:00   | <b>Ready: freigeben und Reihenfolge setzen</b>   | <b>Nur ich</b>               |
| 17:15   | Nacht-Runner starten                             | Ich starte, Modell baut      |

### 8:30 Uhr: `/kontext`

Mein Tag beginnt mit `/kontext`. Der Skill liest meinen Vault, die Projektnotiz und die offenen Issues und gibt mir einen Lagebericht: was in In review liegt, was die Nacht liegen gelassen hat, welche Entscheidung ich vertagt habe.

Ich habe früher in meinen Projekten immer versucht durchzusetzen, dass es so etwas wie ein Entscheidungs-Tagebuch gibt, damit ich später nachvollziehen kann, wer was entschieden hat, warum, und welche Alternativen es gab. Dieses Tagebuch hat nie die zweite Woche überlebt, weil es mühevoll ist, alles aufzuschreiben. Diese Aufgabe übernimmt jetzt die KI. Deshalb brauche ich eine geordnete Dokumentation, die Menschen und Maschinen gemeinsam lesen können.

Ich habe lange ohne diesen Schritt gearbeitet und die ersten zwanzig Minuten jedes Morgens damit verbracht, mich selbst zu sortieren: Board öffnen, Issues überfliegen, den Chat von gestern suchen, den ich dann doch nicht mehr fand. Der Skill kostet zwei Minuten. Ich fange sofort an, statt erst groß recherchieren zu müssen.

Dabei gilt eine Trennung, die ich in Kapitel 17 ausführe: Was `/kontext` lädt, ist mein Gedächtnis, nicht das des Modells. Es liegt in Dateien. Deshalb vergisst der Nacht-Runner nichts, obwohl das Modell alles vergisst: Der Runner macht für jedes Issue eine neue Claude-Session auf, das Modell startet also jedes Mal bei null. Ich habe die Erinnerung aus dem Modell herausgezogen, bevor die Nacht anfang.

## 8:45 Uhr: was die Nacht gebracht hat

Morgens erwartet mich ein Stapel lokaler Commits, die durchgelaufenen Issues in In review, dazu ein Nachtprotokoll.

Der Nacht-Runner startet pro Issue eine frische Session mit `/implement-next`. Ein Issue, eine Session, dann Ende. Das war notwendig, weil der erste Versuch gescheitert ist. Ich wollte mein Board nachts abarbeiten lassen: zwanzig Issues ins Ready-Feld, ein Kommando, schlafen gehen. Der Versuch scheiterte am Kontextfenster, nicht an der KI. Das Modell baut Issue 17 auf dem komprimierten Sediment von 16 Vorgängern. Die Qualität sinkt genau dann, wenn niemand zuschaut.

Parallelität habe ich als Ausweg verworfen. Fünf Sessions im selben Arbeitsverzeichnis committen sich gegenseitig in die Quere, und den Merge-Konflikt um drei Uhr nachts löst keiner auf. Die Lösung war weniger Kontext: Jedes Issue ist das erste und einzige seiner Session, also gibt es keinen Kompressions-Verfall und kein Vergessen mittendrin. Aus demselben Grund laufen bei mir nie mehrere Nacht-Runner im selben Projekt. Verschiedene Projekte mit je einem eigenen Runner sind dagegen kein Problem: Die Konflikte entstehen im geteilten Arbeitsverzeichnis, nicht zwischen Repositories.

Den Erfolg lese ich am Board ab, nicht am Protokoll: Issue in In review heißt fertig. Von acht Issues sind heute sechs durchgelaufen, eines steht noch in Ready, eines ist ins Zeitlimit gelaufen. Ein Issue, das rot geworden ist, wandert zurück ins Backlog. Hängt beim Start noch eines in In progress, stoppt der Runner, statt zu raten, wo der vorige Lauf abgebrochen ist. Und endet ein Lauf mit einem unsauberen Worktree, stoppt er hart. Auf halben Änderungen baut er nicht weiter.

Eine Zeit lang war das das Ende. Dann fiel mir auf, dass ein Teil dieser Abbrüche gar keine waren. Eine Session ohne Aufsicht hat keinen zweiten Zug. Startet sie einen langen Test und ihr Turn endet, bevor das Ergebnis da ist, war die Arbeit fertig und niemand hat es gemerkt. Der Runner prüft an dieser Stelle deshalb selbst nach und gibt einem geretteten Stand eine kurze Extrasession, die aufräumt. Wie das genau abläuft, steht in Kapitel 19.

Gelegentlich passiert der Kaskaden-Fall: Das erste Issue scheitert, zehn andere bauen darauf auf, alle landen wieder im Backlog. Das sieht nach einer verlorenen Nacht aus. Es ist das Gegenteil. Zehn Issues, die auf einem gebrochenen Fundament gebaut worden wären, hätten mich den ganzen nächsten Tag gekostet.

Der Nacht-Runner pusht nie, unter keinen Umständen, mit keinem Flag.

## 9:00 Uhr: `/local-check`

`/local-check` führt die Kommandos aus meiner Config aus, eines nach dem anderen: Build, Tests, Coverage, Mutation Score, statische Analyse, ArchUnit. Danach kommt eine Liste mit grünen Häkchen oder einem roten Stopp.

Ein roter Check blockiert alles Weitere. Es gibt keine Ausnahme, kein Übergehen, kein “das ist nur der Linter”. Eine Leitplanke, mit der sich verhandeln lässt, ist keine.

Deshalb verweigert der Nacht-Runner auch den Start, wenn in der Config keine `buildChecks` stehen und er implementieren soll. Nachts ohne Gate zu implementieren, hieße, stundenlang Code zu produzieren, den nichts geprüft hat. Wer das trotzdem will, muss es mit einem Flag ausdrücklich sagen. Ich finde diese Hürde richtig.

Bei Frontend-Änderungen erinnert der Skill an die manuelle Verifikation im Browser. Die neueren Modelle können inzwischen selbst einen Browser steuern und klicken, das war vor einem Jahr noch anders. Trotzdem prüfe ich selbst. Ob das deployte Bundle im echten Browser auf einer echten Tastatur das Richtige tut, und ob es sich richtig anfühlt, entscheidet der Mensch, der auf den Speichern-Knopf drückt.

## 9:30 Uhr: `/review`

`/review` öffnet eine frische Session ohne den Entstehungskontext und lässt ein zweites Modell über den Diff gehen.

Ein Modell ist gegenüber seiner eigenen Lösung systematisch unkritisch. Es hat sie gerade erzeugt, es würde dieselben Vereinfachungen

noch einmal machen. Wer den Entstehungsweg nicht kennt, liest den Code mit fremdem Blick. Denselben Effekt kenne ich vom Korrekturlesen eigener Texte.

Nach einem Nachlauf ist dieser Schritt wichtiger als nach interaktiver Arbeit. Bei einem Stapel Issues, deren Entstehung ich nicht miterlebt habe, ist das Review meine erste echte Begegnung mit dem Code. Heute kommen elf Befunde zurück, vier weise ich ab, sechs korrigiere ich, einer ist eine zusammengesetzte SQL-Query. Für Security- und Qualitätsmuster verlasse ich mich ohnehin nicht auf das Modell allein. Dafür laufen SonarQube, gitleaks und Semgrep im CI, und was SonarQube findet, kommt als Issue zurück ins Backlog. Kapitel 20 beschreibt diesen Rücklauf. Ein deterministisches Werkzeug teilt mit keinem Sprachmodell einen blinden Fleck.

Meine Freigabe läuft nie nebenher. Ich gebe morgens frei, bewusst.

Je autonomer die KI arbeitet, desto strenger müssen die Stop-Punkte sein. Ein Nacht-Runner ohne Stop-Punkte ist kein Fortschritt. Er ist ein Risiko, das nur größer wird.

## 10:30 Uhr: der Push

Der zweite Stop-Punkt. Ich tippe die Trigger-Phrase, sonst passiert nichts. Kein Skill und kein Runner darf `/push-main` selbst aufrufen, und eine Freigabe von gestern gilt nicht für die Commits von heute Nacht. Jeder Batch braucht seine eigene Freigabe.

Warum ich darauf bestehe, steht in Kapitel 3. Hier zählt die praktische Seite: Zwischen Commit und Push liegt die letzte Stelle, an der ich den Scope überdenken kann. Ich benutze sie.

## 10:45 Uhr: Test-Server und Merge

Der Test-Server zieht main. Ich öffne den Browser und prüfe: Golden Path, die kritischen Randfälle, keine sichtbaren Regressionen. Dafür gibt es bewusst keinen Skill.

Dieser Schritt ist nicht optional. Ein Docker-Upgrade hat mir einmal Traefik geschlachtet. Ein Token brachte nur `PENDING` statt `USER` mit. Ein CORS-Header kannte genau eine Origin zu wenig. Alle drei wurden erst im Smoke-Test auf dem Server sichtbar, obwohl die Tests grün waren.

Stimmt alles, erstellt `/merge-production` den Pull Request von main nach production. Den Merge selbst klicke ich. Ich habe auf dem Test-Server geprüft, also trage ich die Verantwortung, also mache ich es.

Am Ende seines Laufs legt der Skill mir die fertige Zeile für den Git-Tag hin, mit dem Hash des Versions-Commits. Abschicken muss ich sie selbst. Ein Git-Tag ist eine Veröffentlichung, und die gehört zu den Dingen, die ich nicht wegautomatisiere.

`/document` schreibt danach den Tageslog in den Vault und aktualisiert die Projektnotiz. Dort landen auch die Sätze, die mich sonst ein zweites Mal Zeit kosten würden, etwa dass die Strato-Wildcard nur eine Subdomain-Ebene abdeckt. Das ist die Datei, die `/kontext` morgen früh wieder lädt.

Es ist halb zwölf. Die Nacht ist abgearbeitet und in Produktion. Der Rest des Tages gehört dem Denken.

### 13:00 Uhr: `/fachplan` und die PO-Schleife

Nachmittags kommt die neue Anforderung. Bei mir läuft sie über die PO-Schleife, weil ein Product Owner sie fachlich abnehmen will, bevor Technik entsteht.

`/fachplan` überführt die rohe Anforderung, diktiert oder aus einer Mail, in genau ein fachliches Issue. Titel mit dem Präfix `[Fachlich]`, Body im Story-Format: Ziel, fachliche Akzeptanzkriterien, Nicht-Ziele, offene Fragen an den PO. Strikt technikfrei, in PO-Sprache, also ohne Plan, ohne Code, ohne technische Issues.

Das Issue ist das Übergabe-Artefakt. Es geht an die PO-Rolle, die im Body groomt, nicht in Kommentaren. Bis zum Nachmittag kommt es zurück. Kapitel 11 beschreibt die Regeln dahinter und die Leitplanken, die verhindern, dass jemand so ein Issue versehentlich implementiert.

## 14:30 Uhr: /plan und /issues

Kommt die fachliche Freigabe, rufe ich `/plan #N`. Der Skill liest das fachliche Issue mit seinem vollständigen Body als Anforderungsquelle und erzeugt daraus einen technischen Plan: Ziel und Wirkung, betroffene Bereiche und Dateien, architektonische Entscheidungen mit Begründung, offene Fragen, geplante Verifizierung.

Ich lese das ernsthaft, nicht überfliegend. An dieser Stelle wandert die Denkarbeit nach vorn. Was ich hier nicht kläre, kläre ich nachts nicht. Ein Plan, der hängt, ist immer besser als ein Commit, der schon irgendwie passt.

Die Fragen, die hier hängen bleiben, sind die architektonischen. Soll der Plot im Backend oder im Frontend aggregieren? Das beantwortet kein Modell für mich, das ist eine Eigentums-Sache.

`/issues` schneidet den freigegebenen Plan dann in technische Issues nach dem Format aus Kapitel 7. Der Rückverweis auf das fachliche Issue geht in den Kontext, nicht in die Abhängigkeiten, sonst greift die Henne-Ei-Falle.

Ab hier ist das Issue die Quelle der Wahrheit. Die Übergabe an die Nacht läuft über das Issue, nicht über den Chatverlauf. Fällt mir später etwas auf, ändere ich das Issue.

Meine Faustregel für den Schnitt ist vor einem Nachtlauf strenger als sonst: Ein Issue muss so eindeutig sein, dass es ohne Rückfrage umsetzbar ist. Tagsüber kann ich eine Unklarheit im Gespräch auflösen. Nachts wird sie geraten.

## 16:15 Uhr: /issue-review

Genau deshalb schiebe ich vor dem Ready noch eine Runde ein, wenn die Issues in die Nacht sollen. `/issue-review` lässt ein anderes Modell auf jedes frische Arbeitspaket schauen. Übernehmen muss ich die Befunde selbst, das Verfahren steht in Kapitel 9.

Eine Viertelstunde, und es ist die billigste des Tages. Ein unklares Kriterium kostet mich nachts eine ganze Runde und morgens die Zeit, herauszufinden, warum das Ergebnis nicht passt.

## 17:00 Uhr: das Ready

Jetzt kommt der erste Stop-Punkt, und abends wiegt er schwerer als morgens.

Ich ziehe Issues von Backlog nach Ready und bringe sie per Drag&Drop in die Reihenfolge, in der sie gebaut werden sollen, denn der Runner arbeitet die Spalte von oben nach unten ab. Abhängigkeiten trage ich als `#N` im Abhängigkeiten-Abschnitt ein. Fehlt das, kann der Runner nicht entscheiden, ob das Fundament noch trägt, und baut auf Sand weiter. Dann setze ich das Routing-Label `kit:nightrun` auf die Issues, die wirklich nachts laufen sollen. Alles andere in Ready bleibt liegen und wartet auf interaktive Arbeit am nächsten Tag.

In dieser Viertelstunde liegt meine ganze Planung: wie viel Arbeit die Nacht bekommt, in welcher Reihenfolge, mit welchen Abhängigkeiten. Der Runner hat ein Limit pro Nacht, und ich setze es bewusst niedrig, weil ich am Morgen jedes Ergebnis reviewen muss. Den Takt bestimmt der langsamste menschliche Schritt, nicht der schnellste der Maschine.

## 17:15 Uhr: den Nachtlauf starten

Ich starte immer erst mit `--dry-run`. Der Lauf zeigt, welche Issues er anfassen würde, welche er wegen fehlender Labels überspringt und welche er wegen offener Abhängigkeiten zurückstellt. Das kostet zehn Sekunden.

Sieht die Liste richtig aus, starte ich den echten Lauf. Der Runner nimmt sich Issue für Issue, startet pro Issue eine frische Session, wartet auf ihr Ende und prüft am Board, ob es in In review gelandet ist. Dabei benutze ich immer den `--verbose`-Modus: Wenn etwas schiefgeht, sehe ich Schritt für Schritt, wo und warum.

Dann mache ich den Rechner zu. Der Mensch bleibt am Steuer. Auch wenn er schläft.

## Wie ein schlechter Tag aussieht

Dieser Tag lief durch. Die meisten Tage tun das nicht.

Am Anfang habe ich den Nachlauf tagsüber laufen lassen, unter Aufsicht, weil immer irgendwelche Checks fehlten. Ich musste mich da ein bisschen eingrooven. Die Permissions stehen in der `settings.json`, und ich setze sie grundsätzlich explizit, statt dem Modell zu erlauben, alles zu machen, was es will. Es hat bestimmt eine Woche gebraucht, bis der erste Nachlauf wirklich durchlief.

Der häufigste Bruch passiert am Morgen bei `/local-check`. Der Mutation Score fällt unter die Schwelle, ich gehe zurück zu Schritt 5, und der Vormittag ist weg. Der zweithäufigste passiert im Review, wenn sich ein Befund als Designfrage entpuppt und nicht als Korrektur. Dann wird aus dem Review ein neues fachliches Issue, und das Feature geht heute nicht auf production. Das ist kein Scheitern des Prozesses, das ist der Prozess.

Manche Fehler brauchen mehrere Anläufe, und das ist normal. An einem Widget-Flip-Bug habe ich vier Iterationen gebraucht: erst `functional setState`, dann ein stabilerer `droppingItem`, dann `data-grid` auf den Wrappern, dann eine `minHeight` auf dem Grid. Das ist Bug-Hunting. Ein Modell allein hätte beim ersten Versuch aufgehört, weil die Tests grün waren. Ein Mensch allein hätte beim dritten Versuch frustriert irgendetwas hingeschrieben.

Am meisten hüte ich mich vor dem Tag, an dem nachts alles durchläuft und ich morgens das Review überspringe, weil ja alles grün ist. Wer KI unkontrolliert arbeiten lässt, produziert schneller schlechten Code. Ich merke es nur nicht am selben Morgen.

## KAPITEL 11

# Die PO-Schleife: fachliche und technische Issues

In der Praxis bringt ein Product Owner die Anforderungen ein und will den Plan fachlich abnehmen, bevor Technik entsteht. Das ist eine berechnete Erwartung, und sie kollidiert mit dem Prozess aus Kapitel 8, wenn ich ihn wörtlich nehme. Dort entsteht aus der Anforderung sofort ein technischer Plan, und plötzlich wird es sehr konkret: In welcher Datei soll was geändert werden? Das interessiert den PO überhaupt nicht.

Die PO-Schleife löst das, indem sie das Fachliche vom Technischen trennt. Wer keinen PO hat, überspringt dieses Kapitel. `/plan` direkt aufzurufen bleibt der Normalweg.

## Vier Sorten Issues

| Art                      | Präfix                  | Kommt von                                        | Wird implementiert |
|--------------------------|-------------------------|--------------------------------------------------|--------------------|
| Idee                     | <code>[Idee]</code>     | von Hand                                         | nie                |
| Fachliches Issue         | <code>[Fachlich]</code> | <code>/fachplan</code>                           | nie                |
| Plandokument             | <code>[Plan]</code>     | <code>/plan</code> , in Bahn 2 nach der Freigabe | nie                |
| Technisches Arbeitspaket | kein Präfix             | <code>/issues</code>                             | ja                 |

**Ideen** tragen `[Idee]`. Das ist die rohe Notiz aus einem Gespräch, die noch niemand gesichtet hat. Sie entsteht von Hand, hat kein Format und wird nie implementiert. Aus einer Idee wird erst etwas, wenn `/fachplan` oder `/plan` sie anfasst.

**Fachliche Issues** tragen das Präfix `[Fachlich]` im Titel. Sie beschreiben in PO-Sprache das Was und das Warum: Ziel, fachliche Akzeptanzkriteri-

en, Nicht-Ziele, offene Fragen an die PO-Rolle. Strikt technikfrei. Die PO-Rolle groomt sie am Board, implementiert werden sie **nie**.

**Plandokumente** tragen `[Plan]`. Ist der Plan in Bahn 2 freigegeben, legt `/plan` ihn als eigenes Issue ab, mit dem Plantext als Body und dem Modell, das ihn geschrieben hat, im Kopf. Damit überlebt der Plan die Sitzung, in der er entstanden ist: Ich kann ihn am nächsten Tag zerlegen, jemand anderes kann es auch, und wer will, prüft ihn vorher. `/issues #N` zerlegt ihn. In Bahn 1 entsteht kein Plandokument, dort gibt es keinen Plan.

**Technische Issues** haben das Vier-Abschnitt-Format aus Kapitel 7. Sie entstehen erst, wenn die fachliche Freigabe steht.

Die Trennung folgt dem Discovery-Delivery-Muster. Die PO-Rolle soll über den fachlichen Inhalt streiten können, ohne dabei über Dateinamen zu reden. Solange beides in einem Dokument steht, wird über beides gestritten oder über keines.

Drei der vier Sorten werden nie implementiert. Missversteht eine Session ein Plandokument als Arbeitsauftrag, setzt sie es um, und es wandert nach In review. Am Board sieht das aus wie ein Erfolg. Was tatsächlich passiert ist: Ein Modell hat einen Plan gebaut, statt ein Feature.

## Der Ablauf

1. `/fachplan <Anforderung>` erzeugt das fachliche Issue.
2. Die PO-Rolle groomt daran, bis die fachliche Freigabe steht.
3. `/plan #N` liest das fachliche Issue mit vollständigem Body und erzeugt den technischen Plan.
4. `/issues #N` zerlegt das Plandokument in technische Issues. Jedes trägt die Rückverweise im Kontext-Abschnitt.
5. Ab hier der normale Weg: Ready, Implementierung, Review, Push.

Zwei Rückverweise, beide im Kontext, untereinander, in dieser Reihenfolge:

Plan: Issue #M  
 Fachliche Quelle: Issue #N

Die `Plan:`-Zeile entsteht nur, wenn ein Plandokument als Quelle vorlag. Habe ich den Plan in derselben Sitzung freigegeben und zerlegt, bleibt sie weg. Sie hat nichts mit der `Plan-Modell:`-Zeile zu tun, die weiter oben im Issue steht. Dort steht, wer den Plan geschrieben hat, hier, wo er liegt.

Ob neue fachliche Issues im kanban-kit direkt im Backlog landen oder zunächst im Ideen-Pool, entscheidet `toolbox.ideaStored` in der `workflow.config.json`. Ohne Eintrag gilt `false`, dann gehen sie ins Backlog. Wer mit der PO-Schleife arbeitet, setzt das Flag: Dann kommt die Anforderung erst in den Pool und wandert ins Backlog, sobald jemand sie eingeplant hat.

Die drei Stufen haben dort eine klare Bedeutung. Ideen-Pool heißt: liegt vor, noch nicht eingeplant. Ins Backlog eingeplant heißt fachlich in Arbeit, ab da ist das Issue adressierbar und groombar. `/plan #N` aufzurufen heißt fachlich freigegeben.

## Die Verhandlung läuft im Body

Diese Regel habe ich oft brechen sehen. Stünden die Antworten als Kommentare unter dem Issue, wäre die fachliche Dokumentation zerstückelt, ein Teil im Body, ein Teil in den Kommentaren.

Deshalb wandern PO-Antworten und Ergänzungen direkt in den Text des Issues. Das fühlt sich zuerst falsch an, weil ich den Diskussionsverlauf verliere. Genau das will ich. Am Ende soll dort eine Anforderung stehen, kein Protokoll einer Diskussion.

Am meisten bringen dabei die **Nicht-Ziele**. Dort sagt die PO-Rolle, was ausdrücklich nicht gebaut wird. Das ist die Grenze, die das Modell später nicht von allein zieht, weil es zu jeder Unklarheit eine plausible Erweiterung erzeugt und jede einzelne davon sinnvoll aussieht.

## Drei Leitplanken

**Was nie implementiert wird, geht nie nach Ready.** Ready heißt implementierbar, und ein fachliches Issue ist es nicht, eine Idee auch nicht, ein Plandokument erst recht nicht. Diese Regel sichere ich mechanisch ab: Landet doch eines in Ready, erkennen die Implementierungs-Skills und der Nacht-Runner es am Präfix und stellen es kommentiert zurück ins Backlog, ohne überhaupt eine Session zu starten. Ich verlasse mich nicht darauf, dass ich abends aufpasse.

**Die Rückverweise stehen im Kontext, nie in den Abhängigkeiten.** Sonst entsteht die Henne-Ei-Falle aus Kapitel 7: Das fachliche Issue wird erst fertig, wenn die Kinder fertig sind, und jedes Kind wartet auf den Eltern-Teil. Beim Plandokument ist es noch schlimmer, denn niemand setzt es um, also wird es nie fertig. Seine Kinder blieben für immer zurückgestellt.

**Das fachliche Issue bleibt als Klammer offen.** Auf Done setzt es ein Mensch, wenn die technischen Kinder durch sind. Kein Skill schließt es.

## Was die Schleife kostet und was sie bringt

Sie kostet einen zusätzlichen Schritt und eine Runde Warten auf den PO. In einem Projekt ohne PO wäre das Bürokratie.

Sie bringt zwei Dinge. Erstens trennt sie die Verantwortung ehrlich: Die PO-Rolle verantwortet das Was und kann es abnehmen, ohne über Technik zu urteilen, für die sie nicht zuständig ist. Zweitens macht sie die technischen Issues besser, weil `/plan` auf einem Text aufsetzt, der eine Runde Klärung hinter sich hat. Spezifikationsqualität ist der Engpass. Wer unscharf beschreibt, bekommt unscharfen Code, nur eben in höherem Tempo.

## PO und Proxy-PO

In der Praxis, die ich kenne, teilen sich zwei Personen die PO-Rolle, und die PO-Schleife bildet genau das ab.

Auf der einen Seite steht der fachliche Product Owner, meist jemand aus dem Geschäft. Diese Person kennt die Domäne, die Regeln und die Gründe. Sie weiß, warum eine Frist am Monatsletzten endet und welcher Sonderfall in der Buchhaltung wirklich vorkommt. Sie schreibt keine Issues, sie erklärt Sachverhalte.

Auf der anderen Seite steht der Proxy-PO, häufig aus der IT oder an ihrer Schnittstelle. Diese Person kennt die Fachlichkeit gut genug, um sie zu verstehen, und die Entwicklung gut genug, um sie so aufzuschreiben, dass ein Team daraus bauen kann. Sie übersetzt zwischen zwei Sprachen, und genau dafür gibt es die Rolle in vielen Organisationen.

Genau hier setzt das fachliche Issue an. An ihm arbeiten beide, ohne sich zu behindern. Der fachliche PO liest Ziel, Akzeptanzkriterien und Nicht-Ziele und kann sagen, ob das stimmt, ohne eine Zeile Technik zu sehen. Der Proxy-PO formuliert, schärft und stellt die offenen Fragen, die ihm beim Blick auf die Umsetzung auffallen. Erst wenn beide zufrieden sind, entsteht mit `/plan #N` der technische Plan.

Ohne diese Trennung passiert regelmäßig eines von zwei Dingen. Entweder bekommt der fachliche PO ein Dokument mit Dateinamen und Schichten auf den Tisch und winkt es durch, ohne es beurteilen zu können. Oder der Proxy-PO rät die Fachlichkeit und merkt den Fehler erst, wenn das Feature auf dem Test-Server steht. Das fachliche Issue macht die Übergabe zwischen beiden sichtbar und überprüfbar.

KAPITEL 12

# Takt und Events

Ein Prozess lebt von seinen Wiederholungen. Die folgenden fünf Events bilden das Mindestgerüst eines KI-augmentierten Teams. Sie ersetzen die Scrum-Events nicht eins zu eins, aber sie greifen die Funktionen auf, die dort Sprint Planning, Daily, Review und Retrospektive leisten.

| Event                     | Takt                | Wer                  | Zweck                                                |
|---------------------------|---------------------|----------------------|------------------------------------------------------|
| Daily-Standup             | Täglich, 15 Minuten | Menschen             | Was steht an, was ist reviewbereit, welche Releases? |
| Issue-Grooming            | Kontinuierlich      | PO, Senior           | Backlog klären, Issues reif für Ready machen         |
| Implementations-Iteration | Stunden bis 1 Tag   | Modell, Mensch prüft | Ready-Issues umsetzen                                |
| Stakeholder-Review        | Wöchentlich         | Team, Stakeholder    | Stand zeigen, Feedback einsammeln                    |
| KI-Retrospektive          | Alle 1 bis 2 Wochen | Menschen             | Mensch-Modell-Zusammenarbeit reflektieren            |

## Daily-Standup

Fünfzehn Minuten, unter Menschen. Drei Fragen: Welche Issues wandern heute von Ready nach In progress? Welche stehen in In review zur Freigabe an? Welche Push- oder Merge-Entscheidungen fallen heute?

Das Modell nimmt am Daily nicht teil. Das Daily koordiniert zwischen Menschen, die Verantwortung tragen. Das Modell hat keine eigene Tageslage, kein Gefühl für Stakeholder-Druck, keinen Kalender. Was es wissen muss, steht im Issue, am Board und in den Konventionsdateien.

Ich halte es für offen, ob das Daily in kleinen Teams langfristig überlebt. Wenn das Team sich ohnehin über Board, Pull Requests und Pipelines synchronisiert, verliert ein festes morgendliches Treffen an Wert. Drei Menschen brauchen kein fünfzehnminütiges Standup, sie können sich fünf Minuten an der Kaffeemaschine besprechen. Wo das Daily bleibt, erfindet es sich als Abstimmung zwischen den prüfenden Menschen neu, nicht als Statusrunde.

## Issue-Grooming

Grooming läuft bei mir nebenher, nicht als eigenes Meeting. Mindestens einmal täglich, typischerweise als Teil der Morgenroutine, geht die PO-Rolle durch den Backlog. Welche Issues haben offene Designfragen, die ich heute klären kann? Welche sind reif für Ready? Welche sind veraltet und können weg?

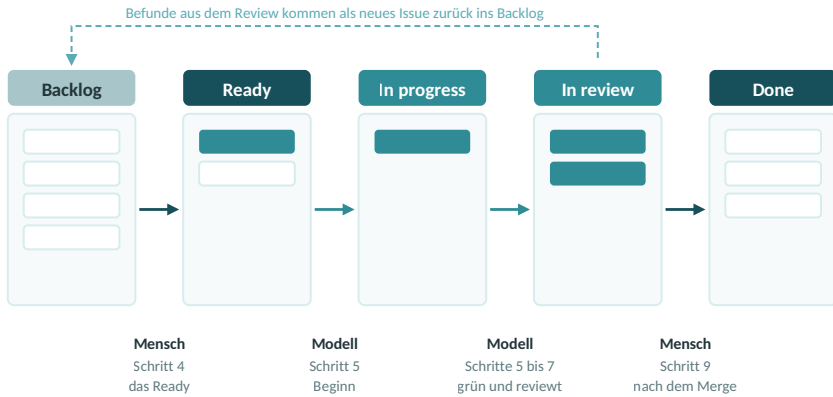
Die dritte Frage stelle ich mir erst, seit ein Backlog von mir zur Ablage geworden ist. Ein Backlog, den niemand aufräumt, ist kein Vorrat, er ist Schuldenlast. Der Ideen-Pool aus Kapitel 5 hilft dabei, weil ich Dinge parken kann, ohne sie wegzuerwerfen und ohne dass sie das Board vermüllen.

Das Modell ist beim Grooming nützlich: Issues nach Diktat formulieren, Akzeptanzkriterien vorschlagen, Designfragen aufdecken, die nicht offensichtlich sind. Nach Ready schiebt es kein Issue. Dieser Schritt macht daraus einen verbindlichen Auftrag.

## Implementations-Iteration

Das zentrale Arbeits-Event. Es beginnt mit einer Trigger-Phrase und endet, wenn Ready leer ist oder das Limit erreicht. Jedes Issue durchläuft dieselben Schritte: nach In progress, Issue lesen, Tests und Code schreiben, lokal committen, nach In review.

Sie liefert Commits und Issues in der In-review-Spalte. Nichts ist gepusht, nichts ist in Produktion.



Zwei der vier Bewegungen macht ein Mensch, und es sind die erste und die letzte.

In progress enthält nur, was das Modell gerade implementiert. Rückwege führen nie dorthin.

Bei mir läuft diese Iteration überwiegend nachts. Das ändert an ihrer Struktur nichts, es verschiebt nur, wann sie stattfindet und wann ich prüfe. Kapitel 10 zeigt den Tag, der daraus entsteht.

## Stakeholder-Review

Wöchentlich, mit der auftraggebenden Person, auf dem Test-Server. Hier wird “funktionierende Software über umfassende Dokumentation” konkret: Ich zeige keinen Foliensatz. Ich zeige die laufende Anwendung.

Ich halte an einem festen wöchentlichen Termin fest, obwohl der Takt der Arbeit viel kürzer ist. Stakeholder brauchen Verlässlichkeit, und ein Termin, den es immer gibt, senkt die Schwelle für Kritik, die sonst niemand ausspricht.

Aus dem Review entstehen neue Backlog-Einträge, manchmal Korrekturen an bestehenden Issues. Das Modell kann diese Anpassungen unmittelbar nach dem Review mitformulieren, solange das Gespräch noch frisch ist. Ich kann auch schon im Review Notizen diktieren und sie später sortieren und den einzelnen Issues zuordnen.

## KI-Retrospektive

Das einzige Event, das es in Scrum so nicht gibt. Kapitel 23 gehört ihm allein. Sie ist kein Tagesordnungspunkt der allgemeinen Retrospektive. Sie ist ein eigenes Format alle ein bis zwei Wochen. Drei Fragen: Wo hat die Zusammenarbeit gehakt? Welche Memory-Einträge sind veraltet? Welche Workflow-Regel braucht eine Schärfung?

Sie ändert Dateien. Eine Retrospektive, die keine Datei verändert, war zu abstrakt.

## Was sich gegenüber Scrum verschoben hat

Drei Beobachtungen, die ich für belastbar halte.

**Das Team stimmt sich seltener ab.** Der Status steht am Board und ist jederzeit lesbar. Ein Treffen, das diesen Status nur verliert, hat seinen Zweck verloren.

**Die Arbeits-Events sind schneller geworden, die menschlichen nicht.** Die Implementations-Iteration ist von zwei Wochen auf Stunden geschrumpft. Das Stakeholder-Review ist wöchentlich geblieben, weil daran Menschen beteiligt sind, deren Kalender sich nicht beschleunigt hat.

**Ein Event ist dazugekommen.** Der Prozess muss sich selbst überarbeiten können, weil sich Werkzeuge und Konventionen monatlich verschieben. Die klassische Retro leistet das nicht, weil sie über Menschen und Zusammenarbeit spricht, nicht über Konventionsdateien.

## KAPITEL 13

# Iterationsmodelle

Wie lang eine Iteration ist, bestimmt, wie schnell ein Team auf Feedback reagiert, wie viel Arbeit zwischen zwei Reviews liegt und wie häufig Fehler in Produktion landen.

Vier Modelle stehen zur Wahl, und sie schließen sich nicht aus. Bei mir laufen alle vier nebeneinander, jedes für eine andere Art von Arbeit.

### Die Tages-Iteration als Default

Für die meisten Teams halte ich einen Tag für den realistischen Default. Morgens das Grooming, tagsüber die Implementierung, abends der Push und, wenn es passt, der Merge.

Dieser Takt hat zwei Vorteile. Er ist überschaubar, ein Mensch kann am Abend ehrlich beurteilen, was er am Morgen versprochen hat. Und er liefert täglich Ergebnisse auf den Test-Server, was Vertrauen bei Stakeholdern aufbaut. Wer mit weniger Takt arbeitet, lässt im Hintergrund Dinge entstehen, die niemand mehr nachvollziehen kann.

### Die Vier-Stunden-Iteration

Schnellere Iterationen sind möglich, aber nicht für alle Arbeitsarten geeignet. Sie passen zu klar abgegrenzten Mikro-Features, kleinen UI-Verbesserungen, Bugfixes, bei denen Feedback unmittelbar einfließen soll. Sie passen, wenn das Team in engem Austausch mit der auftraggebenden Person steht.

Sie passen nicht für Architektur-Arbeit. Ein neues Modul, ein größeres Refactoring, eine Schemaerweiterung in der Datenbank brauchen Denkzeit, in der niemand alle vier Stunden einen Zwischenstand erzwingt. Wer hier zu eng taktet, bekommt eine Folge halbgarer Pushes, von denen jeder einzeln den Test-Server destabilisiert.

Solange die Iteration ein abgeschlossenes, deploybares Issue produziert, ist sie sinnvoll. Sobald ein Issue gezwungen wird, in zwei Iterationen zerteilt zu werden, weil der Takt drängt, ist der Takt falsch.

## Die Nacht-Iteration

Das ist bei mir der Normalfall. Abends gebe ich frei, nachts wird gebaut, morgens prüfe ich und liefere aus. Kapitel 10 beschreibt den Tag, der daraus entsteht, Kapitel 19 die Technik dahinter.

Die Nacht setzt eine harte Grenze. Zwischen Freigabe und Prüfung liegen acht Stunden, in denen niemand nachsteuern kann. Das erzwingt genau die Disziplin, die tagsüber optional wirkt: eindeutige Issues, eingetragene Abhängigkeiten, verdrahtete Pflicht-Checks.

Meine Grenze ist der Morgen, nicht die Nacht. Ich gebe so viele Issues frei, wie ich am nächsten Vormittag reviewen kann. Wer zwanzig Issues bauen lässt und sechs reviewt, hat keinen produktiven Tag gehabt. Er hat Schulden aufgenommen.

## Continuous AI

Parallel zur menschlich getakteten Iteration können Hintergrund-Loops laufen: Triage, Dokumentationspflege, kleine Qualitätsverbesserungen. Der Name ist an Continuous Integration angelehnt: Statt Builds laufen hier KI-Agenten kontinuierlich im Hintergrund. Sie öffnen kleine Pull Requests, die ein Mensch reviewt und merged.

In meinem eigenen Prozess gibt es solche Loops bisher nicht. Ich halte sie für eine gute Ergänzung. Ein Ersatz sind sie nicht. Was diese Loops produzieren, muss reviewbar sein und durch dieselbe menschliche Freigabe gehen wie alles andere. Wer Continuous AI ohne diese Disziplin einsetzt, bekommt ein Repository, das sich in eine Richtung entwickelt, die niemand bewusst gewählt hat.

Die Trennlinie, die ich ziehe: In den Hintergrund gehört, was kein menschliches Urteil braucht, während es entsteht. Alles, was Architektur-Entscheidungen, Sicherheitsabwägungen oder Stakeholder-Kommunikation berührt, gehört in die menschlich getaktete Iteration.

## Sprint oder Flow

Bleibt die Grundsatzfrage. Klassisches Scrum arbeitet in Sprints, also festen Zeiträumen mit klarem Anfang und Ende. Kanban arbeitet im Flow.

Für KI-augmentierte Entwicklung ist der Flow aus meiner Sicht passender. Issues sind kleinteilig, sie dauern Stunden statt Tage. Ein Zwei-Wochen-Sprint mit dreißig Issues ist eher Bürokratie als Hilfsmittel. Ein Board mit klarer Reihenfolge in Ready ist die natürliche Form.

Sprints behalten ihren Sinn dort, wo ein externer Takt sie erzwingt: eine zweiwöchige Release-Vorgabe, ein fester Stakeholder-Rhythmus, ein Vertrag, der Zwischenstände vorschreibt. Das ist kein Rückfall. Wer seine Umgebung nicht allein kontrolliert, passt sich ihr an.

## Wie ich die Modelle mische

| Arbeitsart                                  | Modell                            |
|---------------------------------------------|-----------------------------------|
| Klar geschnittene Features, mehrere pro Tag | Nacht-Iteration                   |
| Architektur, größere Refactorings           | Tages-Iteration, interaktiv       |
| Bugfixes mit direktem Feedback              | Vier-Stunden-Iteration            |
| Alles, was ein Stakeholder sehen muss       | Wöchentlicher Review-Takt darüber |

Diese Mischung ist kein Modell, das ich empfehle. Sie ist gewachsen. Der gemeinsame Nenner liegt in der Regel darunter: Der Takt wird vom langsamsten menschlichen Schritt bestimmt, nicht vom schnellsten Werkzeug-Schritt. Wer diese Regel einhält, kann die Taktlänge frei wählen.

Wer sie verletzt, produziert in jedem Takt dasselbe Ergebnis, nur unterschiedlich schnell.

## TEIL IV

# Einrichtung und Betrieb (technisch)

Dieser Teil ist zum Nachschlagen gedacht, nicht zum Durchlesen. Wer den Prozess erst einmal verstehen will, überspringt ihn und kommt zurück, sobald er etwas einrichten muss.

Sieben Kapitel: das Board betreiben, das Kit installieren, die Skills im Einzelnen, die Kontextdateien und der Vault, die Wahl zwischen GitHub, GitLab und lokal, der Nachtbetrieb, und zuletzt die Qualitäts- und Security-Gates.

Zwei Hinweise vorweg. Erstens: Was hier steht, ist der Stand bei Redaktionsschluss. Konfigurationsdetails ändern sich schneller, als ein gedrucktes Buch nachziehen kann. Die gepflegte Referenz steht online, und dieser Teil erklärt die Zusammenhänge, nicht jeden Schalter.

Zweitens: Kapitel 20 ist das wichtigste dieses Teils. Wer sonst nichts aus Teil IV braucht, sollte wenigstens dieses eine Kapitel lesen. Es handelt davon, warum Qualitätssicherung und Sicherheit nicht ins Modell gehören, sondern in Werkzeuge, die den Build zum Scheitern bringen.

## KAPITEL 14

# kanban-kit betreiben

Hier steht, wie ich das kanban-kit betreibe. Wer GitHub Projects, GitLab oder den lokalen Modus benutzt, kann dieses Kapitel überspringen.

Ein Hinweis vorweg für den ganzen Teil IV: Was hier steht, ist der Stand bei Redaktionsschluss. Konfigurationsdetails ändern sich schneller, als ein gedrucktes Buch nachziehen kann. Die laufend gepflegte Referenz steht online unter [kanban.mwolff.org/docs](https://kanban.mwolff.org/docs) und [docs.mwolff.org](https://docs.mwolff.org). Dieses Kapitel erklärt die Zusammenhänge, nicht jeden Schalter.

## Der Stack

Vier Container über Docker Compose:

- **Caddy** für TLS und als Reverse Proxy
- **manban-api**, das Spring-Boot-Backend, das auch das gebaute Frontend ausliefert (`manban` ist der historische Arbeitstitel des kanban-kit, der Name steckt noch in Containern und Datenbank)
- **Postgres** für die Daten
- **MinIO** als Objektspeicher für Anhänge

Ohne Docker-Laufzeit geht nichts. Auf macOS benutze ich Colima. Wenn `docker ps` meldet, dass es keine Verbindung zum Daemon gibt, läuft sie nicht.

## Starten

Im Repo-Verzeichnis, dort wo die `docker-compose.yml` liegt:

```
docker compose up --build -d
```

Das `--build` ist wichtiger, als es aussieht. Es baut das Image neu, inklusive npm-Build des Frontends und Maven-Jar. Nach jeder Codeänderung

brauche ich es, sonst startet ein reines `docker compose up -d` das alte Image, und ich suche den Fehler an der falschen Stelle.

Danach im Browser `https://localhost`. Für localhost benutzt Caddy ein selbstsigniertes Zertifikat, der Browser warnt entsprechend, und das ist lokal so gewollt. Für eine echte Domain setze ich `MANBAN_DOMAIN`, dann läuft Let's Encrypt automatisch.

## Die Umgebungsvariablen, die zählen

Ich konfiguriere alles über eine `.env` neben der `docker-compose.yml`. Compose lädt sie automatisch, und sie steht in der `.gitignore`.

| Variable                                  | Bedeutung                              | Default                        |
|-------------------------------------------|----------------------------------------|--------------------------------|
| <code>MANBAN_BASE_URL</code>              | Basis-URL für Links in E-Mails         | <code>https://localhost</code> |
| <code>MANBAN_BOOTSTRAP_ADMIN_TOKEN</code> | Einmal-Token für den ersten Admin      | leer                           |
| <code>MANBAN_MAIL_ENABLED</code>          | Echten Mailversand aktivieren          | <code>false</code>             |
| <code>MANBAN_CLEANUP_ENABLED</code>       | Aufräum-Jobs für Done und Papierkorb   | <code>true</code>              |
| <code>MANBAN_DONE_RETENTION_DAYS</code>   | Tage bis Done-Karten archiviert werden | <code>30</code>                |
| <code>MANBAN_SESSION_SECRET</code>        | HMAC-Secret der Session-Cookies        | Dev-Default                    |
| <code>MANBAN_COOKIE_SECURE</code>         | Session-Cookie nur über HTTPS          | <code>true</code>              |

Zwei davon setze ich in Produktion immer: `MANBAN_SESSION_SECRET`, weil der Dev-Default bekannt ist, und `MANBAN_BASE_URL`, weil sonst Links in E-Mails auf localhost zeigen.

## Ohne Mailserver arbeiten

Im Standard ist der Mailversand aus. Verifikations-, Passwort-Reset- und Einladungslinks landen stattdessen im Log:

```
docker compose logs manban-api | grep "Verifikations-Link"
```

Den geloggtten Link im Browser öffnen, fertig. Für den ersten Aufbau und für Testinstanzen ist das der schnellste Weg. Sobald mehrere Menschen mitarbeiten, lohnt sich ein echter Mailversand, weil Einladungen sonst über Copy-and-paste laufen.

## Den ersten Admin einrichten

Es gibt kein vordefiniertes Admin-Konto. Alle registrierten Konten sind zunächst Plattform-USER, und Admin ist eine Rolle, die ich einem echten Account gebe. Das ist richtig so, aber es führt beim ersten Start zur Henne-Ei-Frage.

**Weg A, der vorgesehene.** `MANBAN_BOOTSTRAP_ADMIN_TOKEN` in der `.env` setzen und neu bauen. Dann normal registrieren, E-Mail bestätigen, einloggen. Erst danach `/admin/bootstrap` öffnen und den Token eingeben. Die Reihenfolge ist wichtig: Der Bootstrap stuft das gerade eingeloggte Konto hoch, ohne Login landet der Aufruf auf der Login-Seite. Der Token wirkt nur, solange kein Admin existiert, damit kann sich also niemand aussperren. Danach fliegt er wieder aus der `.env`.

**Weg B, der schnelle.** Registrieren und danach direkt in der Datenbank freischalten:

```
docker compose exec -T postgres psql -U manban -d manban \
  -c "UPDATE app_user SET email_verified = true, platform_role =
  'ADMIN' WHERE email = 'DEINE@MAIL';"
```

Danach ab- und wieder anmelden, denn das Frontend lädt die Rolle nur beim Login.

## **Betrieb unter einer eigenen Domain**

Für den öffentlichen Betrieb läuft der Stack bei mir hinter Traefik. Das ist mein Setup, Traefik muss es nicht sein. Die dafür nötigen Schritte stehen in der Online-Doku unter “Produktions-Deployment”, weil sie stark von der Hosting-Umgebung abhängen.

Egal wo der Stack läuft: Anhänge liegen in MinIO, die Daten in Postgres. Beides gehört in die Sicherung. Ein Board, das den Status des Projekts trägt, ist kein Wegwerf-Dienst.

## KAPITEL 15

# claude-workflow-kit installieren

Das `claude-workflow-kit` wird mit einem Node-Skript installiert, ohne Hintergrundprozess, ohne Service, ohne Registry-Einträge. Was danach im Repo liegt, sind Textdateien und ein Adapter.

## Voraussetzungen

Node ab Version 18, git, eine aktuelle Version von Claude Code. Dazu je nach gewähltem Ziel die passende CLI: `gh` für GitHub, `glab` für GitLab, beide einmalig per `auth login` authentifiziert. Im lokalen Modus brauche ich davon gar nichts.

Wer GitHub oder GitLab als Issue-Tracker benutzt, braucht außerdem ein Projekt-Board mit fünf Spalten: Backlog, Ready, In progress, In review, Done. Bei GitHub sind das Projekt-Board-Spalten, bei GitLab Labels. Die Namen sind case-sensitiv, was mich schon einige Fehlersuchen gekostet hat.

## Installieren

Im Projektordner:

```
npx claude-workflow-kit
```

Alternativ das Skript herunterladen und starten:

```
curl -O https://docs.mwolff.org/install.mjs  
node install.mjs
```

Der Installer fragt der Reihe nach: global oder projektlokal, Code-Host, Issue-Tracker, Name des main-Branch, Name des production-Branch, Review-Umfang und Review-Modell. Bei globaler Installation kommt eine Frage nach dem Vault-Pfad dazu, die leer bleiben darf.

Die erste Frage ist die einzige, bei der ich eine klare Empfehlung habe. **Projektlokal** legt die Skills in `.claude/skills/` ab und schreibt eine `.gignore`, die das Verzeichnis ausnimmt und genau eine Datei per Negation zurückholt:

```
.claude/*  
!.claude/workflow.config.json
```

Die Skills bleiben damit auf dem eigenen Rechner, die Config liegt im Repo. Das ist die richtige Aufteilung, weil in der Config die Qualitätschecks stehen, und die müssen für das ganze Projekt gelten und nicht pro Person. Wer klonet, bekommt dieselben Build-Kommandos, dieselben Schwellen, dieselben Branch-Namen. Global nehme ich, wenn ich über viele Projekte hinweg für mich allein arbeite.

Danach Claude Code neu starten. Die Skills werden beim Start geladen; sichtbar werden sie in der Befehlsliste, die aufklappt, sobald ich `/` eintippe. Der vergessene Neustart ist mit Abstand die häufigste Stolperstelle nach der Installation.

## Die Config

Alles Projekt-Spezifische steht in `.claude/workflow.config.json`. Die Skills selbst sind projekt-unabhängig geschrieben und lesen aus dieser Datei. Ein Update an einem Skill gilt damit in allen Projekten, ohne dass ich in jedem Repo etwas anpassen muss.

```
{
  "codeHost": "github",
  "issueTracker": "github",
  "buildChecks": ["<build>", "<test>"],
  "mutationCommand": "",
  "mainBranch": "main",
  "productionBranch": "production",
  "reviewScope": "diff",
  "reviewModel": "<modell-id>",
  "triggers": { "ready": "ready", "push": "push main", "merge":
"merge production" }
}
```

Zwei Felder erkläre ich genauer.

**buildChecks** ist die Liste der Kommandos, die `/local-check` nacheinander ausführt. Alle müssen grün sein, bevor der Skill Vollzug meldet. In diesem Feld verankere ich die Qualitätssicherung im Prozess, und Kapitel 20 handelt von nichts anderem. Wer es leer lässt, hat den Prozess ohne Gate installiert.

**mutationCommand** steht bewusst außerhalb von `buildChecks`, weil Mutation Testing deutlich länger läuft und ich es manchmal separat ziehen will. Ein leerer String deaktiviert es.

| Stack       | buildChecks                                      | mutationCommand                                             |
|-------------|--------------------------------------------------|-------------------------------------------------------------|
| Java, Maven | <code>["mvn verify"]</code>                      | <code>"mvn org.pitest:pitest-maven:mutationCoverage"</code> |
| Node, npm   | <code>["npm test", "npm run build"]</code>       | <code>""</code>                                             |
| Python      | <code>["pytest", "python -m build"]</code>       | <code>""</code>                                             |
| Go          | <code>["go test ./...", "go build ./..."]</code> | <code>""</code>                                             |

**reviewScope** steuert, ob das Review nur den Diff oder alle Dateien sieht. Für kleine Änderungen reicht `diff`, für größere Refactorings ist `full`

aussagekräftiger, kann aber bei sehr großen Repos das Kontextfenster überlasten. `reviewModel` pinnt das Modell über Sessiongrenzen hinweg, damit das Review nicht davon abhängt, was gerade voreingestellt ist.

Die `triggers` halten die natürlichsprachlichen Phrasen für Ready, Push und Merge. Sie lassen sich ändern, und ich rate dazu, sie im Team einmal bewusst festzulegen und dann nicht mehr anzufassen. Eine Trigger-Phrase, über die diskutiert wird, verliert ihre Funktion.

## Die Config im Team

Die `workflow.config.json` ist eingecheckt und gilt für alle. Das ist ihr Zweck, und genau das stört, sobald zwei Leute verschiedene Modelle benutzen oder verschiedene Phrasen tippen wollen.

Dafür gibt es eine zweite Datei, `.claude/workflow.config.local.json`. Sie ist gitignored und überschreibt die geteilte Fassung, aber nur in vier Feldern: `reviewModel`, `reviewScope`, `triggers` und `toolbox.tokenFile`. Alles andere darin wird ignoriert, mit einer Meldung auf stderr, damit niemand rätselt, warum sein Eintrag nichts bewirkt.

Die vier hängen von der Person ab und nicht vom Projekt: welches Modell jemand fürs Review benutzt, wie viel es zu sehen bekommt, welche Worte diese Person tippt und wo ihr Token liegt. `buildChecks` steht bewusst nicht dabei. Lokal überschreibbar hieße, jeder konfiguriert sich sein Gate weg, und dann ist die Qualitätssicherung eine Meinung.

## Aktualisieren und in weitere Projekte übernehmen

Ich aktualisiere das Kit, indem ich den Installer erneut laufen lasse. Das geht, weil die Skills projekt-unabhängig sind und nur die Config projekt-lokal ist. Sie bleibt erhalten, der Installer fragt vor dem Überschreiben.

In einem neuen Projekt genügt es, den Installer auszuführen oder die `workflow.config.json` aus einem bestehenden Projekt zu kopieren und die Branch-Namen anzupassen.

Adapter und Runner sind Kopien im Projekt und altern dort. Auf welchem Stand ein Repo ist, frage ich so ab:

```
node .claude/kit/board.mjs --version
node .claude/kit/night.mjs --version
```

Der Nacht-Runner warnt von sich aus, wenn er merkt, dass sein Stand hinter dem des Adapters liegt. Das klingt nach Kleinigkeit. Es ist die häufigste Ursache dafür, dass ein Feature “bei mir nicht geht”, das anderswo längst läuft.

## Was der Installer nicht tut

Er schreibt den `issueReview`-Block nicht. Stattdessen legt er eine Vorlage zum Abschreiben daneben, `.claude/workflow.config.example.json`, die er bei jeder Installation auffrischt, während die echte Config unberührt bleibt. Der Grund: Welche Prüfer es gibt, hängt an den CLIs auf der Maschine, und wer wen prüft, ist eine Entscheidung und keine Voreinstellung. Ein automatisch geschriebener Block, von dem ein Prüfer fehlt, würde jeden Lauf rot melden.

Er legt keine projektlokale `kontext.config.json` an. Die ist personenbezogen, weil der Vault-Pfad bei jedem anders liegt. Ich schreibe sie von Hand, Kapitel 17 beschreibt sie. Nur bei globaler Installation schreibt er die Fassung in `~/.claude/`, sofern ich die Frage nach dem Vault-Pfad beantwortet habe. Er richtet keine Qualitäts-Gates ein, denn welche Checks sinnvoll sind, hängt vom Stack ab. Und er baut kein Board, außer bei GitLab, wo er auf Nachfrage die fünf Status-Labels anlegt.

## KAPITEL 16

# Die Skills im Detail

Jeder Skill hat genau eine Aufgabe und ausdrückliche Grenzen. Ein scharf geschnittener Skill sagt, was in seinem Schritt passieren soll, und genauso, was dort nicht passieren darf. Das ist die Konstruktionsregel, und daran hängt, dass der Prozess hält.

Ich gehe die Skills der Reihe nach durch. Claude ruft sie auch von sich aus auf, wenn der Kontext passt. Ich rufe sie trotzdem lieber selbst auf, das ist die sichere Seite.

## Session-Rahmen

**/kontext** lädt zum Session-Start den Lageüberblick. Er liest `kontext.config.json`, zuerst global, dann lokal, wobei lokale Werte gewinnen. Mit Vault lädt er die `always`-Dateien, erkennt die Projektnotiz am Repo-Namen und liest zusätzliche `projectDocs`. Ohne Vault holt er die offenen Issues über den Adapter und liest die Konventionsdateien aus dem Repo. Ausgabe ist ein kurzer Lagebericht: offene Issues, letzte Entscheidungen, was als Nächstes ansteht.

**/document** schreibt zum Session-Ende einen Tageslog-Eintrag und aktualisiert die Projektnotiz. Mit Vault an die Stelle, die `logPath` angibt, standardmäßig `{vault}/Log/YYYY-MM-DD.md`, ohne Vault nach `docs/session-log/YYYY-MM-DD.md` im Projekt. Damit schreibe ich die Dokumentation als letzten Handgriff der Arbeitseinheit und nicht als Pflicht hinterher.

**/retro** ist der Wartungsschritt für den Prozess selbst, alle ein bis zwei Wochen. Kapitel 23 beschreibt ihn.

## Planung

**/fachplan** (ohne Nummer, optional) macht aus einer rohen Anforderung genau ein fachliches Issue mit `[Fachlich]`-Präfix, technikfrei, im Story-

Format. *Grenze:* Er erstellt keinen technischen Plan und keine technischen Issues.

`/plan` (Schritt 2) erzeugt aus der Anforderung einen Plan: Ziel und Nutzerwirkung, betroffene Bereiche und Dateien, architektonische Entscheidungen mit Begründung, offene Fragen, geplante Verifizierung. Mit `/plan #N` liest er die Anforderung aus einem fachlichen Issue. *Grenze:* Er implementiert nichts und legt keine technischen Issues an. Der Plan ist Diskussionsgrundlage, nicht Freigabe.

Ist der Plan freigegeben, legt `/plan` ihn in Bahn 2 als `[Plan]`-Issue ab. Der Body hat sechs feste Überschriften, immer in dieser Reihenfolge: Ziel, betroffene Bereiche, architektonische Entscheidungen, geplante Änderungen, offene Fragen, Verifizierung. Ein Abschnitt, der leer bleibt, bekommt ausdrücklich ein "Keine." und fällt nicht weg. Ein fehlender Abschnitt und ein leerer Abschnitt sehen gleich aus, meinen aber Verschiedenes.

`/issues` (Schritt 3) schneidet den freigegebenen Plan in kleinteilige Issues mit den vier Abschnitten aus Kapitel 7. Liegt der Plan als `[Plan]`-Issue vor, zerlegt ihn `/issues #N`. Zum Abschluss gibt er pro Issue eine Modell-Empfehlung: ein schnelleres Standardmodell für mechanische Aufgaben, das stärkste verfügbare für Architektur- und Sicherheitslogik, jeweils mit einem Satz Begründung. So entscheide ich vor dem Ready, womit ich jedes Issue umsetze, ohne den Plan noch einmal zu lesen. Im Nachtbetrieb entscheide ich das nicht pro Issue: Dem Nacht-Runner sage ich beim Start, welches Modell er nehmen soll, und dabei bleibt es für den ganzen Lauf.

Welches Modell tatsächlich gearbeitet hat, geht dabei nicht verloren. Der Wert wandert als Umgebungsvariable in jede Session, der Adapter schickt ihn als Header mit, und der Aktivitätsverlauf am Board zeigt ihn an. Die `Autor-Modell:`-Zeile aus Kapitel 7 stammt aus derselben Quelle. Es bleibt trotzdem eine Selbstauskunft, das Board führt sie deshalb als Angabe und nicht als Messung.

## Implementierung

Vier Varianten für denselben Schritt 5. Welche ich nehme, hängt davon ab, wie viel ich sehen will.

| Skill                         | Umfang                              | Wofür                                        |
|-------------------------------|-------------------------------------|----------------------------------------------|
| <code>/implement-ready</code> | Ganze Ready-Spalte in einer Session | Der Normalfall bei der Tagesarbeit           |
| <code>/implement-test</code>  | Nur die Tests zu einem Issue, rot   | Wenn der Testvertrag sichtbar sein soll      |
| <code>/implement-done</code>  | Gegen die roten Tests bis grün      | Fortsetzung von <code>/implement-test</code> |
| <code>/implement-next</code>  | Genau ein Issue, dann Ende          | Baustein des Nachtbetriebs                   |

`/implement-ready` arbeitet die Spalte nach Issue-Nummer sortiert ab. Pro Issue: Karte nach In progress, Issue vollständig lesen, testgetrieben implementieren, lokal committen, Karte nach In review. Ist Ready leer, meldet er Vollzug.

`/implement-next #N` ist ein verbindlicher Auftrag. Der Skill weicht nicht auf ein anderes Issue aus, und steht `#N` nicht mehr in Ready, endet er ergebnislos, statt sich etwas zu suchen. Der Nacht-Runner baut darauf auf.

Drei Grenzen gelten für alle vier: Sie pushen nie, sie ziehen keine Backlog-Issues eigenmächtig nach Ready, und sie starten keine Session für ein Issue mit `[Fachlich]`-, `[Idee]`- oder `[Plan]`-Präfix, sondern stellen es kommentiert zurück ins Backlog.

Das Paar `/implement-test` und `/implement-done` ist die feinkörnige Variante. Der erste schreibt ausschließlich Tests, kein Produktionscode, kein Commit. Läuft bereits ein Issue in In progress, stoppt er und verweist auf den zweiten. Der zweite findet das laufende Issue über die In-progress-Spalte und implementiert, bis die Tests grün sind. Ich empfehle dieses Paar für den Einstieg und für alles, wo der Testvertrag die eigentliche Denkarbeit ist.

## Prüfung

`/issue-review` prüft das Dokument, gegen das implementiert wird, nicht den Code. Er sitzt zwischen Schritt 3 und dem Ready und hat deshalb keine Nummer. Welche der drei Prüfstufen greift, liest er am Titel-Präfix ab: `fachlich`, `plan` oder `issue`. Ohne Argument nimmt er das ganze Backlog, mit Nummern genau die genannten. *Grenze:* Er schreibt den Body nie von sich aus, er zieht nichts nach Ready, und er ersetzt weder `/review` noch `/local-check`. Kapitel 9 beschreibt das Verfahren.

`/local-check` (Schritt 6) führt alle Kommandos aus `buildChecks` nacheinander aus, danach `mutationCommand`, sofern gesetzt. Bei Frontend-Änderungen erinnert er an die manuelle Verifikation im Browser und vermerkt im Bericht, wenn sie nicht möglich war. Neuere Modelle können den Browser selbst steuern, das erleichtert die Prüfung. Die menschliche Abnahme bleibt trotzdem nötig. Ausgabe ist eine Liste mit grünen Häkchen oder einem roten Stopp. Ein roter Check blockiert den weiteren Prozess, ohne Ausnahme.

`/review` (Schritt 7) öffnet eine frische Session ohne den Implementierungskontext, im Modell aus `reviewModel`, und lässt sie je nach `review-Scope` den Diff oder alle Dateien lesen. Die Befunde landen im Issue oder im Pull Request.

`/review` ist ausdrücklich nicht das Sicherheitsnetz für Security. Secrets-Scan, Injection-Muster und fehlende Validierung gehören in deterministische Werkzeuge im CI. Der Skill ergänzt sie. Kapitel 20 begründet das.

## Freigabe

`/push-main` (Schritt 8) pusht den aktuellen Commit-Batch auf den main-Branch. Der Skill ist gegen selbsttätigen Aufruf gesperrt und reagiert nur auf die Trigger-Phrase. Eine frühere Freigabe in derselben Session gilt nicht für neue Commits. Ein roter `/local-check` blockiert ihn mechanisch.

`/merge-production` (Schritt 9) erstellt den Pull Request von main nach production. Auch er ist gegen selbsttätigen Aufruf gesperrt. Den Merge führt der Mensch durch, der auf dem Test-Server geprüft hat.

## Release

Liegt eine `RELEASING.md` im Projekt-Root, lesen `/push-main` und `/merge-production` sie bei jedem Lauf und führen aus, was dort steht, bevor sie pushen oder den Pull Request anlegen. Liegt keine da, entfällt der Schritt ersatzlos.

Das ist keine Kit-Funktion, das ist eine Konvention. Jedes Projekt hat eigene Release-Schritte, Versionsnummer hochzählen, Changelog pflegen, was auch immer, und niemand soll dafür die generischen Skills forken müssen. Eine Datei an einem verabredeten Ort reicht.

Am Ende seines Laufs gibt `/merge-production` die fertige Kommandozeile für den Tag aus, mit dem Hash des Versions-Commits, den er selbst erzeugt hat:

```
git tag -a vX.Y.Z <hash> -m "Release vX.Y.Z" && git push origin  
vX.Y.Z
```

Absetzen muss ich die Zeile selbst. Das ist der vierte Punkt aus Kapitel 3.

## Warum das Ganze als Skills und nicht als eine Datei

Das ist die Geschichte von Kapitel 6: Eine Datei, die am Sessionanfang gelesen wird, verliert im Lauf der Session an Gewicht. Ein Skill wird in dem Moment geladen, in dem sein Schritt dran ist. Das ist der ganze Unterschied, und er ist größer, als er klingt.

## KAPITEL 17

# Persistente Kontext-Dateien und der Vault

Ein Sprachmodell beginnt jede neue Session bei null. Das lässt sich nicht wegprompten, so sind die Modelle gebaut. Ärgern hilft nicht, Vorkehrungen schon.

Ich ziehe also die Erinnerung aus dem Modell heraus und lege sie in Dateien. Diese Dateien sind das Gedächtnis des Teams, nicht das Gedächtnis der KI.

## Fünf Klassen von Dateien im Repo

**Die zentrale Konventionsdatei** im Repo-Root beschreibt die projektspezifischen Regeln. Bei Claude Code heißt sie typischerweise `CLAUDE.md`, bei anderen Werkzeugen anders. Das Konzept ist überall dasselbe, nur der Dateiname wechselt. Ich beschreibe darin, was in diesem Projekt anders ist als in anderen.

**Die Workflow-Konventionsdatei**, im Beispiel `CLAUDE-workflow.md`, beschreibt den Prozess selbst. Ich halte sie bewusst projekt-unabhängig, damit ich sie in andere Projekte kopieren und dort mit wenigen Handgriffen anpassen kann. Sie wird vom Installer angelegt.

**Tech-spezifische Konventionsdateien**, etwa `CLAUDE-java.md` oder `CLAUDE-react.md`, sammeln Regeln pro Sprache und Framework: Testpyramide, Code-Styling, Architektur-Muster.

**Eine Security-Konventionsdatei**, etwa `CLAUDE-security.md`, bündelt die Checks vor Commit und Push. Sie beschreibt, worauf zu achten ist. In Kapitel 20 zeige ich, warum ich gerade hier Werkzeuge außerhalb des Modells brauche.

**Persönliche Memory-Dateien** im Workspace der entwickelnden Person ergänzen das um individuelle Vorlieben, bevorzugte Werkzeuge, Routinen.

Alle diese Dateien pflege ich wie Code. Änderungen daran sind Commits. Sie sind Teil des Reviews. Sie werden mit dem Repo versioniert.

## Der Vault

Der Vault ist ein Speicher außerhalb des Repos. Er hält projektübergreifendes Wissen: Profil, Arbeitsregeln, Entscheidungshistorie, Tages-Logs. Er ist kein persönliches Eigentum, er gehört zum Projekt. `/kontext` lädt ihn zu Session-Beginn, `/document` schreibt am Session-Ende.

Ich benutze dafür Obsidian, aber nur, weil mir die Struktur gefallen hat:

|                               |                                          |
|-------------------------------|------------------------------------------|
| <code>/pfad/zum/vault/</code> |                                          |
| <code>Index.md</code>         | Übersicht, was im Vault liegt            |
| <code>Profil.md</code>        | always-Datei                             |
| <code>Projekte/</code>        |                                          |
| <code>{repo-name}/</code>     |                                          |
| <code>{repo-name}.md</code>   | Projektnotiz, von <code>/document</code> |
| aktualisiert                  |                                          |
| <code>Log/</code>             |                                          |
| <code>YYYY-MM-DD.md</code>    | Tages-Logs                               |

Das steht in `kontext.config.json`. Die projektlokale Fassung legt der Installer bewusst nicht an, denn der Pfad ist bei jedem anders. Global schreibt er sie, wenn ich ihm beim Installieren einen Vault-Pfad genannt habe.

```
{
  "vault": "/pfad/zum/vault",
  "always": ["Index.md", "Profil.md"],
  "projectDocs": ["CLAUDE-*", ".claude/CLAUDE-*"]
}
```

`always` sind Dateien, die immer gelesen werden. `projectDocs` sind Dateien oder Glob-Muster relativ zum Projektverzeichnis; Muster ohne Treffer werden stillschweigend übersprungen. Die Projektnotiz wird automatisch am Repo-Namen erkannt. Weicht der Vault-Ordnername davon ab, setze ich `project` in der lokalen Config.

## Eine Datei, ein Schreiber

Sobald mehr als ein Projekt an demselben Vault hängt, wird `logPath` wichtig. Das Feld bestimmt, wohin `/document` den Tageslog schreibt, und steht per Default auf `Log/{date}.md`. Dann schreiben alle Projekte in dieselbe Datei desselben Tages. Läuft irgendwo eine Synchronisation mit, entstehen Konfliktkopien, und der Log ist die Datei, die das am wenigsten verträgt: Sie wird angehängt, nicht überarbeitet.

Bei mehreren Repos hole ich deshalb das Projekt in den Namen:

```
"logPath": "Log/{date}-{project}.md"
```

Die Regel dahinter gilt über den Log hinaus. Eine Datei, ein Schreiber. Wo zwei Instanzen an dieselbe Stelle schreiben, ohne voneinander zu wissen, ist der Konflikt eine Frage der Zeit.

Die Vault-Struktur legt niemand automatisch an, die baue ich einmal von Hand. Die Projektnotiz pflegt `/document` danach selbst, die übergeordnete `Index.md` nicht: Dort fragt er einmal nach und zeigt dabei den konkreten Eintrag, den er ergänzen würde.

## Warum mehrere Config-Dateien

Das verwirrt am Anfang, ist aber richtig. `workflow.config.json` ist repo-spezifisch: Build-Kommandos, Branch-Namen, Review-Modell. Sie gehört ins Repo, damit alle dieselbe Prozessgrundlage haben. `kontext.config.json` ist personenbezogen, denn jeder checkt das Projekt in ein anderes Verzeichnis aus. Selbst wenn der Vault mit im Repository läge, wäre der Pfad dorthin bei jedem anders, und genau den hält diese Datei. Deshalb gehört sie nicht ins Repo.

Dazu kommt eine dritte, optionale: `workflow.config.local.json` für die wenigen Werte, die im selben Projekt bei jedem anders sein dürfen. Kapitel 15 beschreibt sie. Was für das Projekt gilt, wird geteilt. Was an der Person hängt, bleibt daneben liegen.

## Degraded Mode

Ohne Vault laufen `/kontext` und `/document` weiter. Der erste lädt die offenen Issues über den Adapter und liest die Konventionsdateien aus dem Repo, der zweite schreibt den Log ins Projektverzeichnis. Beide melden am Ende, dass sie ohne persistentes Memory arbeiten.

Das ist der richtige Einstieg. Wer den Prozess ausprobieren will, sollte nicht vorher eine Vault-Infrastruktur aufbauen. Der Vault lohnt sich, sobald mehrere Menschen an einem Projekt arbeiten. Niemand soll seine Erfahrungen für sich behalten. Wissen und Projektfortschritt gehören allen im Team.

## Memory-Drift

Eine Konvention, die vor sechs Monaten richtig war, ist heute vielleicht überholt. Regeln ändern sich vor allem dann, wenn sie öfter zu Fehlern führen. Deswegen gehören sie entweder geschärft oder abgeschafft.

Wer Memory-Dateien nur ergänzt, ohne sie zu pflegen, baut eine Schuldenlast auf. Das Ergebnis sind alte Regeln, neue Realität und wachsende Reibung. In der KI-Retrospektive aus Kapitel 23 prüfe ich genau das, und dafür gibt es sie als eigenes Format.

Dieser Aufwand liegt beim Menschen, und ich will das nicht beschönigen. Die Zeit, die das Modell beim Implementieren spart, fließt zu einem Teil in die Pflege des Kontexts zurück. Das ist einer der Gründe, warum viele, die mit KI entwickeln, meinen, es gehe nicht schneller. Das ist tatsächlich so: Nur die Implementierung geht schneller, und mit den richtigen Leitplanken wird sie besser.

## KAPITEL 18

# GitHub, GitLab oder lokal

Die Skills wissen nicht, gegen welche Plattform sie arbeiten. Alle Issue- und Board-Operationen laufen über `.claude/kit/board.mjs`. Der Adapter kennt eine Schnittstelle, und die ist bewusst klein: Issue anlegen, auflisten, lesen, verschieben, kommentieren, dazu ein paar Kommandos für Pull Requests und für das Issue-Review. Anhang A listet sie vollständig.

Weil die Schnittstelle so schmal ist, ist ein Plattformwechsel eine Zeile in der Konfiguration und kein neues Projekt.

## Zwei unabhängige Achsen

`codeHost` bestimmt, wo Repository und Pull Requests liegen. `issueTracker` bestimmt, wo Issues angelegt und Board-Karten bewegt werden. Beide dürfen auf verschiedene Ziele zeigen.

Am Anfang habe ich beides zusammen behandelt, Repository und Issue-Tracker. Erst später habe ich gemerkt, dass sich die beiden sehr gut trennen lassen.

Am nützlichsten finde ich `codeHost: github` mit `issueTracker: local`. Der Code liegt, wo das Team ihn erwartet, die Issues liegen als Dateien im Repo und laufen durch dieselben Reviews wie der Code.

## Die vier Ziele

| Ziel                    | Voraussetzung                        | Wie der Status abgebildet wird                      |
|-------------------------|--------------------------------------|-----------------------------------------------------|
| <code>github</code>     | <code>gh</code> , authentifiziert    | Spalten in GitHub Projects                          |
| <code>gitlab</code>     | <code>glab</code> , authentifiziert  | Labels <code>~Backlog</code> bis <code>~Done</code> |
| <code>local</code>      | nichts                               | YAML-Frontmatter in <code>issues/</code>            |
| <code>kanban-kit</code> | Instanz, Token, <code>tbx.mjs</code> | Spalten im Board                                    |

## Was bei GitLab anders ist

Der Pull Request heißt Merge Request, das Ergebnis ist dasselbe.

Der Board-Status läuft über Labels statt über Board-Karten. GitHub Projects hat eine API, über die das Kit die Spalte direkt setzt, GitLab bietet das per CLI noch nicht vollständig. Die fünf Spalten sind deshalb fünf Labels. Der Weg durch die Status bleibt derselbe. Er ist nur nicht in der Spalte zu sehen. Er steht als Label an der Karte.

Der Installer legt diese fünf Labels auf Nachfrage automatisch an. Das Repository muss dafür existieren, und ich muss im geklonten Verzeichnis stehen, damit `glab` das richtige Projekt erkennt. Die Board-Spalten in der GitLab-Oberfläche lege ich einmal von Hand an.

Wer den Nachtbetrieb benutzt, braucht dort ein Routing-Label. Weil bei GitLab Labels bereits der Status-Mechanismus sind, muss dieser Name kollisionsfrei sein. Der Default `kit:nightrun` trägt dafür ein Namespace-Präfix.

## Der lokale Modus

Mit `issueTracker: local` legt der Adapter Issues als Markdown-Dateien mit YAML-Frontmatter in `issues/` an. Es gibt kein Board, keine externe CLI, keine Anmeldung, keine Netzwerkabhängigkeit.

Ich halte diesen Modus für unterschätzt. Er kann drei Dinge, die kein gehostetes Board kann. Die Issues sind versioniert, ich sehe also, wann jemand eine Anforderung geändert hat und wer. Sie sind Teil des Diffs und laufen durch dasselbe Review wie der Code. Und sie funktionieren ohne alles, auch im Zug ohne Netz.

Die Übersicht fehlt. Der Status steht im Frontmatter der Issue-Datei. Wer ein Issue von Backlog nach Ready holt, öffnet die Datei, ändert eine Zeile, speichert. Bei einem Issue ist das nichts, bei zwanzig ist es eine Fleißarbeit, und sie steht ausgerechnet vor dem Schritt, der den Prozess in Gang setzt.

Dafür gibt es ein kleines Werkzeug. `board-ui.mjs` liegt auf `docs.mwolff.org` zum Herunterladen, kommt nach `.claude/kit/` und wird aus dem Wurzelverzeichnis des Projekts aufgerufen:

```
node .claude/kit/board-ui.mjs
```

Es zeigt die fünf Spalten, ich ziehe ein Issue von Backlog nach Ready, und danach sehe ich zu, wie das Kit die Datei selbst nach In progress und nach In review schiebt. Den Rest erklärt `--help`. Mehr kann es nicht.

Sobald mehrere Menschen den Stand sehen wollen, reicht das nicht mehr. Dann kommt ein richtiges Board dazu, und ob es GitHub Projects, GitLab oder das kanban-kit wird, ist Geschmackssache.

## Wechseln

`issueTracker` und `codeHost` lassen sich jederzeit in der Config ändern, alle Skills passen sich beim nächsten Aufruf an. Ich habe das öfter gemacht, von GitHub zum lokalen Modus gewechselt und umgekehrt. Das kostet mich nichts, weil die konkreten Details hinter dem Adapter liegen. Die vorhandenen Issues wandern dabei nicht mit. Der Adapter übersetzt Operationen, er transportiert keine Daten.

Wer wechseln will, hat drei Möglichkeiten: die laufenden Issues im alten Tracker abschließen und nur neue im neuen anlegen, von Hand migrieren, oder sich ein Werkzeug dafür schreiben. Ich habe beim vorletzten Wechsel die erste Variante gewählt, weil ein halb migrierter Stand schlimmer ist als zwei getrennte Zeiträume.

Beim letzten habe ich mich anders entschieden und ein Einmalwerkzeug dafür geschrieben. Es hat drei Läufe: exportieren aus dem alten Tracker, importieren ins neue, und einen dritten, der beide Seiten vergleicht und erst dann grün meldet. Der Vergleichslauf ist der wichtigste, denn ohne ihn hätte ich genau den halb migrierten Stand, vor dem ich mich gefürchtet habe.

Drei Entscheidungen haben den Umzug getragen. Ich migriere nur die offenen Issues, und sie behalten ihre Originalnummern, damit alte

Commit-Botschaften mit `#N` nicht ins Leere zeigen. Der alte Tracker bleibt eingeschaltet und wird zum Archiv, ich lösche ihn nicht. Und jede migrierte Karte trägt sichtbar im Body, woher sie kommt und in welcher Spalte sie vorher lag. Ohne diese Kopfzeile sähen Karten aus einer Spalte, die es im neuen System nicht gibt, im Backlog aus wie normale Arbeit.

Der Import muss dabei mehrfach ausführbar sein, ohne Dubletten zu erzeugen. Ein Umzug läuft selten beim ersten Versuch durch.

## Warum kein Multi-Tool-Adapter

Das Kit bringt die Skills im Claude-Code-Format mit. Das Konzept ist auf andere Werkzeuge übertragbar, das Format nicht: Codex liest `AGENTS.md`, Cursor liest `.cursor/rules`.

Wer mehrere Engines parallel einsetzen will, braucht die Skill-Bibliothek in allen passenden Formaten im Repo. Das ist machbar, im Kit steckt es nicht. Der Grund ist derselbe wie bei Jira: Ich habe es nicht gebraucht. Fairerweise: Das ist eine typische Aufgabe, die ein Sprachmodell wunderbar und akkurat löst.

## KAPITEL 19

# Nachts arbeiten lassen: der Nacht-Runner

Warum der Nachtbetrieb bei mir der Normalfall ist und wie sich ein Tag darum herum anfühlt, steht in Kapitel 10. Hier steht, wie der Runner bedient wird.

Der Nacht-Runner ist ein Node-Skript und kommt mit dem Installer. Je nach Installation liegt er als `.claude/kit/night.mjs` im Projekt oder global als `~/.claude/kit/night.mjs`. Teil des Repos ist er in keinem der beiden Fälle, dafür sorgt die `.gitignore` aus Kapitel 15. Er startet pro Issue eine Headless-Session mit `/implement-next`, wartet auf ihr Ende und prüft den Erfolg ausschließlich am Board: Issue in In review heißt Erfolg.

Das ist sein erster Modus. Er hat einen zweiten, der prüft statt baut. Bis der weiter unten drankommt, ist immer der erste gemeint.

## Voraussetzungen

Vier Dinge müssen stimmen, sonst lohnt der Start nicht.

**Die Issues sind eindeutig.** Tagsüber lässt sich eine Unklarheit im Gespräch auflösen, nachts wird sie geraten. Das Schnittkriterium aus Kapitel 7 gilt hier verschärft.

**Die Abhängigkeiten stehen als #N im Abhängigkeiten-Abschnitt.** Ob das Fundament trägt, kann der Runner nur entscheiden, wenn ich diesen Abschnitt ordentlich pflege.

**Der Workspace ist leer.** Es darf keine Commits aus früheren Arbeiten mehr geben, und es darf kein Issue in In progress sein.

**buildChecks ist gefüllt.** Ohne Gate verweigert der Runner den Start, und das ist richtig so. Wer es trotzdem braucht, muss `--no-checks-ok` ausdrücklich setzen.

Die letzten beiden Punkte gelten für den Implementierungs-Modus. Die Review-Nacht baut nichts und committet nichts, dort entfallen sie.

## Starten

```
node .claude/kit/night.mjs --dry-run # zeigt an, was liefere,  
startet nichts  
node .claude/kit/night.mjs          # echter Lauf
```

Ich starte grundsätzlich erst mit `--dry-run`. Der Lauf weist aus, welche Issues er anfassen würde, welche er wegen fehlender Labels überspringt und welche er wegen offener Abhängigkeiten zurückstellt. Zehn Sekunden, die mehrere Nächte gerettet haben.

| Flag                                 | Wirkung                                                 | Default                   |
|--------------------------------------|---------------------------------------------------------|---------------------------|
| <code>--dry-run</code>               | Nur anzeigen, nichts starten                            | aus                       |
| <code>--max &lt;N&gt;</code>         | Session-Limit pro Nacht                                 | 10                        |
| <code>--model &lt;id&gt;</code>      | Modell für den Lauf                                     | stärkstes verfügbares     |
| <code>--timeout-min &lt;N&gt;</code> | Zeitlimit pro Runde                                     | 60                        |
| <code>--label &lt;name&gt;</code>    | Routing-Label, <code>none</code> schaltet den Filter ab | <code>kit:nightrun</code> |
| <code>--verbose</code>               | Live-Verlaufsprotokoll                                  | aus                       |
| <code>--no-checks-ok</code>          | Start trotz leerer <code>buildChecks</code>             | aus                       |
| <code>--yolo</code>                  | Alle Freigaben übergehen, Einzelfall                    | aus                       |

Ohne `--verbose` protokolliert der Runner pro Runde nur Start und Ende. Bei einer langen Session sehe ich dann nicht, woran sie gerade arbeitet, weil die Zusammenfassung erst hinterher kommt. Mit `--verbose` liest er den Stream mit und schreibt kompakte Ereigniszeilen mit Issue-Nummer ins Log:

```
[18:24:10] #401 > Bash: mvn -q verify
[18:25:02] #401 > Edit: src/main/java/.../
ProjectIdeaEventService.java
[18:26:11] #401 > Claude: Tests grün, ich committe jetzt.
```

## Das Routing-Label

Standardmäßig verarbeitet der Runner aus Ready nur Issues mit dem Label `kit:nightrun`. Alle anderen bleiben unangetastet liegen, ohne Verschieben, ohne Kommentar.

Ich will auf einem einzigen Board markieren können, was nachts laufen darf, und den Rest für die interaktive Arbeit am Tag behalten. Die Alternative wäre ein zweites Board mit eigenem Token, und damit wären `#N`-Abhängigkeiten zwischen den Boards unauflösbar.

Bei GitLab sind Labels bereits der Status-Mechanismus. Der Routing-Label-Name muss dort kollisionsfrei sein. Der Default trägt dafür ein Namespace-Präfix.

Wer den Runner früher ohne Labels benutzt hat, muss seine Nacht-Issues jetzt markieren oder `--label none` setzen, sonst findet der Lauf nichts.

## Die Leitplanken

| Situation                       | Verhalten                                             |
|---------------------------------|-------------------------------------------------------|
| Immer                           | Kein Push. Der Runner baut lokale Commits, mehr nicht |
| <code>buildChecks</code> leer   | Kein Start, außer mit <code>--no-checks-ok</code>     |
| Checks rot, Worktree sauber     | Issue zurück ins Backlog, nicht nach In review        |
| Worktree am Rundenende unsauber | Erst ein Rettungsversuch, dann harter Stopp des Laufs |

| Situation                                                           | Verhalten                                    |
|---------------------------------------------------------------------|----------------------------------------------|
| Issue hängt beim Start in In progress                               | Runner stoppt, statt zu raten                |
| Unerfüllte #N-Abhängigkeit                                          | Issue wird zurückgestellt                    |
| Titel mit [Fachlich], [Idee] oder [Plan]                            | Kommentiert zurück ins Backlog, ohne Session |
| Weder Issue-Review: noch Prüfung: Verzicht, bei requiredBeforeReady | Kommentiert zurück ins Backlog, ohne Session |
| Session-Limit erreicht                                              | Lauf endet, Rest bleibt in Ready             |

Die Präfix-Zeile sichert die PO-Schleife aus Kapitel 11 mechanisch ab. Ready heißt implementierbar, und keine der drei Sorten ist es. Der gefährlichste Fall ist das Plandokument: Eine Session, die es als Auftrag liest, baut den Plan nach und meldet Vollzug.

Die Zeile darunter ist das Gate aus Kapitel 9. Nachts wird ein ungeprüftes Issue hart zurückgestellt, tagsüber weisen die Skills nur darauf hin und fragen. Nachts antwortet ja niemand.

Freigeben können zwei Dinge: der Marker aus einem gelaufenen Review, oder mein ausdrücklicher Verzicht. Sich selbst freigeben kann der Runner nicht. Eine Zeile, die die Prüfung verringert, weist der Adapter im unbeaufsichtigten Lauf ab.

## Wenn eine Runde schiefgeht

Drei Fälle, und der Runner behandelt sie verschieden.

**Die Umgebung ist kaputt.** Die Session endet mit einem Fehlercode, weil die Anmeldung abgelaufen ist oder das CLI nicht startet. Dann stoppt der Lauf hart, und das Issue bleibt unangetastet in Ready. Mit ihm ist ja nichts falsch. So räumt eine kaputte Umgebung nicht die ganze Ready-Spalte leer.

**Die Arbeit ist gescheitert.** Die Session endet sauber, aber das Issue steht nicht in In review. Der Runner kommentiert und stellt es zurück ins Backlog, der Lauf geht mit dem nächsten Issue weiter.

**Das Zeitlimit ist gerissen.** Der Runner behandelt das wie einen fachlichen Fehlschlag, denn es bedeutet dasselbe: Die Aufgabe war zu groß geschnitten.

Bevor der Runner bei einem unsauberen Worktree hart stoppt, sieht er nach, ob die Arbeit vielleicht doch fertig ist. Warum das nötig wurde, steht in Kapitel 10.

Der Runner fährt dafür die `buildChecks` selbst. Für diese Vorprüfung mischt er den `env`-Block aus `.claude/settings.json` und `.claude/settings.local.json` in seine Umgebung, sonst fehlen ihm projektspezifische Variablen und er meldet ein falsches Rot. Sind die Checks grün, bekommt genau eine Rettungs-Session pro Issue die Aufgabe, den Stand gegen das Issue zu prüfen, zu committen und das Board zu bewegen. Zehn Minuten, keine Builds mehr. Scheitert sie, oder waren die Checks rot, bleibt es beim harten Stopp, jeweils mit eigener Zeile im Log.

**Ein Formatverstoß darf keinen Lauf kippen.** Ein einzelner falsch umbrochener Javadoc-Kommentar hat mir einmal einen kompletten Nachtlauf beendet, obwohl die Arbeit korrekt war. Dafür gibt es `formatFixCommand`:

```
"formatFixCommand": "mvn spotless:apply"
```

Sind die Checks in der Vorprüfung rot, läuft das Kommando genau einmal, danach wiederholt der Runner die Checks. Sind sie dann grün, geht der Lauf weiter, und das Protokoll weist den Eingriff mit `FORMAT-FIX angewendet` aus, er passiert also nie still. Bleiben sie rot, war das Format nicht die Ursache, und es bleibt beim harten Stopp. Ein Formatverstoß löst sich deterministisch auf, ein fehlgeschlagener Test nicht.

`buildChecks` und `formatFixCommand` sind frei geschriebene Kommandozeilen und brauchen deshalb eine Shell, anders als die festen Kommandos des Board-Adapters. Der Runner startet sie unter macOS und Linux in `/bin/sh`, unter Windows in der ComSpec-Shell, im Regelfall `cmd.exe`. Bewusst nicht PowerShell: Der Wert ist Nutzer-Konfiguration, und wer unter Windows ein Build-Kommando einträgt, denkt an `cmd.exe`. Pow-

erShell hat eine eigene Operator-Syntax, `&&` gibt es dort erst ab Version 7.

Auf gemischten Rechnern wird das zum Problem. `mvn verify` und `npm test` laufen überall, aber `&&`-Ketten, Pipes und `2>/dev/null` verhalten sich unter `cmd.exe` anders oder gar nicht. Für Nachtläufe halte ich die Kommandos deshalb einfach.

## Der zweite Modus: die Review-Nacht

Derselbe Runner kann prüfen statt bauen. `--review` nimmt Dokumente aus dem Backlog und lässt sie von `/issue-review` durch fremde Modelle lesen.

```
node .claude/kit/night.mjs --review --dry-run
node .claude/kit/night.mjs --review
```

Morgens liegen die Befunde am Board: geschärfte Vorschläge als Kommentar, und die unauffälligen Dokumente tragen bereits den Prüf-Marker ihrer Stufe.

| Flag                                               | Wirkung                                                 | Default                      |
|----------------------------------------------------|---------------------------------------------------------|------------------------------|
| <code>--review</code>                              | Prüfen statt implementieren                             | aus                          |
| <code>--review-label &lt;name&gt;</code>           | Routing-Label, <code>none</code> schaltet den Filter ab | <code>kit:nightreview</code> |
| <code>--stufe &lt;fachlich, plan, issue&gt;</code> | Welche Dokumente genommen werden                        | <code>issue</code>           |

Ein Aufruf fährt genau eine Stufe. `fachlich` nimmt die `[Fachlich]`-Issues, `plan` die `[Plan]`-Issues, `issue` weder das eine noch das andere. `[Idee]` bleibt in jeder Stufe draußen. Zwischen den Stufen steht die menschliche Freigabe, deshalb gibt es keinen Lauf, der alle drei hintereinander abarbeitet.

Das Zeitlimit liegt fest bei fünfzehn Minuten pro Issue, `--timeout-min` gilt hier nicht. Das Maß für eine ganze Implementierungsrunde wäre

hier die falsche Größe. `--review` ist exklusiv, die Implementierungsschleife läuft dann nicht, und die `buildChecks` -Pflicht entfällt.

**Warum aus dem Backlog und nicht aus Ready.** Der Ablauf ist zweigeteilt: Review-Nacht, morgens sichten und nach Ready ziehen, Implementierungs-Nacht. Zwei Nächte statt einer. Zwischen Review und Implementierung liegt das Ready, und das Ready ist menschlich. Würde der Runner ein Ready-Issue erst prüfen und dann bauen, hätte ich mein Ready auf einen Text gegeben, der bei der Implementierung nicht mehr gilt. Ich hätte die Schwelle umgangen, ohne es zu merken.

Deshalb auch ein eigenes Routing-Label. `kit:nightreview` markiert Backlog-Issues zur Prüfung, `kit:nightrun` markiert Ready-Issues zur Umsetzung. Die beiden Modi meinen verschiedene Spalten.

**Der Vorflug.** Vor der ersten Review-Session startet der Runner genau eine Prüfsession, die nachsieht, ob alle konfigurierten Reviewer wirklich laufen und der Tracker erreichbar ist. Fehlt etwas, stoppt der Lauf, bevor die erste Prüfung beginnt. Abschalten lässt sich das nicht. Wer nur wissen will, ob alles steht, fährt `--dry-run`, der meldet den Befund, ohne abzuberechnen.

Interaktiv fragt der Skill nach, wenn ein Reviewer fehlt. Nachts steht niemand daneben, und ein Lauf mit einem Reviewer statt zweien sieht am Board aus wie ein vollständiger.

Der Vorflug läuft in einer eigenen Session, nicht im Runner. Das klingt nach Umstand. Laufen muss der Reviewer aber in den Review-Sessions, und die sind eigene Kindprozesse, jeder mit seiner Sandbox, seiner Netzwerk-Allowlist und seinen Freigaben. Ein Probelauf im Runner beweist nur, dass der Runner das Werkzeug starten darf.

In der Nacht vom 8. August lief genau das schief. Der Probelauf im Runner war sauber, während `codex exec` in der Session an "Run outside of the sandbox" scheiterte und `board.mjs issue get` an einer leeren Netzwerk-Allowlist. Ein Lauf, der vollbesetzt startete und mit einem Reviewer arbeitete. Seitdem nennt jeder Befund die Umgebung, in der er entstanden ist, und die Erreichbarkeit des Trackers ist ein eigener Befund. Bei Issue #248 scheiterte nicht der Reviewer, sondern das `issue`

get). Wer das als Reviewer-Ausfall meldet, schickt mich morgens in die falsche Ecke.

Pro Issue gibt es drei Ausgänge:

| Was die Session hinterlässt        | Was es heißt                                                    |
|------------------------------------|-----------------------------------------------------------------|
| Marker im Body                     | geprüft, nichts Gewichtiges gefunden                            |
| Kein Marker, Befunde als Kommentar | geprüft, mit Befund. Wartet auf mich                            |
| Nichts                             | ohne Ergebnis. Das Issue wird kommentiert, der Lauf geht weiter |

Der mittlere Fall ist ein Erfolg. Es sind genau die Issues, bei denen sich der Review gelohnt hat. Kein Issue wird in diesem Modus bewegt, die Kandidaten liegen ja schon im Backlog. Hinterlässt eine Review-Session Änderungen im Working Tree, stoppt der Lauf hart. Sie hat dort nichts zu suchen.

**Was die Nacht darf und was nicht.** Sie schreibt nie den Issue-Body, auch nicht, wenn der Review nichts gefunden hat. Der geschärfte Vorschlag geht als Kommentar ans Issue, als Text zum Übernehmen. Beim Groomen lese ich ihn von dort.

Den Marker setzt sie, wenn nichts zu ändern ist: kein Fund mit Schweregrad **BLOCKER** oder **WICHTIG**, kein ausgefallener Reviewer, keine Unterbesetzung. Ein einziger gewichtiger Fund reicht, und der Marker bleibt aus.

Die Verantwortungsschwelle liegt beim Ändern der Anforderung, nicht beim Feststellen, dass nichts zu ändern ist. Den Marker zu setzen ist eine Protokollhandlung. Das Ready bleibt vollständig meins, nach Ready zieht weiterhin nur der Mensch.

In eine fachliche Anforderung und in einen Plan schreibt die Nacht nie. Dort stehen PO-Antworten und Architekturentscheidungen, die ein Mensch getroffen hat.

Ein nächtlich gesetzter Marker ist als solcher erkennbar:

Issue-Review: codex (2026-08-06, Nachtlauf)

## Ein eigenes Board für die Nacht

Läuft das Projekt gegen ein kanban-kit, lässt sich der ganze Nachtlauf auf ein separates Night-Board umschalten. Dafür in der Admin-Oberfläche ein zweites Token erzeugen und an das Night-Board binden, als gitignorete Datei ablegen und den Runner mit `TBX_TOKEN` starten:

```
TBX_TOKEN="$(cat .claude/tbx-night.token)" caffeinate -i
node .claude/kit/night.mjs
```

Das `caffeinate -i` auf macOS ist kein Zierrat. Ein Rechner, der um halb zwei in den Ruhezustand geht, beendet den Lauf mitten im Issue.

Ein Klartext-Token gehört nicht in die `workflow.config.json`, denn die ist eingechekkt und wird geteilt. Der Adapter bricht ab, wenn er dort eines findet, statt das Secret still zu benutzen.

`TBX_TOKEN` wird dem Kommando vorangestellt und nicht in der `.zshrc` exportiert. Exportiert gewinnt es in jedem Projekt gegen dessen `tokenFile`, und dann arbeitet irgendwann ein Nachtlauf gegen ein Board, das er nicht meinen sollte. Vorangestellt gilt es für diesen Aufruf und für die Sessions, die er startet, weil der Runner seine Umgebung an jeden Kindprozess weiterreicht.

Was dabei bewusst nicht geht: aus einem Board ziehen und an ein anderes melden. Das Token gilt für den ganzen Lauf, Lesen und Schreiben gehen an dieselbe Stelle. Ein Runner, der die eine Hälfte woanders hinschreibt als die andere, hinterlässt einen Zustand, den morgens niemand mehr auflösen kann.

## Berechtigungen

Ein Punkt, der beim ersten Einrichten Zeit kostet. Eine unbeaufsichtigte Session kann nicht zurückfragen, ob sie ein Kommando ausführen darf. Was nicht vorab erlaubt ist, wird sofort abgelehnt. Die Session imple-

mentiert dann weiter, kann ihre Checks aber nicht ausführen und committet nicht. Die Runde endet ohne Ergebnis, und zwar zügig, nicht erst am Zeitlimit. Lieber eine verlorene Runde als eine unbeaufsichtigte Aktion.

Drei Schichten müssen zusammenpassen, und ich verwechsle sie bis heute regelmäßig. Die Allowlist erlaubt das Kommando. `sandbox.excludedCommands` befreit es von der Isolation, etwa weil es einen Docker-Daemon braucht. Der `env`-Block versorgt es mit den Variablen, die es zum Arbeiten braucht. Fehlt eine der drei, scheitert der Lauf, und die Fehlermeldung zeigt selten auf die Schicht, an der es liegt. Die Sessions selbst startet der Runner mit `--permission-mode acceptEdits`, damit Dateiänderungen nicht einzeln bestätigt werden müssen.

Ich erlaube das Werkzeug, nicht das einzelne Kommando: `Bash(mvn:*)` statt `Bash(mvn verify:*)`. Der Eintrag greift über das Präfix. `Bash(mvn verify:*)` deckt `mvn verify` ab, aber nicht `mvn -q verify`, nicht `mvn clean verify`, nicht `mvn test`. Das kostet Enge und spart Nächte. Die vier lesenden Git-Kommandos gehören mit hinein, damit eine Session beim Vorbereiten eines Commits nicht an einem harmlosen `git status` scheitert.

Ein Muster fällt aus beiden Schichten heraus: `env VAR=wert mvn verify`. Das erste Token ist dort `env` und nicht `mvn`, und weder die Allowlist noch `excludedCommands` erkennen das Kommando dahinter. Wer Variablen setzen muss, schreibt sie in den `env`-Block, statt sie vorne an die Kommandozeile zu hängen.

Die genaue Schreibweise dieser Einträge hängt von der Claude-Code-Version ab und steht in der Online-Doku. Wer morgens ein Protokoll findet, in dem jede Runde ohne Ergebnis geendet hat, sollte zuerst hier nachsehen.

Ein Beispiel, wie so eine `settings.json` aussehen kann:

```

{
  "env": {
    "DOCKER_HOST": "unix:///Users/manfredwolff/.colima/default/
docker.sock",
    "TESTCONTAINERS_DOCKER_SOCKET_OVERRIDE": "/var/run/
docker.sock"
  },
  "sandbox": {
    "enabled": true,
    "excludedCommands": ["mvn *"]
  },
  "permissions": {
    "allow": [
      "Bash(node .claude/kit/board.mjs:*)",
      "Bash(git add:*)",
      "Bash(git commit:*)",
      "Bash(npm test:*)",
      "Bash(npm run build:*)",
      "Bash(mvn:*)",
      "Bash(npm --prefix frontend run build:*)",
      "Bash(npm --prefix frontend run lint:*)",
      "Bash(npm --prefix frontend run test:coverage:*)",
      "Bash(git status:*)",
      "Bash(git diff:*)",
      "Bash(git log:*)",
      "Bash(git show:*)"
    ]
  }
}

```

Das ist meine echte Datei, kein Lehrbeispiel. Die `npm`-Zeilen stehen dort noch einzeln, weil ich sie eingetragen habe, bevor mir das Präfix-Matching klar war. Heute schreibe ich `Bash(npm:*)` und `Bash(npm --prefix frontend:*)` und spare mir drei Zeilen.

Für die Review-Nacht kommt eine Unterscheidung dazu. Ein Reviewer mit `kind: "claude"` läuft als Subagent über das Agent-Tool und braucht keine Permission. Ein Reviewer mit `kind: "command"` läuft über Bash, und steht er nicht in der Allowlist, erscheint nachts ein Permission-

Prompt, den niemand beantwortet. Das ist kein Fehler mit Log-Zeile. Die Session hängt bis zum Timeout.

```
"Bash(codex:*)"
```

Mehrere fremde CLIs werden einzeln eingetragen. Wer ausschließlich mit `kind: "claude"` prüft, braucht davon nichts.

Es gibt ein Flag, das all das abschaltet: `--yolo`, intern `--dangerously-skip-permissions`. Dann hängen die Stop-Punkte allein am Prompt der Skills. Eine Abkürzung ist das nicht. Ich setze es im Einzelfall und nie als Default.

## Was der Nachtbetrieb nicht ist

Der Nachtbetrieb verdoppelt nicht meine Arbeitszeit. Das reine Implementieren von Code kann aber vollständig automatisch laufen, wenn die Leitplanken stimmen. Diese Zeit spare ich mir gern, ich brauche sie an anderer Stelle.

Was nachts gebaut wird, ist das, was abends klar war. Wer abends unklare Issues freigibt, bekommt morgens unklaren Code, und zwar eine ganze Menge davon.

## KAPITEL 20

# Qualität und Security-Gates im Build verdrahten

Qualitätssicherung und Sicherheit gehören nicht ins Modell. Das ist die Kernaussage dieses Kapitels.

**Sie gehören außerhalb davon, in deterministische Werkzeuge, die den Build zum Scheitern bringen.**

Das ist keine Geringschätzung des Modells. Ich sage damit, was ein Gate leisten muss und was ein Sprachmodell strukturell nicht leisten kann.

## Warum das Modell kein Gate sein kann

**Es ist nicht deterministisch.** Derselbe Code, zweimal geprüft, ergibt nicht zwangsläufig denselben Befund. Ein Gate, das beim zweiten Versuch durchwinkt, ist kein Gate. Genau dieses Verhalten lädt mich ein, es einfach noch einmal zu versuchen, bis es passt.

**Es teilt seine blinden Flecken mit sich selbst.** Ein Modell ist gegenüber seiner eigenen Lösung systematisch unkritisch. Es hat sie gerade erzeugt und würde sie noch einmal genauso vereinfachen. Ein zweites Modell hilft, weil es andere blinde Flecken hat. Auch diese Hilfe hat Grenzen: Alle Modelle sind auf einer ähnlichen Basis trainiert, GitHub, GitLab, Stack Overflow und was es sonst an öffentlichem Code gibt. Die blinden Flecken zweier Modelle überschneiden sich deshalb wahrscheinlich stark.

**Es optimiert lokal.** Ein Sprachmodell löst das Problem, das vor ihm liegt. Es prüft, was im Kontextfenster steht. Ein Secret in einer Datei, die es nicht gelesen hat, findet es nicht. Ein deterministischer Scanner liest alles, jedes Mal.

**Es lässt mit sich reden.** Ich kann ein Modell von einem Befund abbringen, oft ohne es zu wollen. Das erinnert mich an Entwicklerinnen und Entwickler, die immer einen Grund finden, warum ausgerechnet bei ih-

nen keine hundertprozentige Testabdeckung möglich ist. Ein fehlschlagender Build lässt sich nicht überzeugen. Leitplanken müssen scheitern können, nicht argumentieren.

**Es meldet nicht, wo sein Muster aufhört.** Das ist der Punkt aus Kapitel 3, hier in seiner teuersten Form. Ein Modell antwortet auf die bekannte und auf die unbekannte Stellung im selben Ton, mit derselben Sicherheit. Bei einem Sicherheitsbefund heißt das: Ein “sieht gut aus” sagt nichts über den Code. Es sagt etwas über die Ähnlichkeit zu bekannten Mustern.

Dazu kommt ein Befund: **Sicherheit ist bei KI-generiertem Code strukturell fragil, nicht zufällig.** Das Modell schlägt SQL-String-Konkatenation vor, weil die Trainingsdaten voll davon sind. Es schlägt Debug-Logging mit sensiblen Daten vor, weil Hello-World-Tutorials so aussehen. Es übersieht Validierungen, weil es eilig zum Ergebnis will. Das ist kein Ausrutscher, das liegt in der Methodik des Modells.

Ein deterministisches Werkzeug teilt mit keinem Sprachmodell einen blinden Fleck. Ein roter Build blockiert den Push mechanisch, verlässlicher als jedes Modell.

## Wenn das Modell sich selbst prüft

Ein Modell, das mit seinem eigenen Code weitertrainiert wird, wird schlechter. Runde für Runde, bei jedem Modell, das die Forscher getestet haben. Die Arbeit stammt von der Emory University und der Universität Tokio, Juni 2026. Sie lassen ein Modell Code erzeugen, nehmen diesen Code als Lernmaterial für die nächste Version desselben Modells und wiederholen das. Ohne Prüfung sinkt die Trefferquote am schnellsten. Mit Compiler und statischen Prüfregelein wird der Verfall langsamer, er hört aber nicht auf. Der Code kompiliert weiterhin und wird trotzdem schlechter.

Der dritte Befund: Lassen sie das Modell seinen eigenen Code bewerten und nur das Gute behalten, steigen die Noten, die es sich selbst gibt,

während die tatsächliche Trefferquote fällt. Das ist der blinde Fleck gegenüber der eigenen Lösung, empirisch vermessen.

Die Bildforschung kennt das seit 2024. Die Rice University hat es Model Autophagy Disorder getauft, in Anlehnung an BSE, weil Kühe damals mit verarbeiteten Kühen gefüttert wurden.

Dieses Experiment ist kein Laborversuch. Es läuft gerade draußen, und niemand hat es gestartet. GitHub meldet im Mai 2026, dass an mehr als jedem fünften Code-Review auf der Plattform ein Agent beteiligt ist. Diese Repositories sind das Lernmaterial für die nächste Modellgeneration. Was darin steckt, misst GitClear: Der Anteil verschobenen Codes, das beste Maß für Aufräumen und Wiederverwenden, ist von 21 Prozent im Jahr 2022 auf 3,8 Prozent im Jahr 2026 gefallen, Copy und Paste im selben Zeitraum von 9,4 auf 15,7 Prozent gestiegen. GitClear verkauft Werkzeuge zur Codeanalyse, das sind also Anbieterzahlen. Eine unabhängige Arbeit von Anfang 2026 kommt zum selben Befund: Agenten lassen Gelegenheiten zur Wiederverwendung liegen und produzieren dadurch mehr Redundanz als Menschen.

In so einem Kreislauf verschwindet die Korrektheit nicht zuerst. Zuerst verschwindet die Bandbreite: die ungewöhnliche Lösung, der elegante Sonderweg, das Stück Code, von dem ich beim Lesen etwas lerne. Übrig bleibt der Durchschnitt. Das Fazit der Autoren, sinngemäß: Stabiles Weitertrainieren braucht eine Prüfinstanz, die nicht am Modell hängt. Das ist die Kernaussage dieses Kapitels, eine Etage höher.

## Die Werkzeuge

Welche Werkzeuge konkret passen, hängt vom Stack ab. Die Kategorien sind stabil, die Namen wechseln. Ich nenne hier den Stand 2026, als Ausgangspunkt gedacht und nicht als Rangliste.

| Kategorie               | Wogegen                                  | Werkzeuge             |
|-------------------------|------------------------------------------|-----------------------|
| Gesamtqualität, zentral | Code Smells, Duplikate, Trends über Zeit | SonarQube, SonarCloud |

| Kategorie                | Wogegen                                  | Werkzeuge                                                             |
|--------------------------|------------------------------------------|-----------------------------------------------------------------------|
| Secrets                  | API-Keys, Tokens, Zertifikate im Diff    | gitleaks, TruffleHog                                                  |
| Injection und Datenfluss | SQL-Konkatenation, fehlende Validierung  | Semgrep, SpotBugs mit FindSecBugs                                     |
| Statische Analyse        | Fehlermuster, Nullability, Bad Practices | PMD, ErrorProne, SpotBugs, ESLint                                     |
| Code-Style               | Uneinheitliche Formatierung              | Spotless, Checkstyle, Prettier                                        |
| Architektur              | Verletzte Schichtengrenzen               | ArchUnit, ArchUnitNET, PyTestArch, Deptrac, phpat, dependency-cruiser |
| Testgüte                 | Tests, die nichts prüfen                 | PIT für Java, Stryker für JS                                          |
| Abhängigkeiten           | Bekannte CVEs in Bibliotheken            | OWASP Dependency-Check, Trivy, npm audit                              |
| Typen                    | Typfehler im Frontend                    | tsc, Vite-Build                                                       |
| Veraltete APIs           | Abgekündigte Idiome, flächig ausgerollt  | @typescript-eslint/no-deprecated, -Xlint:deprecation -Werror          |

Die letzte Zeile hat einen KI-spezifischen Grund. Ein Sprachmodell reproduziert das häufigste Muster seines Trainingskorpus, nicht das aktuellste. Eine vor Monaten abgekündigte Schnittstelle steht in Millionen Zeilen Altcode noch als der normale Weg, und genau so kommt sie zurück. Das Ergebnis ist ein Denkfehler in vielfacher Ausführung: dasselbe veraltete Idiom, sauber über alle Aufrufstellen ausgerollt.

Dagegen hilft kein Prompt und keine handgepflegte Verbotsliste. Der Compiler und der Linter wissen ohnehin, was abgekündigt ist, ich muss es nur scharf schalten. Der Analyzer skaliert mit dem Ökosystem, die Liste skaliert nur mit der Pflegedisziplin.

Am breitesten nutzt mir davon SonarQube, weil es die Befunde über Zeit verfolgt. Der interessante Wert dort ist der Trend auf neuem Code, nicht die absolute Zahl. Ein Projekt mit Altlasten wird nie sauber, aber alles, was ab heute dazukommt, kann es sein.

Am wichtigsten für den KI-Modus halte ich zwei andere: ArchUnit und Mutation Testing.

**ArchUnit** formuliert Architekturregeln als automatische Tests. ArchUnit selbst ist ein Java-Werkzeug, das Konzept gibt es aber für andere Sprachen auch: ArchUnitNET für C#, PyTestArch für Python, Deptrac und phpat für PHP, im JavaScript-Umfeld dependency-cruiser. Seit ich das einsetze, bekomme ich vom Modell keinen Code mehr, der Schichtengrenzen verletzt. Der Build schlägt fehl, das Modell muss es richtig machen. Regeln gehören in Dateien, nicht in Köpfe.

**Mutation Testing** prüft, ob die Tests etwas taugen. Coverage zeigt nur, ob eine Zeile durchlaufen wurde, nicht ob sie geprüft wurde. Genau diese Lücke füllt ein Modell besonders gern mit Tests, die grün sind und nichts aussagen.

## Befunde zurück ins Backlog

Bei SonarQube mache ich eine Rückkopplung: Alle Befunde, die es findet, spiegele ich zurück ins Backlog. Dort kann ich sie priorisieren und entscheiden, ob ein Befund ein Fehlalarm ist, ob ich ihn akzeptieren kann oder ob ich ihn im nächsten Schritt mitbearbeite.

Wenn SonarQube in einem Lauf neunzehn Issues meldet, liegen diese neunzehn Punkte anschließend als Karten auf meinem Board. Damit werden sie zu dem, was jede andere Arbeit auch ist: etwas, das ich priorisieren, schneiden und freigeben kann. Ein Dashboard reicht mir dafür nicht. Die Befunde wandern ins Backlog, wo ich immer wieder über sie stolpere.

Technische Schuld wird sonst zu einer Sache, die ich irgendwann mal angehe, und irgendwann ist nie. Als Karte im Backlog konkurriert sie ganz normal mit den Features, und manchmal gewinnt sie.

Praktisch heißt das bei mir: Ich habe Nächte, in denen der Runner ausschließlich Refactorings und Bugfindings aus SonarQube abarbeitet. Abends ziehe ich zehn oder fünfzehn dieser Karten nach Ready, setze das Routing-Label, starte den Nacht-Runner, und am nächsten Morgen prüfe ich das Ergebnis wie jeden anderen Nachtlauf. Solche Nächte sind unspektakulär und gehören zu den produktivsten, die ich habe.

Diese Karten sind Issues wie alle anderen, also gilt für sie dasselbe Schnittkriterium. Ein SonarQube-Befund, der lautet “Cognitive Complexity zu hoch in Methode X”, ist ein gutes Nacht-Issue. Ein Befund, der eine Architekturentscheidung berührt, ist es nicht. Der wandert in den Plan, und morgens bespreche ich ihn.

## Schwellenwerte

Meine Zielwerte sind unbequem, und ich vertrete sie trotzdem.

**Coverage: 100 Prozent oder ein begründeter, dokumentierter Ausschluss.** Ein unbegründeter Ausschluss ist ein Leck.

**Mutation Score: so hoch wie möglich, und wenn ich priorisieren muss, gewinnt er gegen Coverage.** 98 Prozent Coverage mit 70 Prozent Mutation Score ist schlechter als 95 Prozent Coverage mit 98 Prozent Mutation Score. In meinen Projekten liegt der Mutation Score immer bei 100 Prozent. Ich hatte es auch schon mit Modellen aufgegeben, weil sie die 100 Prozent nicht erreicht haben. Und dann hat Fable es doch geschafft.

**Statische Analyse: keine offenen Findings.** Nicht “wenige”, keine. Es gibt False Positives, und es gibt Befunde, die ich kleine Schönheitsfehler nenne. Die dürfen bleiben, aber nicht, weil niemand hingeschaut hat: Ich habe sie gesehen und im SonarQube wissentlich akzeptiert. Die offene Liste geht nicht, die wird innerhalb von Wochen zur Sammlung ignorierte Findings.

Der Einwand dagegen ist immer derselbe: Das koste zu viel Zeit. Er stimmte, solange das Schreiben von Tests teuer war. Genau dieser Posten ist billig geworden. Wer die KI-Beschleunigung mitnimmt und die Schwellen nicht mit anzieht, hat den Gewinn in Risiko umgetauscht.

## Wie es im Prozess verdrahtet wird

Drei Ebenen, und alle drei müssen stimmen.

**Im Build.** Coverage-Schwellen, Mutation-Schwellen, ArchUnit-Tests und statische Analyse richte ich im Maven- oder npm-Build als verbindliche Gates ein. Ein Build, der nicht grün ist, blockiert. Diese Verdrahtung kostet am Anfang Zeit und spart sie über Monate.

**In der Config.** Die Kommandos wandern in `buildChecks`. `/local-check` führt sie nacheinander aus, ein roter Check stoppt den weiteren Prozess, und `/push-main` ist mechanisch blockiert, solange kein Check grün ist. Und der Nacht-Runner startet ohne `buildChecks` gar nicht erst, jedenfalls nicht, wenn er implementieren soll.

**Im CI.** Dieselben Checks laufen noch einmal auf dem Server, plus die, die lokal zu lange dauern. Lokale Checks lassen sich umgehen, ein Hook lässt sich überspringen. Der CI-Lauf ist die Instanz, die niemand überredet.

Sicherheits-Gates gehören ausdrücklich in diese dritte Ebene und nicht in einen Skill. Der Review-Skill ergänzt sie, er ersetzt sie nicht.

## Was das Modell im Review wirklich leistet

Das Review ist deswegen nicht überflüssig.

Ein Modell findet Dinge, die kein Werkzeug findet: ob der Code das Problem löst, das im Issue steht. Ob ein Randfall fehlt, den niemand bedacht hat. Ob eine Benennung in die Irre führt. Ob eine Änderung an einer Stelle eine Annahme an einer anderen Stelle bricht. Ob die Lösung umständlich ist, obwohl sie funktioniert.

Das sind Urteilsfragen, und dafür lohnt sich ein zweiter Blick. Die Werkzeuge prüfen, was mechanisch prüfbar ist. Das Modell liest, was Bedeutung hat. Und der Mensch entscheidet, was mit beidem geschieht.

## Die manuelle Verifikation

Ein Punkt zum Schluss, der in keiner Werkzeugliste steht.

Lange galt: Das Modell kann nicht im Browser klicken. Das stimmt so nicht mehr. Die neueren Versionen können einen Browser starten, navigieren und klicken. Damit läuft ein Golden Path automatisiert durch, und das ist eine echte Verbesserung.

Die Zuständigkeit ändert sich dadurch nicht. Ein Modell kann prüfen, ob ein Klick zu einer erwarteten Ausgabe führt. Ob eine Oberfläche verständlich ist, ob eine Fehlermeldung hilft, ob eine Ladezeit sich falsch anfühlt, ist eine andere Art von Urteil. Bei jeder Frontend-Änderung gehört deshalb weiterhin eine eigene Verifikation dazu: Dev-Server starten, Golden Path durchgehen, mindestens einen Randfall prüfen. Drei Fehler, die mich Produktionszeit gekostet haben, waren erst im Smoke-Test auf dem Server sichtbar: ein Docker-Upgrade, das Traefik zerlegt hat, ein Token, das nur `PENDING` statt `USER` mitbrachte, und ein CORS-Header, der genau eine Origin zu wenig kannte. Alle drei bei grünen Tests.

Dazu gehört, die Testinstanzen auseinanderzuhalten. Bei mir sind es drei Stufen: lokal auf dem Rechner, wo ich oft auch mit Docker teste. Dann der Testserver, der die Produktionsumgebung eins zu eins abbildet und auf dem ich alles durchspielen kann. Und zum Schluss geht es auf Produktion.

Wenn diese Verifikation nicht möglich ist, weil kein Docker läuft oder kein lokaler Build erreichbar ist, gehört das in den Abschlussbericht. Einen Schritt nicht ausführen zu können, ist keine Schande. Ihn zu verschweigen, schon.

## TEIL V

# Den Prozess am Leben halten

Ein Prozess, den niemand pflegt, beschreibt nach einem halben Jahr nur noch, was einmal galt. Dieser Teil handelt davon, wie er am Leben bleibt.

Kapitel 21 zeigt, wie ich den Prozess in einem neuen Team einführe, und beginnt mit einem Rat, der dem Rest des Buches widerspricht: ohne Infrastruktur anfangen. Kapitel 22 sammelt die Anti-Patterns, alle real beobachtet und die meisten bei mir selbst. Kapitel 23 beschreibt die KI-Retrospektive, die einzige Veranstaltung, in der sich der Prozess selbst überarbeitet.

Kapitel 24 blickt nach vorn, so weit das seriös möglich ist. Ich habe dort keinen fertigen Vorschlag, und ich misstraue allen, die jetzt schon ein neues Framework verkaufen wollen. Die Praxis ist der Theorie gerade voraus.

## KAPITEL 21

# Erste Schritte für neue Teams

Wer den Prozess in einem neuen Team einführen will, kann das an einem Vormittag tun. Die folgende Liste ist die Minimalausstattung. Vieles davon ist nicht spektakulär. Es trennt trotzdem das Team, das mit KI arbeitet, von dem, das von der KI bearbeitet wird.

## Klein anfangen

Fang ohne Infrastruktur an. Das ist mein erster Rat, und er widerspricht dem Rest dieses Buches.

Kein Vault, kein selbst gehostetes Board, keine neue Plattform. Installiere das Kit im lokalen Modus, lass die Issues als Dateien im Repo liegen und fahre den Prozess eine Woche lang für dich allein. Danach weißt du, welche Teile du wirklich brauchst. Wer mit der Infrastruktur anfängt, verbringt drei Tage mit Konfiguration und hat den Prozess noch kein einziges Mal durchlaufen.

## Die Checkliste

**1. Dateien im Repo-Root.** Die zentrale Konventionsdatei als Projekt-Doku: was tut dieses Projekt, welche Standards gelten. Die Workflow-Konventionsdatei mit dem Neun-Schritt-Prozess. Tech-spezifische Dateien je Stack. Eine Security-Konventionsdatei mit der Checkliste vor dem Commit. Diese Dateien sind zwischen Projekten kopierbar. Einmal sorgfältig erstellt, wandern sie ins nächste Projekt mit.

**2. Ein Board mit fünf Spalten.** Backlog, Ready, In progress, In review, Done. Wichtig ist die Konvention dahinter, nicht das Werkzeug: Ready-Bewegungen macht der Mensch, In-progress- und In-review-Bewegungen macht die KI, Done macht der Mensch.

**3. Trigger-Phrasen vereinbaren.** Eine für die Implementierungsfreigabe, eine für den Push, eine für den Merge. Sie stehen in der Workflow-

Datei und gelten für alle. Legt sie einmal bewusst fest und fasst sie dann nicht mehr an. Eine Trigger-Phrase, über die diskutiert wird, hat ihre Funktion verloren.

**4. Pflicht-Checks im Build verdrahten.** Coverage-Schwellen, Mutation-Schwellen, ArchUnit-Tests, statische Analyse als verbindliche Gates im Build, danach dieselben Kommandos in `buildChecks`. Diese eine Investition kostet am Anfang Zeit und spart sie über Monate. Kapitel 20 sagt, warum sie nicht optional ist.

**5. Ein zweites Modell für Reviews festlegen.** Welches, ist zweitrangig. Es muss nur ein anderes sein als das implementierende. Das Setup gehört in eine kurze Anleitung, damit ein neues Teammitglied am ersten Tag damit arbeiten kann.

**6. Das erste Issue: das Setup selbst.** Das klingt rekursiv. Es ist der erste Praxistest des Prozesses: Plan, Issue, Ready, Implementierung, Review, Push. Wer den Prozess einmal durchlaufen hat, kennt ihn vom ersten Tag an, statt ihn nachzulesen.

## Die Reihenfolge, in der ich es heute machen würde

| Tag          | Was                                                              |
|--------------|------------------------------------------------------------------|
| 1            | Kit im lokalen Modus installieren, einen Durchlauf allein fahren |
| 2 bis 5      | Täglich arbeiten, Konventionsdateien beim Arbeiten füllen        |
| Ende Woche 1 | Erste KI-Retrospektive, Regeln schärfen                          |
| Woche 2      | Pflicht-Checks verdrahten, zweites Modell einrichten             |
| Woche 3      | Board dazunehmen, Team aufschalten                               |
| Woche 4      | Vault einrichten, falls mehrere Projekte laufen                  |
| Danach       | Nachtbetrieb, wenn die Issues sauber genug geschnitten sind      |

Der Nachtbetrieb steht bewusst am Ende. Er verzeiht keine unklaren Issues, und eindeutig zu schneiden lernt ein Team erst nach einigen Wochen Praxis.

## Woran es in Teams scheitert

Drei Muster sehe ich immer wieder.

**Der Prozess wird eingeführt, die Gates nicht.** Dann gilt er, solange jemand daran denkt. Nach zwei Wochen denkt niemand mehr daran. Ein Gate, das nicht im Build steht, ist eine Bitte.

**Die Stop-Punkte gelten als Bremse und fallen weg.** Meistens beginnt es beim Ready, weil der Plan ja schon abgenommen war. Kapitel 22 handelt davon.

**Es gibt keinen Ort, an dem der Prozess sich ändern darf.** Dann befolgt ihn das Team starr, bis er nicht mehr passt, oder gibt ihn still auf. Deshalb gehört die Retrospektive aus Kapitel 23 von Anfang an dazu, nicht erst, wenn es hakt.

## KAPITEL 22

# Anti-Patterns und wie man sie erkennt

Ein Prozess wird erst konkret, wenn er seine typischen Verletzungen benennt. Die folgenden Anti-Patterns habe ich alle erlebt, die meisten bei mir selbst. Wer sich bei einem dieser Gedanken ertappt, sollte anhalten.

## “Plan-Approval reicht, ich fange einfach an”

Das ist die Verletzung, die mir am häufigsten begegnet. Ich akzeptiere einen Plan, das Modell liest das als Freigabe zum Implementieren und fängt an zu coden.

Plan-Approval ist Schritt 2, das Ready ist Schritt 4. Diese Trennung ist die zentrale Sicherheitsschwelle, und wer sie aufweicht, kann den ganzen Rest auch streichen.

Zwischen Plan und Ready liegt das Schneiden in Issues, und genau dabei fällt auf, was im Plan noch unklar war.

## “Schnell pushen, ist eh trivial”

Ein kleiner Bugfix, ein CSS-Tweak, ein Doku-Update. Die Versuchung kenne ich gut. Falsch ist sie trotzdem. Jeder Push verändert den Test-Server, und auch eine CSS-Änderung kann ein Layout zerschießen, das ein Stakeholder gleich sehen sollte.

Jeder Commit-Batch braucht seine eigene Freigabe. Eine Freigabe von gestern gilt nicht für die Commits von heute.

Bahn 1 aus Kapitel 8 spart den Plan, das Issue und das Ready. Den Push spart sie nicht.

## “Tests folgen später”

Sie folgen nicht. Wer Test-First aufschiebt, landet bei Test-Never.

Im KI-Modus kommt ein zweiter Grund dazu. Wer sagt “schreib mir Funktion X”, bekommt eine plausibel aussehende Funktion X. Wer sagt “schreib einen Test, der Verhalten X spezifiziert, und dann eine Funktion, die ihn grün macht”, bekommt eine Funktion mit ausdrücklichem Vertrag. Mit dem Test binde ich das Modell an die Realität.

### **“Coverage bei 98 Prozent reicht”**

Der Denkfehler steckt in der Zahl selbst. Coverage misst, ob ein Test über eine Zeile gelaufen ist, nicht ob er sie geprüft hat. Wer diese Zahl zum Ziel macht, bekommt genau das: Tests, die durchlaufen und nichts behaupten. Die Schwellenwerte und ihre Begründung stehen in Kapitel 20.

### **“Die KI macht das schon richtig”**

Vertrauen ohne Verifikation ist die teuerste Form der Naivität. Ein Modell ist sehr gut darin, plausibel zu klingen. Seine eigenen Fehler findet es nicht. Prüfen muss der Mensch.

### **“Bei komplexen Aufgaben gebe ich mehr ab”**

Dieser Denkfehler ist weit verbreitet und aus meiner Erfahrung der teuerste. Er klingt vernünftig: Je schwerer die Aufgabe, desto mehr Hilfe.

Er ist falsch herum, und Kapitel 3 führt aus, warum. Woran ich merke, dass ich gerade abrutsche: Ich lese einen Vorschlag und nicke, statt ihn nachzurechnen. Ich frage das Modell, wie etwas funktioniert, statt es zu prüfen. Ich akzeptiere einen Plan, dessen Begründung ich nicht wiedergeben könnte. Bei komplexen Aufgaben wandert mehr Denkarbeit nach vorn, nicht weniger.

## **Der Auto-Modus**

Ein schleichendes Muster, das ich an einem Tag zweimal erlebt habe. Das Modell bleibt nach einem Bug in Bewegung, statt anzuhalten. Es fixt etwas ohne Plan und ohne Ready. Jeder einzelne Schritt wirkt sinnvoll, und in der Summe ist der Prozess weg.

Der Code war beide Male in Ordnung. Es war kein technischer Bug, es war ein Prozess-Bug. Zurückgerollt gehört der Workflow, nicht der Code.

## **Memory-Drift**

Die Memory-Dateien wachsen, aber niemand pflegt sie. Am Ende stehen alte Regeln in einer neuen Realität. Kapitel 17 beschreibt den Mechanismus, Kapitel 23 zeigt, wo ich ihn prüfe.

## **Halluzinationen als verifizierte Wahrheit**

Das Modell erfindet plausibel klingende Bibliotheksnamen, Methodensignaturen, Konfigurationsoptionen. Wer eine solche Erfindung ungeprüft übernimmt, baut Phantomschulden ein und zahlt sie später mit Zinsen zurück.

Halluzinationen gehören zum Normalbetrieb. Jede Behauptung mit externem Bezug prüfe ich nach.

## **Drei Modelle als Vier-Augen-Prinzip**

Ein Muster, das nach Sorgfalt aussieht. Wenn ein zweites Modell gut ist, sind drei besser, und wenn alle drei zustimmen, muss es stimmen.

Das ist ein Trugschluss. Mehr Modelle liefern mehr Material, sie ersetzen das Urteil nicht. Jedes Modell hat andere blinde Flecken, und der Wert liegt genau in den Diskrepanzen. Der Mensch dazwischen interpretiert sie. Drei zustimmende Modelle sind kein Beleg, sie sind drei Stimmen aus einer ähnlichen Verteilung.

Die Zahl ist nur die falsche Stellschraube. Drei Modelle mit demselben Prompt sind derselbe Blick dreimal, drei Modelle mit verschiedenen Rollen sind drei Fragen an dasselbe Dokument. Wie ich das im Prozess gelöst habe, steht in Kapitel 9. Auch dort entscheide ich, nicht die Mehrheit.

## Überautomatisierung ohne Stop-Punkte

Nachtläufe sind gut. Nachtläufe ohne Stop-Punkte sind es nicht. Wer einen Hintergrundprozess startet, der für jedes neue Issue eigenständig einen Branch öffnet, Code schreibt, Tests grün macht und einen Pull Request aufmacht, hat die menschliche Kontrolle verloren.

Autonomie vervielfacht die Konsequenzen jeder Entscheidung, die niemand getroffen hat. Stop-Punkte gehören deshalb zur Struktur des Prozesses.

## Das gefährlichste Muster

Der Tag, an dem alles glatt läuft. Vor dem hüte ich mich am meisten, weil er sich nicht wie ein Fehler anfühlt.

Nachts sind acht Issues durchgelaufen, alle Checks sind grün, das Review erscheint überflüssig. Wer KI unkontrolliert arbeiten lässt, produziert schneller schlechten Code. Ich merke es nur nicht am selben Tag.

## KAPITEL 23

# Die KI-Retrospektive

Die KI-Retrospektive ist das Event, das ich neu dazugenommen habe, und sie ist die einzige Veranstaltung, in der sich der Prozess selbst überarbeitet.

Ein Tagesordnungspunkt in der allgemeinen Retrospektive reicht dafür nicht. Die klassische Retro spricht über Menschen und Zusammenarbeit. Wenn die Mensch-Modell-Fragen dort als letzter Punkt auftauchen, fallen sie aus meiner Erfahrung hinten runter oder bekommen eine zu allgemeine Antwort.

Takt: alle ein bis zwei Wochen. Beteiligt: Menschen. Dauer: eine halbe Stunde reicht.

## Die drei Fragen

**Wo hat die Zusammenarbeit gehakt?** Nicht “war das Modell gut”. Konkrete Momente: Wo musste ich dreimal dasselbe sagen? Wo hat ein Skill etwas getan, was er nicht sollte? Wo bin ich in den Auto-Modus gerutscht?

**Welche Memory-Einträge sind veraltet oder falsch?** Welche Konvention gilt nicht mehr? Welche Regel habe ich mehrfach gebrochen, und war das Nachlässigkeit oder ist die Regel überholt?

**Welche Workflow-Regel braucht eine Schärfung?** Wo hat sich ein Anti-Pattern wiederholt? Wo ist eine Anweisung so allgemein, dass sie jeder anders liest?

## Der Output sind Dateien, keine Erkenntnisse

Ob eine Retrospektive etwas taugt, sehe ich am Ergebnis.

Aus einer KI-Retrospektive kommen konkrete Änderungen an den Konventionsdateien, an den Memory-Dateien oder an der Prompt-Biblio-

thek. Wenn eine Retrospektive keine Datei verändert, war sie zu abstrakt.

“Wir sollten sorgfältiger mit Issues umgehen” ist kein Ergebnis. “Die Workflow-Datei bekommt den Satz, dass ein Issue ohne Out-of-Scope-Abschnitt nicht nach Ready darf” ist eines.

Der `/retro`-Skill hilft mir dabei: Er stellt die drei Fragen und formuliert die vorgeschlagenen Änderungen als Diff. Welche davon ich übernehme, entscheidet er nicht.

## Was regelmäßig herauskommt

Drei Sorten von Ergebnissen sehe ich immer wieder.

**Regeln, die zu allgemein waren.** Aus “Tests schreiben” wird “vor der Implementierung Testfälle definieren, weil sie die Akzeptanzkriterien absichern”. Das Modell braucht die Begründung, weil sie beim Kompaktieren als Letztes wegfällt.

**Regeln, die niemand mehr braucht.** Eine Konvention, die für ein Framework galt, das ich inzwischen ersetzt habe. Solche Einträge streiche ich genauso konsequent, wie ich neue schreibe, denn jede Zeile in einer Konventionsdatei kostet Kontextfenster.

**Regeln, die ins Werkzeug gehören statt in eine Datei.** Wenn ich dieselbe Regel dreimal verletze, steht sie am falschen Ort. Eine Architekturregel gehört in einen ArchUnit-Test, der den Build bricht, nicht in eine Konventionsdatei. In der Retrospektive fällt mir so etwas auf.

## Warum es dieses Format braucht

Das zwölfte agile Prinzip verlangt, dass ein Team regelmäßig reflektiert, wie es effektiver werden kann. Im KI-Modus bekommt dieses Prinzip eine zweite Ebene. Ich halte es damit für stärker und nicht für komplizierter.

Dazu kommt der praktische Grund: Die Werkzeuge verschieben sich monatlich. Wer nicht regelmäßig reflektiert, sammelt Memory-Drift an. Ein

Prozess, der sich nicht in demselben Takt überarbeiten darf, ist nach einem halben Jahr eine Beschreibung dessen, was einmal galt.

Deshalb ist auch dieses Buch versioniert. In Anhang E steht, wie es entstanden ist und nach welchen Regeln ich es überarbeite. Ich kann schlecht einen Prozess beschreiben, der sich selbst überarbeitet, und den Text darüber unverändert stehen lassen.

## KAPITEL 24

# Ausblick

Alles in diesem Buch ist eine Momentaufnahme. Die Werkzeuge ändern sich monatlich, die Best Practices entsprechend. Zwei Schritte stehen bei mir schon fest, danach kommt, was ich für absehbar halte.

## Die nächsten zwei Schritte

Version 1.6 bekommt einen durchgehenden Lauf, von der fertigen fachlichen Anforderung bis zum Issue in In review, ohne Halt dazwischen. Heute liegen auf dieser Strecke drei Schritte und ein Stop-Punkt: Der Fachplan wird zum Plan, der Plan wird zu Issues, und dann ziehe ich nach Ready.

Das steht quer zu Kapitel 3, und ich löse den Widerspruch nicht auf, ich grenze ihn ein. Ein Lauf, der Plan, Schnitt und Freigabe in einem Zug erledigt, nimmt mir genau die Stelle weg, an der ich sonst merke, dass die Anforderung nicht trägt. Deshalb ist dieser Modus für kontrolliertes Vibe-Coding gedacht: für einen Prototyp, für ein kleines Werkzeug, für etwas, das ich notfalls wegwerfe. Für ein großes Softwaresystem ist er nicht gedacht, und das schreibe ich im Kit auch so hin.

Version 1.7 wird Spec Driven Development. Als ich mir angesehen habe, was dort gefordert wird, war das für mich ein Abgleich, keine neue Methode. Vier der fünf Stufen standen bei mir schon, nur unter anderen Namen: die nicht verhandelbaren Prinzipien in der Config, der Entwurf als Fachplan und Plan, die Aufgaben als kleingeschnittene Issues, die Umsetzung als `/implement-ready` samt Build. Der Gedanke selbst ist alt. Dan North hat JBehave 2004 veröffentlicht, und noch im selben Jahr wurde aus der Testtechnik eine Technik für Anforderungen.

Die fünfte Stufe fehlte, und es ist die wichtigste: die Spec. Was tut das System heute? Auf diese Frage hat mein Prozess kein Artefakt. Das Issue beschreibt eine Änderung, das Archiv beschreibt, warum sie so ausgefal-

len ist, der Vault beschreibt eine Entscheidung. Den Zustand beschreibt keines von dreien.

Deshalb liest ein Modell, das ein bestehendes Feature ändern soll, heute Produktionscode, und zwar bei jedem Plan neu. Das geht systematisch schief, sobald der Code mehr kann als beabsichtigt, und auf die Frage, warum etwas so ist, antwortet er gar nicht. Mein Ziel ist deshalb hart formuliert: Der Plan liest für Bestandsverhalten keinen Produktionscode mehr, und wo er es doch tut, weist er die Stelle als Spec-Lücke aus.

Damit aus der Spec nicht wird, was aus Dokumentation sonst wird, hängt sie am Build. Jede Aussage bekommt eine ID, die ID verweist auf einen Test, und eine Aussage ohne Test-Verweis lässt die Prüfung scheitern. Wer eine Spec von Hand ändert, fliegt damit auf. Das ist dieselbe Regel wie überall in diesem Buch, nur prüft sie diesmal die Beschreibung des Codes und nicht den Code.

In ein Bestandssystem kommt so eine Leitplanke nicht an einem Tag. Jede Spec-Datei trägt deshalb einen Reifegrad. Im Entwurf schaut das Gate weg, ab gültig greift es, und das Hochstufen ist eine menschliche Handlung. Kein Skill setzt einen Reifegrad, auch meiner nicht. Die Spec wächst dabei aus den grünen Tests, nicht aus fünfhundert Issues: Deren Summe ergibt die Historie und nicht den Ist-Zustand. Zuerst wächst sie dort, wo sich am meisten ändert.

Wie viel davon in 1.7 landet, weiß ich noch nicht. Der Vortrag, aus dem diese Überlegungen stammen, ist von August 2026, und die Baustelle ist offen.

## Mehr Autonomie

Ich rechne damit, dass die heutige Implementations-Engine morgen in Mehr-Agenten-Aufstellungen läuft: ein planender Agent, ein implementierender, ein testender, ein dokumentierender, koordiniert von einem Menschen.

Mein Prozess funktioniert auch in einer solchen Welt, ich besetze nur einige Rollen anders. Unverändert gilt: Je autonomer die KI arbeitet, desto

strenger müssen die menschlichen Stop-Punkte sein. Autonomie ist nicht das Problem. Autonomie ohne Stop-Punkte ist es.

Mein eigener Nachtbetrieb ist ein kleiner Vorgriff darauf. Der Gewinn kam nicht davon, dass nachts mehr entstand. Er kam davon, dass ich abends sauberer schneiden musste.

## **Continuous AI wird Normalität**

Heute heißt das noch Vision, in zwei Jahren ist es aus meiner Sicht selbstverständlich. Ich rechne mit Repositories, die Hintergrund-Loops haben, in denen kleine Verbesserungen kontinuierlich entstehen.

Damit muss ich umso schärfer trennen, was menschlich getaktet läuft und was im Hintergrund. In den Hintergrund gehört, was kein menschliches Urteil braucht, während es entsteht. Alles, was Architektur, Sicherheit oder Stakeholder berührt, gehört in den menschlichen Takt.

## **Lokale Modelle**

Was ins Kontextfenster wandert, hat den eigenen Rechner verlassen. Für die meisten Projekte ist das vertraglich lösbar, und in neun von zehn Fällen, die ich erlebt habe, ist die Aufregung darüber größer als das Risiko. Es bleiben Fälle, in denen sie berechtigt ist: echte Air-Gap-Anforderungen, stark regulierte Branchen, nicht verhandelbare Kundenaufgaben.

Lokale Modelle sind die richtige Antwort darauf. Für Reviews und kleine Refactorings sind sie schon heute brauchbar. Für die Implementations-Engine reicht die Hardware auf einem Arbeitsplatzrechner im Jahr 2026 nicht.

Das erledigt sich schnell. Ich rechne damit, dass dieser Punkt als Erstes von allem in diesem Buch überholt ist.

Die Tür dafür ist erstaunlich klein. Claude Code spricht ausschließlich die Anthropic-API, und wohin es dabei spricht, hängt an einer einzigen Variablen. Zeigt sie woandershin, läuft der ganze Apparat weiter: dieselben Skills, dieselben Werkzeuge, derselbe Prozess, nur ein anderes Modell dahinter. Das Kit merkt davon nichts. Es reicht einen Modellnamen

durch und schreibt ihn ins Protokoll. Was hinter dem Endpunkt antwortet, ist ihm gleich.

Für ein lokales Modell brauche ich dazwischen einen Übersetzer, weil Ollama und Verwandte das OpenAI-Format sprechen. Üblich ist LiteLLM. Gefahren bin ich das selbst nicht. Was jetzt kommt, ist ein begründeter Vorschlag, kein Erfahrungsbericht.

```
ANTHROPIC_BASE_URL=http://localhost:4000 \
  node .claude/kit/night.mjs --model lokal-qwen --label kit:lokal
--max 3
```

Vorangestellt, nicht exportiert, aus demselben Grund wie beim Token in Kapitel 19: So gilt die Variable nur für diesen Lauf, und eine parallel laufende interaktive Sitzung bleibt unberührt.

Die Arbeit teile ich über Labels auf, ein neues Flag brauche ich dafür nicht: Einfache Issues bekommen ein eigenes, dann laufen zwei Durchgänge nacheinander. Das Review ändert sich dadurch nicht, es nimmt weiter das Modell aus der Config.

Eine Nacht-Session muss Werkzeuge zuverlässig aufrufen, eine mehrstufige Kette durchhalten und sauber committen. Ich erwarte, dass kleine Modelle genau daran scheitern und nicht am Programmieren. Projekte mit scharfen Gates, also Mutationstests, Coverage-Schwellen, mehrstufige Build-Ketten, sind für ein kleines lokales Modell außer Reichweite. Was bleibt, sind Änderungen ohne Testpflicht: Dokumentation, Textkorrekturen, Konfigurationswerte, kleine mechanische Anpassungen.

Kaputtmachen kann so ein Lauf wenig. Die Vorprüfung fährt die Checks selbst, rote Checks führen zum harten Stopp, unkommittierte Reste beenden den Lauf. Ein gescheiterter lokaler Lauf kostet Strom und Zeit, nicht die Codebasis.

## Die Nachwuchsfrage bleibt ungelöst

Kaum jemand redet über diese Entwicklung, und mich beschäftigt sie am meisten.

Die Karrierepyramide bricht an der Basis weg. Wenn die Nachwuchspipeline jetzt austrocknet, fehlt in fünf Jahren die Erfahrung, nicht der Nachwuchs. Und Senior-Kompetenz braucht Jahre, die bestellt niemand zur nächsten Hauptversammlung nach.

Ich habe darauf keine vollständige Antwort, nur zwei Ansätze aus Kapitel 4: Review-first statt Code-first, und Reibung bewusst wieder einbauen. Beides ist Didaktik, kein Maschinenstürmertum. Ob es reicht, weiß ich nicht.

## Wie weit die Verdrängung geht

Die schärfste Gegenrede zu meiner Einschätzung kommt von Heiko Dietz, dessen Einwand in der Einleitung steht. Seine These reicht deutlich weiter als meine: Verdrängt werde alles, was nicht Spitzenniveau hat, und das ist die große Breite. Ein Produkt, für das heute ein ganzes Team arbeitet, ließe sich von zwei oder drei Personen bauen, die die Modelle steuern, bei höherer Qualität und deutlich größerem Durchsatz. Er verweist dabei auf Personalentscheidungen großer amerikanischer Technologiekonzerne.

Ich nehme den technischen Teil dieser These ernst und halte den wirtschaftlichen für offen.

Ernst nehme ich sie, weil ich den Durchsatzsprung selbst erlebe. Was ich in einer Nacht durch die Ready-Spalte schicke, hätte vor drei Jahren eine Woche gedauert. Wer das bestreitet, hat es nicht ausprobiert.

Offen halte ich den wirtschaftlichen Teil aus zwei Gründen. Erstens setzt die These voraus, dass die Nachfrage nach Software ungefähr konstant bleibt und höhere Produktivität deshalb Stellen freisetzt. Das war in der Geschichte dieses Berufs noch nie so. Bisher hat billigere Herstellung immer dazu geführt, dass mehr Software gebaut wurde, und nie dazu, dass dieselbe Menge weniger kostete.

Zweitens setzt sie voraus, dass diese zwei oder drei Personen einfach da sind. Sie sind aber genau die Senioren, die aus der Breite hervorgehen, die verschwinden soll. Eine Branche, die nur noch Spitzenkräfte be-

schäftigt, muss erklären, woher die nächste Generation kommt. Das ist die Nachwuchsfrage aus dem vorigen Abschnitt, nur eine Etage höher.

Wer recht behält, entscheidet keine Prognose. Das entscheiden die nächsten Jahre. Heiko Dietz kennt beide Seiten: Er hat die Sache erst abgelehnt und setzt sie heute produktiv ein.

## Die offenen Fragen

Ich habe keinen fertigen Vorschlag, und ich misstraue allen, die jetzt schon ein neues Framework verkaufen wollen. Die Praxis ist der Theorie gerade voraus.

Wie balanciere ich KI-Autonomie und menschliche Aufsicht in Echtzeit, nicht erst an Stop-Punkten? Wer trägt die Verantwortung, wenn der Code von einem Modell stammt: die reviewende Person, die PO-Rolle, das Unternehmen? Erfindet sich das Daily als Abstimmung zwischen den prüfenden Menschen neu, oder verschwindet es? Wie verändert sich die Kultur eines Teams, das täglich mit Sprachmodellen arbeitet?

Diese Fragen löst keine Methode. Sie brauchen Erfahrung. Der Prozess aus diesem Buch ist ein Vorschlag, sie zu sammeln.

Die Grundregel bleibt: Je autonomer die KI, desto strenger müssen die menschlichen Stop-Punkte sein. Wer das vergisst, baut Software, die niemand verantwortet, und solche Software lebt nur Wochen.

## Was bleiben wird

Drei Sätze bleiben.

KI braucht Leitplanken, und diese Leitplanken sind in vierzig Jahren Software Engineering entstanden. Das steht schon in Kapitel 2: Sie sind nicht deshalb richtig, weil sie alt sind. Sie sind richtig, weil sie genau die Fehlerklasse adressieren, die ein Sprachmodell produziert.

Die KI ist ein mächtiges Werkzeug, und deshalb bleibt der Mensch am Steuer.

Die KI liefert Senior-Qualität, wenn die Leitplanken stimmen und der Mensch am Ende entscheidet. Ohne den Senior kein Senior-Code.

## ANHANG

# Anhang

Sechs Anhänge, die ersten vier zum Nachschlagen, gedacht für den Moment, in dem der Prozess schon läuft und eine einzelne Frage offen ist.

Anhang A listet alle Skills mit Schritt, Aufgabe und Grenze. Anhang B beschreibt die beiden Config-Dateien und die Umgebungsvariablen des Boards. Anhang C erklärt die Begriffe, die in diesem Buch eine feste Bedeutung haben, und ein paar davon weichen von der üblichen Verwendung ab. Anhang D nennt die Quellen und die Werkzeuge aus Kapitel 20. Anhang E sagt, wie dieses Buch entstanden ist, und Anhang F, wem dabei zu danken ist.

Für Konfigurationsfragen gilt auch hier: Die Online-Dokumentation ist aktueller als dieses Buch, und im Zweifel hat sie recht.

ANHANG A

# Skill-Referenz

Alle Skills auf einen Blick, mit Schritt, Akteur und der Grenze, die jeweils gilt. Ausführlich beschrieben sind sie in Kapitel 16.

## Kernprozess

| Skill            | Schritt | Was er tut                                    | Was er nicht tut                                                                               |
|------------------|---------|-----------------------------------------------|------------------------------------------------------------------------------------------------|
| /fachplan        | —       | Fachliches Issue im Story-Format, technikfrei | Kein technischer Plan, keine technischen Issues                                                |
| /plan            | 2       | Plan aus Anforderung oder aus #N              | Implementiert nichts, legt keine technischen Issues an                                         |
| /issues          | 3       | Plan in kleinteilige Issues schneiden         | Zieht nichts nach Ready                                                                        |
| /implement-ready | 5       | Ganze Ready-Spalte abarbeiten                 | Pusht nie, zieht nichts nach Ready, startet nichts für [Fachlich], [Idee], [Plan]              |
| /implement-test  | 5       | Nur Tests zu einem Issue, rot                 | Kein Produktionscode, kein Commit                                                              |
| /implement-done  | 5       | Gegen die roten Tests bis grün                | Pusht nie                                                                                      |
| /implement-next  | 5       | Genau ein Issue, dann Ende                    | Kein zweites Issue, auch wenn Ready gefüllt ist; startet nichts für [Fachlich], [Idee], [Plan] |
| /local-check     | 6       | buildChecks und mutationCommand ausführen     | Übergeht keinen roten Check                                                                    |

| Skill                          | Schritt | Was er tut                                                                                | Was er nicht tut                          |
|--------------------------------|---------|-------------------------------------------------------------------------------------------|-------------------------------------------|
| <code>/review</code>           | 7       | Review in frischer Session, zweites Modell                                                | Ersetzt keine Security-Werkzeuge im CI    |
| <code>/push-main</code>        | 8       | Commit-Batch auf main pushen, vorher <code>RELEASING.md</code> ausführen, falls vorhanden | Startet nie selbsttätig, setzt keinen Tag |
| <code>/merge-production</code> | 9       | PR von main nach production, gibt die Tag-Zeile aus                                       | Merged nicht selbst, setzt den Tag nicht  |

## Querschnitt

| Skill                      | Wann                             | Was er tut                                                                                                                             |
|----------------------------|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <code>/issue-review</code> | Zwischen Schritt 3 und dem Ready | Fachliches Issue, Plandokument oder Arbeitspaket durch andere Modelle prüfen lassen. Befunde als Kommentar, Marker nur nach Zustimmung |
| <code>/kontext</code>      | Session-Start                    | Vault, Projektnotiz und offene Issues laden, Lagebericht ausgeben                                                                      |
| <code>/document</code>     | Session-Ende                     | Tageslog schreiben, Projektnotiz aktualisieren                                                                                         |
| <code>/retro</code>        | Alle 1 bis 2 Wochen              | Drei Fragen stellen, Änderungen an Konventionsdateien vorschlagen                                                                      |

## Ohne Skill: die menschlichen Schritte

| Schritt          | Was                            | Wo                     |
|------------------|--------------------------------|------------------------|
| 1                | Anforderung formulieren        | Gespräch, Diktat, Mail |
| 4                | <b>Ready: Issues freigeben</b> | Am Board               |
| Zwischen 8 und 9 | Test-Server prüfen             | Im Browser             |

| Schritt | Was   | Wo              |
|---------|-------|-----------------|
| 9       | Merge | Im Pull Request |

## Der Nacht-Runner

Kein Skill, ein Node-Skript: `.claude/kit/night.mjs`. Im Implementierungs-Modus startet er pro Issue eine frische Session mit `/implement-next`. Details in Kapitel 19.

| Flag                                 | Wirkung                                                 |
|--------------------------------------|---------------------------------------------------------|
| <code>--dry-run</code>               | Zeigt an, was liefе, startet nichts                     |
| <code>--max &lt;N&gt;</code>         | Session-Limit pro Nacht                                 |
| <code>--model &lt;id&gt;</code>      | Modell für den Lauf                                     |
| <code>--timeout-min &lt;N&gt;</code> | Zeitlimit pro Runde                                     |
| <code>--label &lt;name&gt;</code>    | Routing-Label, <code>none</code> schaltet den Filter ab |
| <code>--verbose</code>               | Live-Verlaufsprotokoll                                  |
| <code>--no-checks-ok</code>          | Start trotz leerer <code>buildChecks</code>             |
| <code>--yolo</code>                  | Alle Freigaben übergehen, Einzelfall                    |

Im Review-Modus prüft derselbe Runner Dokumente aus dem Backlog, statt zu implementieren. Die Modi schließen sich aus.

| Flag                                               | Wirkung                                                      |
|----------------------------------------------------|--------------------------------------------------------------|
| <code>--review</code>                              | Prüfen statt implementieren, Zeitlimit fest bei 15 Minuten   |
| <code>--review-label &lt;name&gt;</code>           | Routing-Label, Default <code>kit:nightreview</code>          |
| <code>--stufe &lt;fachlich, plan, issue&gt;</code> | Welche Dokumente genommen werden, Default <code>issue</code> |

## Der Board-Adapter

`.claude/kit/board.mjs`. Alle Skills sprechen ausschließlich über ihn mit der Plattform. Details in Kapitel 14 und 18.

| Kommando                                       | Was es tut                                                                      |
|------------------------------------------------|---------------------------------------------------------------------------------|
| <code>issue create</code>                      | Issue anlegen                                                                   |
| <code>issue list</code>                        | Issues auflisten, mit <code>--status &lt;spalte&gt;</code> in Board-Reihenfolge |
| <code>issue get</code>                         | Ein Issue lesen, Kommentare inbegriffen                                         |
| <code>issue move</code>                        | Issue in eine andere Spalte schieben                                            |
| <code>issue comment</code>                     | Kommentar an ein Issue hängen                                                   |
| <code>issue update &lt;id&gt; --body</code>    | Body ersetzen                                                                   |
| <code>issue-review roles, matrix, check</code> | Rollen, Prüfer-Zuordnung und Reviewer-Stand ausgeben                            |
| <code>code repo-name</code>                    | Repository-Namen ermitteln                                                      |
| <code>code pr</code>                           | Pull Request erstellen                                                          |
| <code>--version</code>                         | Stand des Adapters im Projekt                                                   |

ANHANG B

# Config-Referenz

Drei Config-Dateien, drei Zuständigkeiten. `workflow.config.json` ist repo-spezifisch und wird geteilt. `workflow.config.local.json` hält die wenigen Werte, die im selben Projekt bei jedem anders sein dürfen. `kon-text.config.json` ist personenbezogen und gehört nicht ins Repo. Kapitel 15 und Kapitel 17 erklären die Hintergründe.

Diese Referenz gibt den Stand bei Redaktionsschluss wieder. Die gepflegte Fassung steht unter `docs.mwolff.org`.

## workflow.config.json

Liegt in `.claude/` im Projekt.

| Feld                          | Werte                                                                                          | Bedeutung                                                                                                                |
|-------------------------------|------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>codeHost</code>         | <code>github</code> ,<br><code>gitlab</code> , <code>local</code>                              | Wo Repository und Pull Requests liegen                                                                                   |
| <code>issueTracker</code>     | <code>github</code> ,<br><code>gitlab</code> ,<br><code>local</code> ,<br><code>toolbox</code> | Wo Issues und Board-Bewegungen liegen. <code>toolbox</code> ist der historische Name für das kanban-kit, siehe Kapitel 5 |
| <code>buildChecks</code>      | Liste von Kommandos                                                                            | Was <code>/local-check</code> nacheinander ausführt                                                                      |
| <code>mutationCommand</code>  | Kommando oder <code>" "</code>                                                                 | Mutation Testing, separat wegen Laufzeit                                                                                 |
| <code>formatFixCommand</code> | Kommando oder fehlt                                                                            | Wird bei roten Checks im Nachtlauf genau einmal versucht, danach werden die Checks einmal wiederholt                     |
| <code>mainBranch</code>       | Branch-Name                                                                                    | Ziel von <code>/push-main</code>                                                                                         |
| <code>productionBranch</code> | Branch-Name                                                                                    | Ziel von <code>/merge-production</code>                                                                                  |

| Feld                              | Werte                                  | Bedeutung                                                                                                       |
|-----------------------------------|----------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>reviewScope</code>          | <code>diff</code> , <code>full</code>  | Umfang, den <code>/review</code> zu sehen bekommt                                                               |
| <code>reviewModel</code>          | Modell-ID                              | Modell für die Review-Session                                                                                   |
| <code>triggers.ready</code>       | Phrase                                 | Freigabe zur Implementierung                                                                                    |
| <code>triggers.push</code>        | Phrase                                 | Freigabe zum Push                                                                                               |
| <code>triggers.merge</code>       | Phrase                                 | Freigabe für den PR nach production                                                                             |
| <code>toolbox.host</code>         | URL                                    | Adresse der kanban-kit-Instanz                                                                                  |
| <code>toolbox.ideaStored</code>   | <code>true</code> , <code>false</code> | Ob <code>/fachplan</code> neue Issues in den Ideen-Pool legt statt ins Backlog. Ohne Eintrag <code>false</code> |
| <code>toolbox.tokenFile</code>    | Pfad                                   | Datei mit dem Token, gitignored. Nie den Token selbst eintragen                                                 |
| <code>columns</code>              | Objekt                                 | Anzeigenamen der fünf Spalten. Schlüssel fest, Werte frei                                                       |
| <code>github.projectNumber</code> | Zahl                                   | Nummer des Project Boards. Wird erkannt, solange es genau eines gibt                                            |
| <code>local.issuesDir</code>      | Pfad                                   | Verzeichnis der lokalen Issue-Dateien, Default <code>issues</code>                                              |

Der `toolbox`-Block greift nur bei `"issueTracker": "toolbox"`.

Die fünf Schlüssel unter `columns` sind fest: `backlog`, `ready`, `in_progress`, `in_review`, `done`. Die Werte sind die Namen, die am Board stehen, und dürfen frei gewählt werden. Sie müssen aber mit dem Board übereinstimmen, Groß- und Kleinschreibung eingeschlossen. Eine abweichende Fassung führt nicht zu einer Fehlermeldung, sondern zu einer leeren Issue-Liste, und danach sucht man lange.

```
"columns": {
  "backlog":      "Backlog",
  "ready":        "Ready",
  "in_progress":  "In progress",
  "in_review":    "In review",
  "done":         "Done"
}
```

Neben der eingeecheckten Datei kann eine gitignorete `.claude/work-flow.config.local.json` liegen. Sie überschreibt genau vier Felder: `reviewModel`, `reviewScope`, `triggers` und `toolbox.tokenFile`. Alles andere darin wird ignoriert, mit Meldung auf stderr. Kapitel 15 begründet, warum.

Beispiel:

```
{
  "codeHost": "github",
  "issueTracker": "local",
  "buildChecks": ["mvn verify"],
  "mutationCommand": "mvn org.pitest:pitest-
maven:mutationCoverage",
  "mainBranch": "main",
  "productionBranch": "production",
  "reviewScope": "diff",
  "reviewModel": "<modell-id>",
  "triggers": { "ready": "ready", "push": "push main", "merge":
"merge production" },
  "toolbox": { "host": "https://kanban.example.org",
"ideaStored": false }
}
```

Ältere Repos mit `"provider": "github"` funktionieren weiter, der Adapter migriert das Feld beim Lesen auf `codeHost` und `issueTracker`.

Ein Klartext-Token gehört nicht in diese Datei. Sie ist eingeecheckt und wird geteilt.

`buildChecks` und `formatFixCommand` sind frei geschriebene Kommandozeilen und laufen deshalb in einer Shell: unter macOS und Linux in

`/bin/sh`, unter Windows in der ComSpec-Shell, im Regelfall `cmd.exe`. Wer sie plattformübergreifend braucht, hält sie einfach und verzichtet auf `&&`, Pipes und Umleitungen.

## issueReview und reviewStufen

Zwei optionale Blöcke in derselben Datei, beschrieben in Kapitel 9. Ohne den `issueReview`-Block tut `/issue-review` nichts, und der Nacht-Review bricht im Vorflug ab. Der Installer legt eine Vorlage zum Abschreiben daneben, `.claude/workflow.config.example.json`, und schreibt den Block bewusst nicht selbst: `reviewers` hängt davon ab, welche CLIs auf der Maschine liegen, und `pairs` ist eine Entscheidung, keine Voreinstellung.

| Feld                                         | Werte                                     | Bedeutung                                                                                   |
|----------------------------------------------|-------------------------------------------|---------------------------------------------------------------------------------------------|
| <code>issueReview.rounds</code>              | Zahl                                      | Prüfrunden pro Dokument. Bleibt bei <code>1</code>                                          |
| <code>issueReview.requiredBeforeReady</code> | <code>true</code> ,<br><code>false</code> | Ob ein Ready-Issue die Zeile <code>Issue-Review:</code> braucht. Default <code>false</code> |
| <code>issueReview.reviewers</code>           | Liste                                     | Die verfügbaren Prüfer, siehe unten                                                         |
| <code>issueReview.pairs</code>               | Objekt                                    | Wer wen prüft, nach Autor-Modell                                                            |
| <code>reviewStufen</code>                    | Objekt                                    | Anzahl Prüfer und Rollen je Stufe                                                           |

```

"issueReview": {
  "rounds": 1,
  "requiredBeforeReady": false,
  "reviewers": [
    { "name": "opus", "kind": "claude", "model": "claude-opus-5" },
    { "name": "sonnet", "kind": "claude", "model": "claude-sonnet-5" },
    { "name": "codex", "kind": "command", "cmd": "codex exec" }
  ],
  "pairs": {
    "opus": ["sonnet", "codex"],
    "sonnet": ["opus", "codex"]
  }
},
"reviewStufen": {
  "fachlich": { "reviewer": 2, "rollen": ["form-beobachtbarkeit", "abgrenzung"] },
  "plan": { "reviewer": 2, "rollen": ["architektur-bestand", "schnitt-abhaengigkeiten"] },
  "issue": { "reviewer": 1, "rollen": ["pruefbarkeit"] }
}

```

Ohne `reviewStufen` bleibt es bei der alten Besetzung: zwei Prüfer auf jeder Stufe, mit den Rollen `vollstaendigkeit-pruefbarkeit` und `scope-risiko-bestand`. Erst ein ausdrücklich geschriebener Block aktiviert die neue Aufteilung. Ein Kit-Update ändert das Prüfverfahren also nicht im Vorbeigehen.

**Zu `pairs`.** Wer der Autor eines Issues war, steht als Zeile `Autor-Modell:` in dessen Kontext-Abschnitt. Findet sich dieser Name in `pairs`, gewinnt sein Eintrag. Sonst greift eine Regel: die vordersten Prüfer, die nicht der Autor sind. Wie viele davon laufen, kürzt anschließend die Stufe.

Auf diese Regel sollte man sich nicht verlassen. Sie nimmt immer die vordersten Einträge, und bei vier konfigurierten Prüfern kommt der vierte in keinem einzigen Fall zum Zug. Wer ein fremdes Modell hinten in die Liste schreibt, hat es damit abgeschaltet. Genau das ist mir beim

Bauen des Verfahrens passiert, und es ist der Grund, warum es `pairs` überhaupt gibt.

Nachsehen statt ausrechnen:

```
node .claude/kit/board.mjs issue-review matrix
node .claude/kit/board.mjs issue-review roles --stufe issue --
author claude-opus-5
```

Die Ausgabe von `matrix` nennt in einer eigenen Spalte, ob eine Zeile aus `pairs` stammt oder aus der Regel. Wer dort `regel` liest, obwohl er einen Eintrag erwartet hatte, hat den Autor-Namen anders geschrieben.

Zwei Dinge sind harte Fehler und werden nicht still übergangen: ein Name in `pairs`, den es in `reviewers` nicht gibt, und ein Autor, der sich selbst als Prüfer nennt.

## kontext.config.json

Global in `~/claude/`, optional lokal in `.claude/`. Beide werden gemerget, lokale Werte gewinnen.

| Feld                     | Typ    | Bedeutung                                                                                                                                                       |
|--------------------------|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>vault</code>       | Pfad   | Absoluter Pfad zum Memory-Vault. Fehlt er, läuft der Degraded Mode                                                                                              |
| <code>always</code>      | Liste  | Dateien relativ zum Vault, die immer gelesen werden                                                                                                             |
| <code>projectDocs</code> | Liste  | Dateien oder Glob-Muster relativ zum Projekt                                                                                                                    |
| <code>project</code>     | Text   | Override, wenn Repo-Name und Vault-Ordner abweichen                                                                                                             |
| <code>logPath</code>     | Muster | Wohin <code>/document</code> den Tageslog schreibt. Default <code>Log/{date}.md</code> . Bei mehreren Repos an einem Vault <code>Log/{date}-{project}.md</code> |

```
{
  "vault": "/pfad/zum/vault",
  "always": ["Index.md", "Profil.md"],
  "projectDocs": ["CLAUDE-*", ".claude/CLAUDE-*"]
}
```

Erwartete Vault-Struktur:

```
Index.md
Profil.md
Projekte/{repo-name}/{repo-name}.md
Log/YYYY-MM-DD.md           bzw. was in logPath steht
```

Die Struktur wird einmal von Hand angelegt. `/document` pflegt danach die Projektnotiz selbst, für einen Eintrag in `Index.md` fragt er nach.

## kanban-kit: Umgebungsvariablen

In einer `.env` neben der `docker-compose.yml`. Details in Kapitel 14.

| Variable                                  | Default                        | Bedeutung                                |
|-------------------------------------------|--------------------------------|------------------------------------------|
| <code>MANBAN_BASE_URL</code>              | <code>https://localhost</code> | Basis-URL für Links in E-Mails           |
| <code>MANBAN_BOOTSTRAP_ADMIN_TOKEN</code> | <code>leer</code>              | Einmal-Token für den ersten Admin        |
| <code>MANBAN_MAIL_ENABLED</code>          | <code>false</code>             | Echten Mailversand aktivieren            |
| <code>MANBAN_CLEANUP_ENABLED</code>       | <code>true</code>              | Done-Archivierung und Papierkorb-Leerung |
| <code>MANBAN_DONE_RETENTION_DAYS</code>   | <code>30</code>                | Tage bis Done-Karten archiviert werden   |
| <code>MANBAN_SESSION_SECRET</code>        | Dev-Default                    | In Produktion zwingend setzen            |

| Variable             | Default | Bedeutung                             |
|----------------------|---------|---------------------------------------|
| MANBAN_COOKIE_SECURE | true    | Session-Cookie nur über HTTPS         |
| MANBAN_DOMAIN        | leer    | Echte Domain, aktiviert Let's Encrypt |

## Die fünf Spalten

Für GitHub Projects und kanban-kit als Spalten, für GitLab als Labels. Die Namen sind case-sensitiv.

Backlog · Ready · In progress · In review · Done

## ANHANG C

# Glossar

**Akzeptanzkriterium** Der dritte Abschnitt eines Issues. Beschreibt konkret, messbar oder ausführbar, wie verifiziert wird, dass die Aufgabe erledigt ist.

**ArchUnit** Bibliothek, mit der Architekturregeln als automatische Tests formuliert werden. Verletzt der Code eine Schichtengrenze, bricht der Build.

**Bahn 1** Kleine Änderung: eine Datei, ein Asset oder ein Config-Wert, kein Datenmodell, keine Migration, kein Endpoint, keine Sicherheitslogik, höchstens ein Modul. Wird direkt umgesetzt, ein Commit, ohne Plan, Issue und Ready. Der Push bleibt an die Trigger-Phrase gebunden.

**Bahn 2** Alles andere, also der volle Prozess aus Kapitel 8. Im Zweifel gilt Bahn 2.

**Board-Adapter** `board.mjs`. Schirmt die Skills von der konkreten Plattform ab. Kennt im Kern Issue anlegen, auflisten, lesen, verschieben, kommentieren, dazu Pull Requests und das Issue-Review. Für das kanban-kit delegiert er an `tbx.mjs`.

**buildChecks** Feld in der `workflow.config.json`. Liste der Kommandos, die `/local-check` ausführt. Die Stelle, an der die Qualitätssicherung im Prozess verankert ist.

**Context Drift** Das Nachlassen der Wirkung früher Anweisungen im Lauf einer Session. Das Modell vergisst nichts, es gewichtet seine Aufmerksamkeit um. Die Unterscheidung ist praktisch relevant: Wiederholen hilft wenig, Kontext aufräumen hilft, ein Neustart hilft zuverlässig.

**Continuous AI** Hintergrund-Loops, in denen Agenten kontinuierlich kleine Verbesserungen erzeugen, meist als kleine Pull Requests. Ergänzung der menschlich getakteten Iteration, kein Ersatz.

**Degraded Mode** Betrieb von `/kontext` und `/document` ohne konfigurierbaren Vault. Funktioniert vollständig, nur ohne projektübergreifendes Memory.

**GO** Der frühere Name für das Ready. Bis Version 5 hieß der erste Stop-Punkt so, und die voreingestellte Trigger-Phrase im Kit trug denselben Namen.

**Fachliches Issue** Issue mit `[Fachlich]`-Präfix. Beschreibt in PO-Sprache das Was und Warum, technikfrei. Wird gegroomt, nie implementiert. Siehe PO-Schleife.

**Idee** Issue mit `[Idee]`-Präfix. Rohe, ungesichtete Notiz ohne festes Format, von Hand angelegt. Wird nie implementiert. Aus ihr wird erst etwas, wenn `/fachplan` oder `/plan` sie anfasst.

**Ideen-Pool** Zone im kanban-kit für Karten, die noch nicht auf dem Board sichtbar sein sollen. Liegt vor, noch nicht eingeplant. Ob `/fachplan` seine Issues dort ablegt, steuert `toolbox.ideaStored`.

**Implementations-Engine** Bezeichnung für die Rolle des Sprachmodells im Prozess. Ein leistungsstarker Motor, der eine Hand am Steuer braucht.

**Issue** Die Arbeitseinheit und die Quelle der Wahrheit. Vier Abschnitte: Kontext, Aufgabe, Akzeptanzkriterium, Abhängigkeiten. Kleinteilig genug, um eigenständig getestet zu werden.

**Issue-Review** Prüfung des Dokuments, gegen das implementiert wird, durch andere Modelle. Drei Stufen je nach Titel-Präfix: `fachlich`, `plan`, `issue`. Jede Stufe hat eigene Rollen und hinterlässt eine eigene Nachweiszeile. Von den dreien gibt nur `Issue-Review` die Umsetzung frei, daneben der Prüfungsverzicht.

**Kontextfenster** Der Textbereich, den ein Sprachmodell in einer Anfrage verarbeitet. Begrenzt, und je voller er wird, desto weniger Gewicht haben frühe Inhalte.

**Leitplanke** Sammelbegriff für die Praktiken, die verhindern, dass ein Sprachmodell in die falsche Richtung läuft: Tests, Architekturregeln,

Konventionen, Reviews, statische Analyse. Eine Leitplanke muss scheitern können, nicht argumentieren.

**Mutation Score** Anteil der künstlich eingebauten Codefehler, die von den Tests gefunden werden. Der harte Wert für Testgüte, im Gegensatz zur Coverage.

**Nacht-Runner** `night.mjs`. Arbeitet unbeaufsichtigt, mit einer frischen Session pro Issue. Im Implementierungs-Modus die Ready-Spalte, im Review-Modus Dokumente aus dem Backlog. Pusht nie.

**Plandokument** Issue mit `[Plan]`-Präfix, von `/plan` nach der Freigabe in Bahn 2 angelegt. Trägt den Plantext als Body, in sechs festen Überschriften. Wird nie implementiert, sondern mit `/issues #N` zerlegt.

**PO-Schleife** Optionale Trennung in fachliche und technische Issues, damit ein Product Owner fachlich abnehmen kann, bevor Technik entsteht.

**Prüfstufe** Eine der drei Ebenen des Issue-Reviews. Welche greift, liest der Skill am Titel-Präfix ab. Die Prüfung wandert nach oben, weil ein Fehler nach unten an Reichweite gewinnt.

**Prompt-Bibliothek** Im Repository gepflegte, wiederverwendbare Anweisungen an das Modell. Veraltet schnell und gehört regelmäßig inspiziert.

**Prüfungsverzicht** Zeile `Pruefung: Verzicht` im Kontext-Abschnitt eines Issues. Gibt es ohne Review frei und wirkt wie ein Marker. Erlaubt sind auch `1`, `2`, `3` für die Zahl der Runden. Setzen darf das nur ein Mensch, der Verzicht verfällt, sobald sich der Body ändert.

**Ready** Zweierlei, und beides meint denselben Moment. Die Spalte, in der ein Issue implementierbar ist, der Kern des Boards. Und der erste Stop-Punkt, Schritt 4: die menschliche Freigabe zur Implementierung, am Board sichtbar als Bewegung dorthin. Bewegungen nach Ready macht ausschließlich der Mensch. Nicht zu verwechseln mit der Zustimmung zu einem Plan.

**RELEASING.md** Optionale Datei im Projekt-Root. `/push-main` und `/merge-production` lesen sie bei jedem Lauf und führen die dort beschriebenen Release-Schritte aus. Eine Konvention, kein Kit-Feature.

**Review-Nacht** Zweiter Modus des Nacht-Runners (`--review`). Prüft Dokumente aus dem Backlog, statt zu implementieren. Fährt genau eine Prüfstufe, Zeitlimit fest bei 15 Minuten, bewegt kein Issue am Board. Schreibt nie den Body.

**Routing-Label** Standardmäßig `kit:nightrun`. Markiert auf einem Board die Teilmenge der Ready-Issues, die der Nacht-Runner bearbeiten darf. Die Review-Nacht hat ein eigenes, `kit:nightreview`, weil sie eine andere Spalte meint.

**Salvage** Rettungsversuch vor dem harten Stopp, wenn eine Nacht-Runde einen unsauberen Worktree hinterlässt. Der Runner fährt die `build-Checks` selbst; sind sie grün, bekommt genau eine Session zehn Minuten, um zu committen und das Board zu bewegen.

**Stop-Punkt** Eine der drei Stellen, die bewusst nicht automatisiert sind: das Ready, der Push, der Merge. Der Git-Tag ist ein vierter, impliziter, ohne Trigger-Phrase.

**Subagent** Eine vom Modell selbst gestartete Kindsession über das Agent-Tool. Reviewer mit `kind: "claude"` laufen so und brauchen keine Freigabe in der Allowlist, Reviewer mit `kind: "command"` laufen über Bash und brauchen sie.

**Taskboard** Das hier benutzte Board. Kein Kanban-Board im strengen Sinn, weil Issues nach Ready geschoben und nicht gezogen werden und weil es keine WIP-Limits gibt.

**Trigger-Phrase** Vereinbarte Formulierung, die einen der drei Stop-Punkte auslöst. Eine Sicherheitsarchitektur, keine Bedienungsanleitung.

**Vault** Memory-Speicher außerhalb des Repos für projektübergreifendes Wissen. Er gehört zum Projekt, nicht der Person; personenbezogen ist nur der Pfad dorthin, und der steht in `kontext.config.json`. Optional.

**Vorflug** Gate vor der ersten Session einer Review-Nacht. Prüft in einer eigenen Session, ob alle Reviewer laufen und der Tracker erreichbar ist. Abschalten lässt sich der Vorflug nicht. Jeder Befund nennt die Umgebung, in der er entstanden ist.

**YAGNI** “You Ain’t Gonna Need It”. Nichts auf Vorrat bauen. Im KI-Zeitalter wichtiger, weil das Hinzufügen fast nichts mehr kostet und die Hürde damit von Aufwand auf Disziplin wechselt.

## ANHANG D

# Quellen und weiterführende Links

## Die Werkzeuge

- **claude-workflow-kit**, Dokumentation und Installer:  
[docs.mwolff.org](https://docs.mwolff.org) Quellcode: [github.com/mannewolff/claude-workflow-kit](https://github.com/mannewolff/claude-workflow-kit)
- **kanban-kit**, Dokumentation: [kanban.mwolff.org/docs](https://kanban.mwolff.org/docs) Quellcode:  
[github.com/mannewolff/kanban-kit](https://github.com/mannewolff/kanban-kit)

Beide sind Open Source. Die Online-Dokumentation ist die gepflegte Referenz und in Konfigurationsfragen aktueller als dieses Buch.

## Eigene Texte

- Whitepaper “Ein Prozess zur KI-gestützten Softwareentwicklung”, [mwolff.org/whitepapers](https://mwolff.org/whitepapers) Die konzeptionelle Grundlage von Teil I. Wer den Prozess ohne Werkzeugbezug lesen will, findet ihn dort.
- Whitepaper “Konzeptpapier zur DSGVO-konformen AI-Coding-Umgebung”, [mwolff.org/whitepapers](https://mwolff.org/whitepapers)
- Blog: [blog.mwolff.org](https://blog.mwolff.org). Viele der Fälle in diesem Buch sind dort ausführlicher beschrieben.

## Grundlagen

- Beck, Kent et al.: *Manifesto for Agile Software Development*, 2001.  
[agilemanifesto.org](https://agilemanifesto.org)
- PMI und Agile Alliance: *Manifesto for Enterprise Agility*, 2026.

## Forschung

- *The AI-Native Software Development Lifecycle: A Theoretical and Practical New Methodology*, arXiv 2408.03416, 2024. Quelle des V-Bounce-Modells.

- *Agentsway: Software Development Methodology for AI Agents-based Teams*, arXiv 2510.23664, 2025. Quelle der Rolle des Human Orchestrator.
- *AI and Agile Software Development: A Research Roadmap from the XP2025 Workshop*, arXiv 2508.20563, 2025. Quelle der KI-Assistenzrollen und der eigenständigen KI-Retrospektive.
- *Continuous AI in practice: What developers can automate today with agentic CI*, GitHub Blog, 2026.
- *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity*, METR, Juli 2025. arXiv 2507.09089, [metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study](https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study) Die Studie aus der Einleitung: neunzehn Prozent langsamer, gefühlt zwanzig Prozent schneller. METR selbst führt das Ergebnis inzwischen als historisch, weil es die Werkzeuglage von Anfang 2025 abbildet. Als Beleg für die Wahrnehmungslücke taugt es weiterhin, als Beleg für heutige Produktivität nicht.
- Ryseff, James; De Bruhl, Brandon F.; Newberry, Sydne J.: *The Root Causes of Failure for Artificial Intelligence Projects and How They Can Succeed. Avoiding the Anti-Patterns of AI*, RAND Corporation, 2024. [rand.org/pubs/research\\_reports/RRA2680-1.html](https://rand.org/pubs/research_reports/RRA2680-1.html) Die Quelle für die Zahl aus der Einleitung: Nach den dort referierten Schätzungen scheitern mehr als achtzig Prozent der KI-Vorhaben, doppelt so viele wie bei IT-Projekten ohne KI. Grundlage sind Interviews mit 65 erfahrenen Data Scientists und Engineers.
- Gartner: *Gartner Predicts 30% of Generative AI Projects Will Be Abandoned After Proof of Concept By End of 2025*, Juli 2024. Die vorsichtigere Zahl derselben Debatte, und ein nützlicher Gegenpol: Je nachdem, was als Scheitern gezählt wird, liegt der Wert zwischen dreißig und achtzig Prozent.

## Werkzeuge aus Kapitel 20

Genannt als Ausgangspunkt, nicht als Rangliste. Die Kategorien sind stabiler als die Namen.

| Kategorie                | Werkzeuge                                                                                            |
|--------------------------|------------------------------------------------------------------------------------------------------|
| Gesamtqualität           | SonarQube, SonarCloud                                                                                |
| Secrets                  | gitleaks, TruffleHog                                                                                 |
| Injection und Datenfluss | Semgrep, SpotBugs mit FindSecBugs                                                                    |
| Statische Analyse        | PMD, ErrorProne, SpotBugs, ESLint                                                                    |
| Code-Style               | Spotless, Checkstyle, Prettier                                                                       |
| Architektur              | ArchUnit, ArchUnitNET, PyTestArch, Deptrac, phpat, dependency-cruiser                                |
| Testgüte                 | PIT, Stryker                                                                                         |
| Abhängigkeiten           | OWASP Dependency-Check, Trivy, <code>npm audit</code>                                                |
| Typen                    | <code>tsc</code> , Vite-Build                                                                        |
| Veraltete APIs           | <code>@typescript-eslint/no-deprecated</code> , <code>-Xlint:deprecation</code> <code>-Werror</code> |

## Zu den Studien in Kapitel 20

- *When AI Reviews Its Own Code: Recursive Self-Training Collapse in Code LLMs*, Emory University und Universität Tokio, Juni 2026. arXiv 2606.28438 Die Selbst-Trainings-Studie: Ohne Prüfung fällt die Trefferquote am schnellsten, mit Compiler und statischen Prüfregeleln langsamer. Bewertet das Modell den eigenen Code selbst, steigen die Selbstnoten, während die tatsächliche Trefferquote fällt.
- Alemohammad, Sina et al.: *Self-Consuming Generative Models Go MAD*, Rice University, ICLR 2024. arXiv 2307.01850 Prägt den Begriff Model Autophagy Disorder für selbstkonsumierende generative Modelle in der Bildforschung.
- GitClear: *The Maintainability Gap: 2026 AI Code Quality Research*, 2026. [gitclear.com/the\\_ai\\_code\\_quality\\_maintainability\\_gap](https://gitclear.com/the_ai_code_quality_maintainability_gap) Quelle der Zahlen zu verschobenem und kopiertem Code. Die Werte stehen frei auf der Seite, das PDF gibt GitClear nur gegen Namen und

Geschäftsadresse heraus. GitClear verkauft Werkzeuge zur Codeanalyse, es sind also Anbieterzahlen. Die frei herunterladbare Vorgängerfassung reicht bis 2024 und misst dieselben Kennzahlen an 211 Millionen geänderten Codezeilen: Harding, William: *AI Copilot Code Quality*, Februar 2025.

- Huang, Haoming et al.: *More Code, Less Reuse. Investigating Code Quality and Reviewer Sentiment towards AI-generated Pull Requests*, Januar 2026. arXiv 2601.21276 Die unabhängige Gegenprobe zu GitClear: Agenten lassen Gelegenheiten zur Wiederverwendung liegen, die Redundanz steigt, und Reviewer bewerten die Beiträge dabei neutraler bis positiver als menschliche.
- Griffiths, Andrea: *Agent pull requests are everywhere. Here's how to review them.*, GitHub Blog, Mai 2026. [github.blog](https://github.blog) Quelle für die Verbreitung agentischer Beiträge: An mehr als jedem fünften Code-Review auf GitHub ist ein Agent beteiligt.

## Zu den Studien in Kapitel 4

- Hosseini Maasoum, Seyed Mahdi; Lichtinger, Guy: *Generative AI as Seniority-Biased Technological Change. Evidence from U.S. Résumé and Job Posting Data*, Harvard, August 2025. SSRN 5425555, [papers.ssrn.com/sol3/papers.cfm?abstract\\_id=5425555](https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5425555) Datenbasis: 62 Millionen Beschäftigte in 285.000 Unternehmen, 2015 bis 2025.
- Brynjolfsson, Erik; Chandar, Bharat; Chen, Ruyu: *Canaries in the Coal Mine? Six Facts about the Recent Employment Effects of Artificial Intelligence*, Stanford Digital Economy Lab, 2025. [digitaleconomy.stanford.edu](https://digitaleconomy.stanford.edu) Datenbasis: Gehaltsabrechnungsdaten von ADP.

Zahlen zum deutschen Stellenmarkt habe ich für dieses Buch bewusst weggelassen. Ich habe keine Erhebung gefunden, die ich mit Titel und Jahr belegen kann, und eine Zahl ohne Quelle gehört nicht in ein Buch, das von Verifikation handelt. Aus demselben Grund ist jede Quelle in diesem Anhang frei zugänglich. Eine Zahl, die nur hinter einem Abonnement steht, kann niemand nachprüfen, der dieses Abonnement nicht hat.

## ANHANG E

# Über die Entstehung dieses Buches

Dieses Buch ist nach derselben Methodik entstanden, die es beschreibt: in Ko-Produktion mit einem KI-Werkzeug, konkret mit Claude von Anthropic.

Ich habe Struktur, Kernthesen und alle inhaltlichen Entscheidungen verantwortet. Claude hat die ersten Formulierungen erzeugt, ich habe sie redigiert, präzisiert und an mehreren Stellen ganz verworfen. Die Kontrolle über das Ergebnis bleibt damit beim Autor: Das KI-Werkzeug arbeitet zu, aber es signiert nicht mit.

Der Prozess aus diesem Buch war dabei kein Vorbild, sondern Arbeitsmittel. Es gab einen Plan, bevor eine Zeile stand. Es gab Teile, die in einem Rutsch entstanden, und einen Lektoratsdurchgang über das fertige Manuskript, der Dopplungen, Widersprüche und tote Querverweise gefunden hat, die ich beim Schreiben einzelner Teile nicht sehen konnte. Es gab eine Liste mit Stellen, an denen ein Beispiel erfunden statt erlebt war, und diese Liste wurde abgearbeitet, bis sie leer war. Der Marker, der eine solche Stelle kenntlich macht, ist im Buchsatz sichtbar hinterlegt, damit sie nicht im Fließtext verschwindet.

Zwei Dinge sind mir dabei aufgefallen, und beide stehen so auch im Buch.

Erstens: Was ich nicht selbst erlebt habe, kann ein Modell plausibel erfinden, und es sieht genauso aus wie das Erlebte. Deshalb war die Trennung zwischen beidem der wichtigste redaktionelle Schritt. Mehrere Zahlen und Anekdoten in früheren Fassungen waren gut erfunden und mussten raus.

Zweitens: Auch dieser Text braucht Leitplanken, die scheitern können. Bei Software sind das Tests. Bei einem Buch sind es Prüfungen, die man automatisieren kann: eine Liste von Wörtern, die ich nicht benutze, ein Abgleich aller Querverweise gegen die Kapitel, auf die sie zeigen, eine

Suche nach Formulierungen, die nur die Hälfte der Leserschaft meinen. Jede dieser Prüfungen hat etwas gefunden.

Diese Arbeitsweise habe ich an anderer Stelle ausführlicher beschrieben: [blog.mwolff.org/warum-ich-meine-rohtexte-nie-mehr-ohne-ki-bearbeite-und-trotzdem-die-kontrolle-behalte/](https://blog.mwolff.org/warum-ich-meine-rohtexte-nie-mehr-ohne-ki-bearbeite-und-trotzdem-die-kontrolle-behalte/)

## ANHANG F

# Dank

Ein Buch über einen Prozess, der von Rückmeldung lebt, wäre ohne Rückmeldung nicht entstanden.

**Für den Anstoß.** Heiko Dietz, meinem damaligen Chef, der die Sache anders beurteilt hat als ich. Ohne die Artikel, die er mir geschickt hat, hätte ich die Frage nicht gestellt, aus der alles Weitere folgte. Dass er mir seine Gründe für dieses Buch noch einmal aufgeschrieben hat, obwohl wir in der Sache nie zusammengekommen sind, rechne ich ihm hoch an. Sein zweiter Einwand steht unverändert, und er steht an vielen Stellen dieses Buches zwischen den Zeilen.

**Für den Beweis, dass es geht.** Eduard Andrae, der seine Plattform *trusted blogs* 2025 und 2026 vollständig neu gebaut hat, ausschließlich mit KI. Er hat mir Anfang Mai 2026 gezeigt, dass auch fachlich komplexe Anwendungen so entstehen können. Ich habe ihm das damals schon geschrieben, und es gilt weiterhin.

**Für die Praxis.** Dem IT-Bildungshaus, wo ich im August 2026 den Workshop zum Kit halte. Wer einen Prozess unterrichten soll, muss ihn zu Ende denken, und mehrere Stellen dieses Buches sind schärfer, weil ich sie jemandem erklären musste, der sie am nächsten Tag anwenden wollte.

**Für den Widerspruch.** Vieles hier ist schärfer geworden, weil jemand widersprochen hat, meistens auf LinkedIn, manchmal unter meinen Blogposts. Namen nenne ich bewusst keine. Es wären zu viele, und ich möchte niemanden für ein Buch vereinnahmen, der mir widersprochen hat. Kapitel 1 verdankt diesen Diskussionen am meisten: Die Unterscheidung zwischen Agilität und Scrum habe ich erst deshalb so deutlich formuliert, weil mich jemand zu Recht darauf gestoßen hat, dass ich sie vorher unscharf gelassen hatte.

**Für das Lektorat.** Gabi Rosenbaum, die den Text gelesen hat, bevor er gedruckt wurde. Wer den eigenen Text zum dritten Mal liest, sieht ir-

gendwann nicht mehr, was dort steht, sondern was dort stehen sollte. Jede Stelle, über die beim Lesen niemand stolpert, kann daran liegen, dass vorher jemand anderes darüber gestolpert ist.

**Für die Werkzeuge.** Die Projekte, auf denen beide Kits aufsetzen, sind Open Source und werden von Menschen gepflegt, die davon nichts haben außer der Arbeit: git, Node, Docker, Spring Boot, Postgres, ArchUnit, PIT und die vielen kleineren Bibliotheken dazwischen.

**Und zuletzt** an alle, die den Prozess ausprobieren und mir sagen, wo er nicht trägt. Der Prozess ist eine Momentaufnahme, und er wird besser, wenn Erfahrungen zurückkommen. Die KI-Retrospektive aus Kapitel 23 gilt auch für dieses Buch.

**Software mit KI entwickeln** Das kanban-kit und das claude-workflow-kit in der Praxis

1. Auflage, 2026

© 2026 Manfred Wolff. Alle Rechte vorbehalten.

**Autor und inhaltlich verantwortlich** Manfred Wolff Butjadinger Straße 34a 28197 Bremen

[info@mwolff.org](mailto:info@mwolff.org) [mwolff.org](http://mwolff.org) · [blog.mwolff.org](http://blog.mwolff.org)

**Herstellung** epubli, ein Service der neopubli GmbH, Berlin

Private Auflage ohne ISBN, nicht für den Buchhandel bestimmt.

**Zu den Werkzeugen** Das claude-workflow-kit und das kanban-kit sind frei verfügbar. Dokumentation: [docs.mwolff.org](http://docs.mwolff.org) und [kanban.mwolff.org](http://kanban.mwolff.org) Quellcode: [github.com/mannewolff](https://github.com/mannewolff)

**Zur Herstellung dieses Buches** Der Text ist in Markdown geschrieben und wird über ein eigenes Build-Skript nach HTML und von dort mit WeasyPrint nach PDF gesetzt. Schrift: Carlito, Auszeichnungen in DejaVu Sans Mono. Wie das Buch inhaltlich entstanden ist, steht in Anhang E.

Die in diesem Buch genannten Marken- und Produktnamen sind Eigentum der jeweiligen Rechteinhaber. Alle Angaben erfolgen nach bestem Wissen, jedoch ohne Gewähr. Eine Haftung für Schäden, die aus der Anwendung der beschriebenen Verfahren entstehen, ist ausgeschlossen.

