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 1.4 · August 2026

KI-gestützte Entwicklung · kanban-kit · claude-workflow-kit · Agile · mwolff.org

INHALT

Inhaltsverzeichnis

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 lässt sich KI in der Softwareentwicklung einsetzen?

Die ersten Schritte waren ernüchternd, vor allem was die Qualität anbelangt. 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 lang-

samer statt schneller. RAND kam 2024 auf mehr als achtzig Prozent gescheiterter 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 ist das ganze Problem in einem Satz, und es ist nicht das Problem der Modelle. 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.

Das war der Schwenk, und alles, was in diesem Buch steht, hängt daran. Ich habe aufgehört zu fragen, ob KI in der Entwicklung funktioniert, und angefangen zu fragen, unter welchen Bedingungen sie es tut. 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. Was in dieser Zeit half, waren nicht die Grundsatzdebatten. Es waren 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, sondern 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.

Der Punkt ist, wie er vorgegangen ist. 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.

Es ging mir hier nicht so sehr um die Technik. 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, war für mich aber der Auslöser dafür, dass aus einer Vermutung eine Gewissheit wurde. 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. Es beantwortet eine andere Frage. 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

Über die Einführung von KI gab es in der fuseki GmbH verschiedene Meinungen. Mein damaliger Chef Heiko Dietz hielt KI-gestützte Entwicklung für nicht praktikabel, und wir sind darin nicht zusammengekommen.

Während der Arbeit an diesem Buch 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 man daraus zieht.

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. Interessant ist weniger das Ob als 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 ist die Weiterentwicklung des Whitepapers. 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, das ist der Punkt, den ich am häufigsten wiederhole: Man braucht keines von beiden, um den Prozess zu fahren.

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 herstellergebunden.

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 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. 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 sie im Alltag auch gemeinsam im Einsatz sind.

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, und das ist eine bewusste Entscheidung. 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 sie in der Branche so verwendet werden: 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 für Session-Start, Session-Ende und die Retrospektive. Das Kit ist keine Plattform und kein Agent, es ist eine dünne Schicht aus Textdateien und einem Adapter. Installiert wird es mit einem Node-Skript, das ein paar Fragen stellt.

Skills sind kein Alleinstellungsmerkmal von Claude Code. Codex kennt dasselbe Konzept, Cursor kennt es, und die anderen Werkzeuge ziehen nach. Was sich unterscheidet, ist das Format und der Ort: 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 man wirklich braucht

Die folgende Aussage ist mir besonders wichtig, und deswegen stelle ich sie schon hier in die Einleitung.

Um den Prozess zu fahren, braucht man das kanban-kit nicht. 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. Welche das ist, wird in der Konfiguration festgelegt und kann dort auch geändert werden.

Und genau genommen braucht man auch das claude-workflow-kit nicht. 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, wird im richtigen Moment geladen 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 nicht eines der Werkzeuge. Nimm 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.

	claude-workflow-kit	kanban-kit
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 mit dem, was aus jedem einzelnen im KI-Modus 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 nicht aus Prinzipientreue, sondern als Folge dieser Eigenschaften. Kapitel 4 beschreibt die Rollen, die es dafür braucht, und eine, über die zu wenig gesprochen wird.

Dass dieser Teil am Anfang steht, liegt vor allem an Kapitel 2. 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 sind sie unhaltbar.

Der Grund ist nicht ideologisch, sondern praktisch. Schwergewichtige Methoden setzen auf Plan-Stabilität. Man entscheidet einmal, dokumentiert ausführlich, implementiert lange. Diese Annahme funktioniert nur, wenn sich die Welt zwischen Plan und Auslieferung nicht zu schnell ändert. In einer Welt, in der Sprachmodelle ihre Fähigkeiten monatlich erweitern und in der eine sinnvolle Aufgabe oft in Stunden statt Tagen umsetzbar ist, ist Plan-Stabilität eine Illusion.

Drei Verschiebungen erzwingen Agilität

Die Implementierung wird billig. Ich habe im vergangenen Jahr eine Keycloak-Integration gebaut, ohne vorher je etwas mit Keycloak gemacht zu haben. Null Erfahrung. Nach vier Stunden lief sie. Vor der KI hätte ich dafür zwei bis vier Wochen eingeplant, mit Einarbeitung eher mehr. Wenn das passiert, ist Implementierung nicht mehr die knappe Ressource. Knapp sind nun das Verstehen, das Reviewen, das Entscheiden. Diese Tätigkeiten sind ihrer Natur nach 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. Ihre Notwendigkeit ist gewachsen.

Individuen und Interaktionen über Prozesse und Werkzeuge: gerade weil das Werkzeug jetzt sehr stark geworden ist, ist die Frage, wie Menschen miteinander und mit dem Werkzeug umgehen, wichtiger denn je. Funktionierende Software über ausführliche Dokumentation: gerade weil die KI Dokumentation auf Knopfdruck produziert, ist es wichtig, sie nicht zum Selbstzweck werden zu lassen. 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 ist heute noch wichtiger, weil sich der Aufwand für eine Änderung dramatisch reduziert hat. 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 braucht es keine Aktualisierung. Es braucht, wie immer, den Blick darauf, wie sie konkret umgesetzt werden. Ich habe deshalb die zwölf originalen Sätze genommen und an jeden dieselbe Frage gestellt: Was verlangt er weiterhin, was wird bei seiner Umsetzung leichter, und wo ist eine neue Vorsicht nötig?

Das Ergebnis ist eindeutiger, als ich erwartet hatte. Bei keinem Prinzip ist die Umsetzung im KI-Modus aufwendiger geworden. Bei den meisten ist sie leichter. Bei einigen verschiebt sich die Skala oder die Form. Wer die Prinzipien schon 2001 ernst genommen hat, hat im KI-Zeitalter einen guten Startpunkt.

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 konkret verlangt. Wer die Prinzipien 2001 ernst genommen hat, ist heute gut gerüstet. Wer sie als Lippenbekenntnis behandelte, wird seine Versäumnisse jetzt deutlicher merken. Die KI verstärkt, was sie vorfindet, und sie ist kein Korrektiv.

Scrum passt im KI-Zeitalter nicht mehr, Agilität schon

Diese Unterscheidung ist mir wichtig, weil ich sie im Blogbeitrag selbst unscharf gelassen und das Missverständnis damit provoziert habe. 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 im Vergleich zu dem, was ein Sprachmodell innerhalb eines Vormittags an Vorschlägen produzieren kann.

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, ge-

testet, dokumentiert, gemergt. Ein Sprint-Plan wird in einer Welt, in der sich Anforderungen schneller umsetzen als planen lassen, zur Bremse für genau die Anpassungsfähigkeit, die er eigentlich herstellen sollte.

Auch Boris Gloger hat formuliert, dass die alte, oberflächliche Art, Scrum zu betreiben, am Ende ist. Was mich an der Verteidigungslinie stört, ist 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. Also brauche ich ein System, das das abbildet. Deshalb arbeite ich mit einem Board als zentralem Artefakt und mit dem Issue als Quelle der Wahrheit. Prinzipien statt Prozesse. Flow statt Sprints. Das Issue als Takt, nicht der Sprint als Planungseinheit.

Ein Wort zur Benennung, weil ich hier ungenau sein könnte und es nicht sein will. Was ich benutze, ist 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, und dabei bleibt es. Namen setzen sich nicht nach Lehrbuch durch. Wo es im Buch um die Mechanik geht, spreche ich vom Board oder vom Taskboard, und wo eine Kanban-Regel wirklich gilt, sage ich es dazu.

Was dabei nicht verschwindet: 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. Was ich ablehne, ist Plan-Stabilität über Monate, nicht der Plan.

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 anfing, 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, von denen jede den bisherigen Text als Eingabe hat.

Ein zweiter Punkt hat bei mir länger gebraucht, und er wiegt im Alltag schwerer: **Das Modell setzt ein Gespräch nicht fort, es liest es bei jedem Zug komplett neu.** Für mich fühlt sich der Chat wie eine Fortsetzung an. Für das Modell ist jeder Zug das erste Mal: Der gesamte bisherige Verlauf wird neu übergeben und neu verarbeitet. Der Zustand liegt nicht im Modell, er liegt im Text, den man ihm schickt. Und wenn sich ein Werkzeug an frühere Sessions zu erinnern scheint, dann weil eine Werkzeugschicht sie als Text wieder hineinreicht.

Wer das einmal verstanden hat, versteht auch, warum die Lösung dieses Buches immer wieder dieselbe ist: Was das Modell wissen soll, muss aufgeschrieben werden, aktuell bleiben und im richtigen Moment im Kontext landen.

Fünf Eigenschaften, fünf Regeln

Erstens: Es optimiert auf Plausibilität, nicht auf Korrektheit. Trainiert wurde auf die Frage, welches Token als Nächstes wahrscheinlich ist. Nicht darauf, 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 ist die Stelle, an der die Ausgabe an etwas gebunden wird, das nicht aus der Verteilung stammt.

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

Drittens: Das Fenster ist begrenzt, und innerhalb des Fensters wird umgewichtet. 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 folgt, den Prozess in einzelne Skills zu zerlegen, die im richtigen Moment geladen werden, statt in eine lange Datei am Anfang. Und sie erklärt, warum der Nacht-Runner pro Issue eine frische Session startet.

Viertens: Das Modell markiert nicht, wo sein Muster aufhört. Es gibt keine eingebaute Instanz, die zwischen "das habe ich tausendmal gesehen" und "das rate ich gerade" unterscheidet und diesen Unterschied ausgibt. 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 man warten könnte, also muss man selbst hinsehen.

Fünftens: Es ist nicht deterministisch. Weil aus der Verteilung gezogen wird, 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 nicht Bürokratie, sie sind geronnene Erfahrung aus Projekten, die ohne sie gescheitert sind. Wichtig ist mir dabei, warum sie passen. Sie sind nicht richtig, weil sie alt sind. Sie sind richtig, weil sie genau die Fehlerklasse adressieren, die ein Sprachmodell produziert. Das ist ein Zufall der Geschichte, kein Verdienst der Vergangenheit. Wäre die Fehlerklasse eine andere, bräuchten wir andere Leitplanken.

Der Unterschied zwischen einer Regel im Kopf und einer Regel im Build ist dabei der Unterschied zwischen Wunsch und Wirkung. 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 ohne die Fähigkeit lebte, neue Erinnerungen zu bilden. Intelligent, freundlich, hilfsbereit, und doch kein Gestern.

Zwischen zwei Sessions trifft das Bild. Innerhalb einer Session ist es falsch, und der Unterschied ist nicht akademisch. Wer glaubt, das Modell habe innerhalb der Session 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 hilft zuverlässig.

Diese Korrektur zeigt das Muster: Eine ungenaue Vorstellung davon, wie ein Modell arbeitet, führt zu den falschen Vorkehrungen. Nicht zu gar keinen, sondern zu solchen, die plausibel wirken und nicht wirken.

Was hier bewusst fehlt

Diese Beschreibung ist vereinfacht. Sie lässt aus, was in der Praxis dieses Buches keine Rolle spielt: 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 in ihrer Ausprägung. Reasoning-Modelle etwa erzeugen vor der Antwort eine längere interne Ableitung, was die vierte Eigenschaft abschwächt, aber nicht aufhebt.

Die Details unterscheiden sich außerdem zwischen Anbietern und ändern sich schneller, als ein Buch nachziehen kann. Was sich langsamer ändert, ist die Ebene darüber, und deshalb steht dieses Kapitel im Konzeptteil und nicht im technischen Teil.

Ein letzter Punkt, der in keiner Architekturbeschreibung steht und trotzdem aus ihr folgt: Die KI verstärkt die Arbeitsweise, die sie vorfindet. Sie setzt plausibel fort, was man ihr gibt, und meldet nicht, wenn es schlecht ist. Wer sauber arbeitet, wird schneller sauber. Wer schlampt, wird schneller schlampig. Die KI ist kein Korrektiv. Genau deshalb handelt das nächste Kapitel davon, wer hier eigentlich 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. Und sie wirken nicht nur umsetzbar: 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. Es braucht 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 qualitativ etwas anderes, es ist ein Bruch und keine Fortsetzung. Aber es gehört behandelt wie ein Werkzeug, weil nur so klar bleibt, wer die Verantwortung trägt.

Der Begriff, der die Beziehung am besten trifft, ist der der Implementations-Engine: ein leistungsstarker 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 und nicht bloß dekorativ.

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. Das ist der ganze Unterschied.

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 hier liegt das eigentliche Problem: Das Modell 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.

Dort 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 nicht überflüssig, es macht ihn wichtiger, weil es ihn nicht kann.

Die Denkarbeit wandert nach vorn

Daraus folgt der Satz, der meinen ganzen Prozess trägt: Die Denkarbeit wandert nach vorn.

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. Kein Feature, 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. Es geht nicht um mehr Dokument. Es geht um mehr Klärung. Ein Issue hat vier Abschnitte und passt auf eine Seite. 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. Verstehen und bauen lassen sind zwei verschiedene Dinge. Das Bauen nimmt mir das Modell ab, das Verstehen nicht. Es täuscht es 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 GO. 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.

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

Diese drei Phrasen sind Mikro-Verträge. Ohne sie passiert nichts, mit ihnen passiert genau das, was im Issue steht. Trigger-Phrasen sind deshalb keine Bedienungsanleitung, sie sind eine Sicherheitsarchitektur.

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 GO 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. Ich habe nicht den Code zurückgerollt, sondern den Workflow.

Daraus folgt auch die Rolle des Menschen im Alltag. Das Modell muss aktiv eingebremst werden. 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.

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 wird die KI als Werkzeug eingesetzt, das eigene Charakteristika hat, aber keine Rollenposition im klassischen Sinne. Dieses Kapitel beschreibt beide Seiten und macht die Grenze deutlich.

Product Owner: das Was

Die Rolle des Product Owner verantwortet das Was: Anforderungen definieren, priorisieren, Issues formulieren oder nach Diktat formulieren lassen. Sie entscheidet, welches Issue als nächstes umgesetzt wird und welches warten muss. Sie gibt das GO und die Trigger-Phrasen für Push und Merge. Wenn das Team ein Board nutzt, schiebt sie die Issues nach Ready. Niemals die KI.

Diese Rolle verschwindet im KI-Modus nicht, sie wird wichtiger. Der Grund ist einfach: Spezifikationsqualität wird zum Engpass. Wer unscharf beschreibt, bekommt unscharfen Code, nur eben in höherem Tempo. Was diese Rolle in klassischem Scrum an Aufgaben trug, Sprint Planning, Stakeholder-Management, Release-Strategie, verschwindet nicht. Es verteilt sich nur auf engere Takte. In diesem Modus lassen sich mehrere Mikro-Plannings pro Tag durchführen und mehrere Releases freigeben, weil die KI die operative Last reduziert.

Für die Zusammenarbeit mit einer PO-Rolle gibt es im 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. Sie hat ein Veto 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. In dieser Welt Senior zu sein bedeutet, das Plausible vom Korrekten unterscheiden zu können.

Meine Keycloak-Integration ist dafür das Beispiel, das ich am häufigsten erzähle, und es wird regelmäßig falsch verstanden. 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. Man muss die Fehlerkette im Kopf haben: DNS, Reverse Proxy, Zertifikat, Keycloak-Konfiguration, Client-Konfiguration. Ich konnte die Sache nur lösen, weil ich sie beurteilen konnte, und ich konnte die KI bremsen, 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. Die Kompetenz, die dabei zählt, ist mehr als Wissen. Sie ist das Urteil darüber, was mit diesem Wissen anzufangen ist.

Qualitätsrolle: die Verifizierung

Die Qualitätsrolle verantwortet die Verifizierung. Sie führt die Code-Reviews durch ein zweites Modell anderer Bauart als das implementierende, prüft Coverage- und Mutation-Scores, führt Security-Reviews durch 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 nicht Misstrauen, sondern Statistik. Ein Modell ist gegenüber seiner eigenen Lösung systematisch unkritisch. Es hat sie soeben selbst formuliert, es würde dieselben Vereinfachungen jetzt noch einmal vornehmen. Verschiedene Modelle haben verschiedene blinde Flecken, 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 mehr Material.

In der heutigen Praxis ist das die Rolle, die am ehesten wegrationalisiert wird. 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, wird die Verifizierung zum Engpass, und damit wird diese Rolle wichtiger, nicht unwichtiger. 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 einer generalistisch arbeitenden Person landen.

Die KI als Werkzeug

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

Was sie hat, ist eine sehr 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

Die KI darf	Die KI darf NICHT
Pläne erstellen und zur Diskussion stellen	Eigenständig Issues nach Ready ziehen
Refactorings durchführen, sobald freigegeben	Plan-Approval als GO interpretieren
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

Symbolisch wird die Mitarbeit der KI in Commit-Messages durch eine Co-Authored-By-Zeile kenntlich gemacht. Diese Zeile dokumentiert Transparenz, sie verleiht keinen Teammitglieds-Status. Sie ist eher das, was in einem wissenschaftlichen Paper die Acknowledgement-Sektion ist: Anerkennung dafür, dass das Werkzeug substantiell 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.

Ich halte den Einwand für berechtigt, dass eine Werkzeug-Zeile im Commit widersprüchlich wirkt, 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. Was sich verschiebt, ist die Form: weniger Stand-ups, mehr asynchrone Koordination über Issue und Board. Die Senior-Developer-

Rolle wird dabei zur Architektur-Ankerin, weil das Modell allein keine kohärente Architektur erzeugt.

Die Rolle, über die zu wenig gesprochen wird

Eine Rolle fehlt in dieser Aufzählung, und ihr Fehlen ist gerade dabei, ein Problem zu werden: die der Berufseinsteigerinnen und Berufseinsteiger.

Die Zahlen sind deutlich. 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 nicht sozialpolitisch, es ist betriebswirtschaftlich. 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. Lernen passiert dann, wenn man sich wirklich mit einem Problem auseinandersetzen muss. Wenn die KI diese Phase überspringt, entsteht Oberflächenkompetenz ohne Tiefe. Man fragt, bekommt eine Antwort, baut sie ein, fertig. Kein Ringen, kein Mentalkomplex.

Genau die Fähigkeit, die KI zu bremsen, entsteht in den Lernsituationen, 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 nie eingelöst wird. 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 die Fähigkeit, die 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. Ich habe das nicht selbst erprobt und gebe es deshalb als das weiter, was es ist: eine Idee, die ich für plausibel halte, ohne Beleg.

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

TEIL II

Die zwei Werkzeuge verstehen

> Zwei Kapitel, zwei Werkzeuge, und vorweg der Satz, der für beide gilt: Man braucht 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. Was ich in diesem Teil beschreibe, sind 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 beschreibt das claude-workflow-kit, eine Bibliothek aus Skills, die den Neun-Schritt-Prozess ausführbar macht, samt der Begründung, 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

Bevor ich beschreibe, was das kanban-kit kann, sage ich noch einmal, was es nicht ist: eine 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, nicht weil die anderen Optionen schlecht wären. Ganz ohne Statusträger geht es allerdings nicht: Das GO ist der Moment, in dem ein Mensch ein Issue von Backlog nach 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 lassen sich 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.

Dass die Karte Abhängigkeiten als Nummern kennt, ist keine Kosmetik. Der Nacht-Runner liest genau dieses Feld 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, und die will ich hervorheben.

Den Ideen-Speicher. 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 Speicher. Beide Richtungen brauchen nur das Recht zum Verschieben, nicht zum Löschen. Das ist leichtgewichtiges Grooming: Ich muss nichts wegwerfen, um mein Board sauber zu halten. In der PO-Schleife aus Kapitel 11 ist der Ideen-Speicher 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 entstehen durchschnittliche Lead Time und Cycle Time, die Verweildauer je Spalte, der 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 man sich sonst nicht stellt: 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.

Es gibt zwei Ebenen von Rollen. Plattformweit `USER` und `ADMIN`, pro Projekt `OWNER`, `ADMIN`, `MEMBER` und `VIEWER`. Der letzte OWNER eines Projekts kann nicht entfernt oder degradiert werden, was eine kleine, aber wichtige Sicherung ist. Die Idee für weitere Rollen mit individuell zugeschnittenen Rechten gibt es schon. 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 werden erst nach einer Aufbewahrungsfrist endgültig entfernt. Karten in Done bekommen einen Countdown, nach dem sie archiviert werden.

Der Editiermodus trennt den Alltag von strukturellen Änderungen. Standardmäßig ist er aus und wird nach einem Neustart nicht gemerkt. 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 zugehörige 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 einen Grund, die übrigen Namen jetzt umzubenennen, sah ich bisher nicht. Wer die Zeile liest, weiß jetzt, warum dort etwas anderes steht als auf dem Werkzeug.

KAPITEL 6

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

Obwohl mein Workflow in einer Datei stand, hat ihn das Modell 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 es lohnt sich, genau hinzugucken, damit die richtigen Vorkehrungen getroffen werden können. 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 einmal am Anfang gelesen wird, verliert dadurch zwangsläufig 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 ähnlich aus, die Konsequenz für den Umgang damit ist es nicht.

Die Idee: eine Anweisung pro Schritt, im richtigen Moment

Das Kit ist die Antwort auf die Probleme, die diese Befunde zeigen. 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. Der Unterschied ist ganz klar: Auf der einen Seite ist es jemand, der sich an eine Einarbeitung von vor drei Monaten erinnern soll, und auf der anderen Seite jemand, der vor jedem Arbeitsschritt die passende Checkliste aufschlägt. Der zweite Fall produziert weniger Fehler, nicht wegen größerer Klugheit. Das System stützt ihn.

Ein zweiter Punkt ist mir dabei genauso wichtig: Ich rufe die Skills selbst auf. Wenn der Plan steht und die Issues erzeugt werden sollen, 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 wird geladen, weil ich es entschieden habe.

> Skills lösen das Problem, das ich lange falsch beschrieben habe, und zwar nicht als Automatik, sondern als 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
1.5	Fachliches Issue anlegen (optional)	Modell	`/fachplan`
2	Anforderung planen	Modell	`/plan`
3	Issues anlegen	Modell	`/issues`
4	**GO: Issues nach Ready**	**Mensch**	kein Skill
5	Ready-Issues implementieren	Modell	`/implement-ready`
6	Lokale Checks ausführen	Modell	`/local-check`

Schritt	Was	Wer	Skill
7	Review durchführen	Modell	`/review`
8	**Push auf main**	**Mensch**	`/push-main`
9	**Merge nach production**	**Mensch**	`/merge-production`

Zwischen Schritt 8 und 9 wird der Test-Server im Browser geprüft. Dafür gibt es keinen Skill, Pflicht ist es trotzdem.

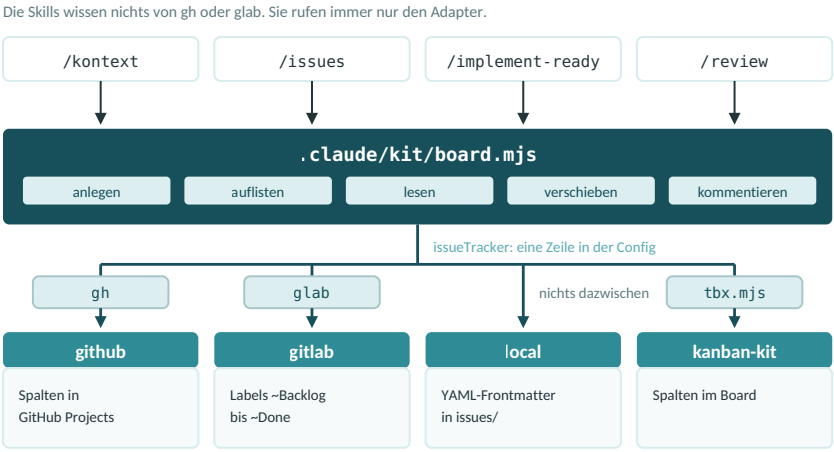
Quer dazu stehen einige weitere Skills. `/kontext` lädt zum Session-Start den Lageüberblick, `/document` schreibt zum Session-Ende den Log, `/retro` ist der Wartungsschritt für den Prozess selbst. `/issue-review` sitzt zwischen Schritt 3 und dem GO und lässt ein fertiges Issue von zwei Modellen gegenlesen; Kapitel 9 beschreibt das ausführlich. `/implement-test` und `/implement-done` sind der granulare 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: Issue anlegen, auflisten, lesen, verschieben, kommentieren.

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



Die Frage in der Softwareentwicklung ist ja immer: Wann baut man 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 Schuld notiert. Erst als ich dann wirklich mit der GitLab-Anbindung begann, brauchte ich die Abstraktionsschicht.

Konfiguriert wird 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
`github`	GitHub CLI (`gh`), authentifiziert, plus ein Projekt-Board mit fünf Spalten
`gitlab`	GitLab CLI (`glab`), authentifiziert, Spalten werden über Labels abgebildet

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

Der lokale Modus ist dabei mehr als eine Notlösung. Es gibt kein Board, kein externes Command Line Interface, 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

Die Frage kommt in jedem Gespräch, deshalb beantworte ich sie hier und nicht im Anhang: Jira wird nicht unterstützt.

Das liegt nicht an Jira. Ein Adapter dafür wäre möglich, die Schnittstelle ist schmal, und Jira hat alles, was der Prozess braucht. Die Antwort ist ganz einfach: 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 er gebraucht wird, und vermutlich an einem Nachmittag. Bis dahin wäre er Code, der existiert, ohne je aufgerufen zu werden.

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 einem Kompromiss und ist in der Praxis oft der ehrlichere, weil ein Ma-

nagement-Jira und ein Board für die tägliche Entwicklung selten dieselbe Granularität haben.

Wie beide Werkzeuge zusammenspielen

Das Zusammenspiel ist unspektakulär. Die Skills schreiben und bewegen Issues mit Hilfe des Adapters 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 stehen in diesem Teil nebeneinander, so wie sie im Alltag auch nebeneinander stehen.

Kapitel 7 beschreibt das Issue, und wenn aus diesem Buch eine einzige Regel bleiben soll, dann die: Das Issue ist die Quelle der Wahrheit, nicht der Chat. Kapitel 8 ist die Landkarte mit den neun Schritten, jeder mit klarem Akteur und klarem Ergebnis. Kapitel 9 lässt das fertige Issue von zwei Modellen gegenlesen, bevor es freigegeben wird. Kapitel 10 zeigt einen kompletten Tag von halb neun bis Feierabend, und dieser Tag sieht anders aus, als die meisten erwarten: Er beginnt mit dem Ernten und endet mit dem Säen.

Danach drei Kapitel, welche 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 lesen.

KAPITEL 7

Das Issue als Single Source of Truth

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

Der Grund ist rein technisch. Ein Sprachmodell beginnt jede Session bei null, und innerhalb einer Session verliert früh Gesagtes an Gewicht, und zwar auf eine Weise, die man nicht bemerkt. Das Modell meldet nicht, wenn ihm etwas entgleitet. Was im Issue steht, wird zu Beginn jeder Bearbeitung frisch gelesen.

Ich habe die Erinnerung aus dem Modell herausgezogen und in Dateien gelegt. Das ist der ganze Trick, und 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 gemacht wird, was vorher fehlte, welche Vorgeschichte dazugehört. Hier steht auch der Rückverweis auf ein fachliches Issue, falls die PO-Schleife aus Kapitel 11 im Spiel ist.

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. Das ist der Abschnitt, an dem sich entscheidet, 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 und nicht mehr missen möchte: eine grobe Aufwandsschätzung und ein ausdrück-

liches Out-of-Scope. Das Out-of-Scope ist die Stelle, an der ich sage, 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. Das habe ich weiter vorne schon beschrieben. Je enger ich die Grenzen formuliere, in denen sich das Modell bewegen darf, desto erfolgreicher und hochwertiger wird der Output.

Warum die Abhängigkeiten Nummern sind

Man könnte jetzt denken, der Prozess ist wichtiger als der Inhalt, aber das Ganze trägt spätestens, wenn der erste Nachtlauf läuft. 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.

Daraus folgt eine Regel, die ich einmal auf die harte Tour gelernt habe: 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 Nachtlauf 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 nicht nur leichter umzusetzen. Sie sind leichter zu reviewen und im Fehlerfall leichter zurückzurollen.

Vor einem Nachtlauf 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. Ungenutzter Code muss weiterhin ausspezifiziert, getestet und gewartet werden.

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. Wird ein Akzeptanzkriterium ergänzt, 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 für die Verhandlung mit einem Product Owner. Sie läuft im Body des Issues, nicht in Kommentaren darunter. Das hat einen banalen technischen Grund: Der Board-Adapter liest keine Kommentare. Was in einem Kommentar steht, sieht eine spätere Session nicht. 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. Es ist nicht nur Implementierung, es ist auch Übersetzung.

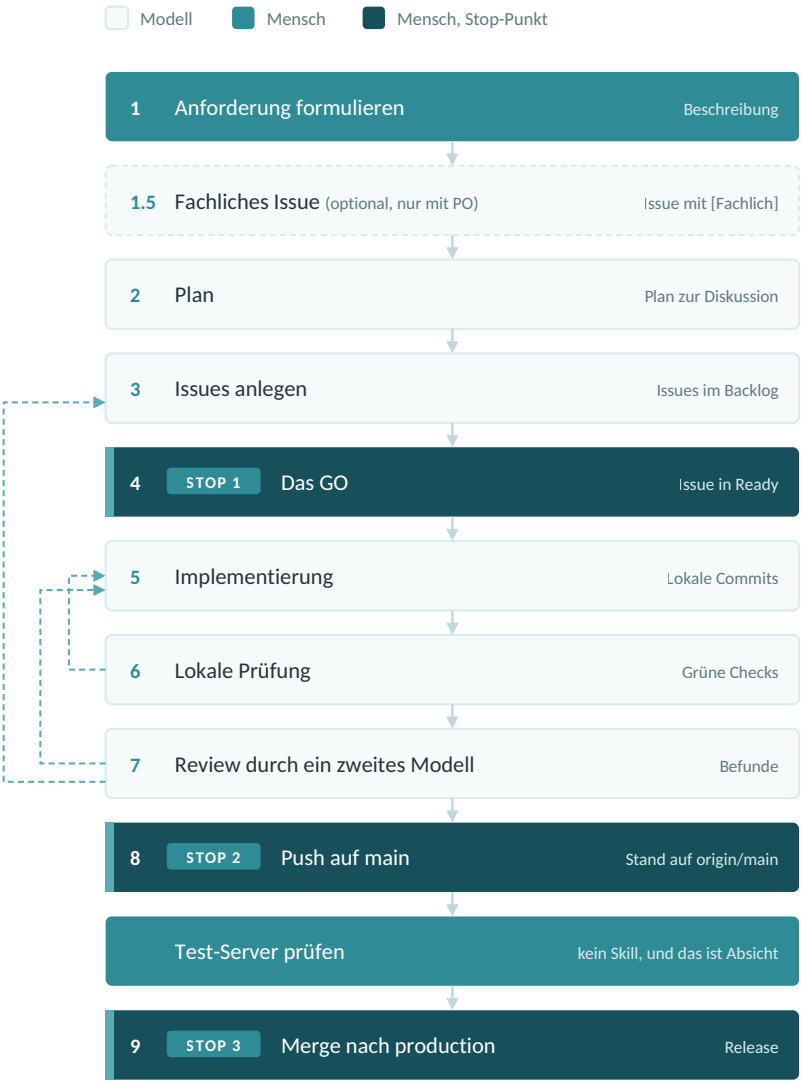
Was das Modell nicht tut, ist die Bewegung nach Ready. Ein Issue zu formulieren und ein Issue freizugeben sind zwei verschiedene Handlungen, und nur die zweite ist eine Entscheidung.

KAPITEL 8

Der Neun-Schritt-Prozess im Überblick

Was in Teil I als Werte, Prinzipien und Rollen beschrieben wurde, lässt sich als konkrete Anleitung formulieren. 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.

01.5 Fachliches Issue (optional)

Modell. Nur in Projekten mit Product Owner. Die rohe Anforderung wird in ein fachliches Issue überführt, 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 werden ausdrücklich gemacht. Der Plan wird zur Diskussion gestellt, nicht zur Implementierung.

Dieser Schritt ist der meistunterschätzte im ganzen Prozess. Hier wandert die Denkarbeit nach vorn, und was hier nicht geklärt wird, wird später im Code geklärt, wo es das Zehnfache kostet.

Output: ein lesbarer Plan, an dem sich entscheiden lässt, 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 GO

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

Eine Plan-Akzeptanz aus Schritt 2 ist nicht das GO. Das ist eine der häufigsten Verletzungen des Prozesses, 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.

Testgetrieben 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, das Urteil darüber bleibt menschlich.

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.

Nicht alles gehört ins Review. 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
1.5 Fachliches Issue	Modell	`[Fachlich]`-Issue
2 Plan	Modell	Plan zur Diskussion
3 Issues	Modell	Issues im Backlog
4 GO	**Mensch**	**Issue in Ready**
5 Implementierung	Modell	Lokale Commits
6 Lokale Prüfung	Modell	Grüne Checks
7 Review	Modell	Befunde
8 Push	**Mensch**	**Stand auf main**
9 Merge	**Mensch**	**Release**

Die Nummerierung wirkt pedantisch, und sie ist es auch. 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.

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. Was nicht passiert: einen Stop-Punkt überspringen, weil das Ergebnis ohnehin gut aussieht.

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.

Nicht gelegentlich. Verlässlich.

Das war zuerst ärgerlich, weil ich glaubte, fertig zu sein. Dann wurde es interessant, weil es sich wiederholte. Was sich wiederholt, ist eine Eigenschaft des Verfahrens.

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 prinzipiell 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.

Zwei Prüfer mit verschiedenen Aufträgen

`/issue-review` sitzt zwischen `/issues` und dem GO. Zwei Modelle, die das Issue nicht geschrieben haben, lesen es und schlagen Schärfungen vor.

Ein Autor und ein Prüfer sind besser als ein Autor allein. Aber ein Prüfer sieht eine Sorte Fehler, die, nach der er sucht. Deshalb bekommen die beiden denselben Text und verschiedene Aufträge.

Rolle	Fragt
Vollständigkeit	Ist jedes Akzeptanzkriterium maschinell prüfbar? Steht Manuelles im dafür vorgesehenen Block? Fehlen Randfälle?
Scope und Bestand	Ist der Schnitt zu groß? Fehlen Abhängigkeiten? Was bricht, das nicht im Issue steht?

Zwei Modelle mit identischem Auftrag sind kein zweiter Blick. Sie sind derselbe Blick zweimal, und der Zugewinn ist entsprechend klein. Der Gewinn liegt im Blickwinkel.

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 etwas dazu. Modelle einer Familie teilen auch die Anschauung darüber, 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.

Praktisch heißt das, dass ein Prüfer ein Adapter sein muss. 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.

Dass der Prompt über die Standardeingabe geht, hat einen praktischen Grund. Ein Issue-Body enthält Backticks, Anführungszeichen und Zeilenumbrüche. Ihn durch eine Kommandozeile zu quoten ist genau die Fehlerklasse, die aus dem Board-Adapter entfernt wurde.

Die Vorlage bringt nur die eigene Modellfamilie mit. Ein fremdes Modell trägt man 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 ersten zwei Prüfer, die nicht der Autor sind.

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

Der vierte war das fremde Modell. Der Prüfer, dessen ganzer Wert darin besteht, die blinden Flecken der anderen nicht zu teilen, war durch die Sortierung stillgelegt. Nichts daran war kaputt. Es lief, es lieferte Befunde, es sah vollständig aus.

Deshalb gibt es `pairs`, und deshalb gibt es einen Befehl, der die fertige Zuordnung ausgibt:

```
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 man ausrechnen muss, verbirgt genau die Fälle, die man nicht bedacht hat.

Was kann raus

Prüfer schlagen Ergänzungen vor. Immer. Ein fehlender Randfall, ein zusätzliches Kriterium, ein Hinweis auf etwas Bedenkenswertes. 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 das Wesentliche im Zusätzlichen untergeht.

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

Wo der Mensch bleibt

Die Befunde gehen als Kommentar ans Issue. Der Body wird nie automatisch überschrieben, der Skill zeigt einen Vorschlag und fragt einmal nach.

Zwei Modelle können sich einig und trotzdem falsch sein. Übereinstimmung sagt nur, dass beide dasselbe gelesen haben.

Dazu kommt der Punkt, der mir wichtiger ist. 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 Issue eine Marker-Zeile mit den beteiligten Prüfern und dem Datum. Lehne ich ab, entsteht kein Marker. Ein Review, dessen Ergebnis verworfen wurde, hat das Issue nicht geschärft.

Das Verfahren fügt dem Prozess keinen vierten Stop-Punkt hinzu. Es schärft den ersten, indem es dafür sorgt, dass ich weiß, 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.

Was es kostet, und was es nicht leistet

Zwei Prüfer je Issue sind zwei zusätzliche Läufe. Bei siebzehn Issues sind das vierunddreißig. Nach meiner Erfahrung lohnt sich das 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.

`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.

Es 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 man sich wünscht, und mehr, als man vorher hatte.

KAPITEL 10

Ein Tag im KI-augmentierten Team

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

Der Tag hat eine Form, die auf den ersten Blick verkehrt herum aussieht: Er beginnt mit dem Ernten und endet mit dem Säen. Morgens prüfe ich, was in der Nacht gebaut wurde, und bringe es nach production. Nachmittags denke ich nach: neue Anforderungen, Grooming, Plan, Issues. Abends gebe ich frei, was die kommende Nacht gebaut wird.

Der Nachtlauf ist bei mir der Normalfall. Das ist die zweite Sache, die vielleicht überrascht, aber sie hat einen klaren Grund. 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	`/kontext`, Lageüberblick und Nachtprotokoll	Ich lese
8:45	Ergebnisse der Nacht sichten, In review durchgehen	Ich
9:00	`/local-check`	Modell, ich lese den Bericht
9:30	`/review`	Zweites Modell, ich entscheide
10:30	**`/push-main`**	**Nur ich**
10:45	Test-Server im Browser prüfen	Nur ich
11:15	**`/merge-production`**	**Nur ich**
11:30	`/document`	Modell
13:00	`/fachplan` für die neue Anforderung	Modell, PO groomt

Uhrzeit	Was	Wer
14:30	`/plan #N` aus dem fachlichen Issue	Modell, ich prüfe
15:30	`/issues`	Modell
17:00	**GO: nach Ready ziehen, Reihenfolge setzen**	**Nur ich**
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 man nachvollziehen kann, welche Entscheidungen getroffen wurden, warum, und welche Alternativen es gab. Dieses Tagebuch hat nie die zweite Woche überlebt, weil es mühevoll ist, alles aufzuschreiben. Diese Aufgabe übernimmt für mich jetzt die KI, und darum ist es so wichtig, dass es eine geordnete Dokumentation gibt, 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. Es ist nicht nur die Zeit, die ich jetzt spare. Es ist einfach die Möglichkeit, dass ich sofort anfangen kann, statt erst groß recherchieren zu müssen.

Wichtig ist mir dabei eine Trennung, die ich in Kapitel 17 ausführe: Was `/kontext` lädt, ist mein Gedächtnis, nicht das des Modells. Das Gedächtnis liegt nicht im Modell, 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 anfing.

8:45 Uhr: was die Nacht gebracht hat

Was mich morgens erwartet, ist 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. Issue 17 wird mit dem komprimierten Sediment von 16 Vorgängern gebaut. 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 nicht mehr Gleichzeitigkeit, sondern weniger Kontext: Jedes Issue ist das erste und einzige seiner Session. Kein Kompressions-Verfall, 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 und nicht nach In review. 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, ist Weitermachen schlicht nicht möglich. Vielleicht finde ich dafür später mal eine Lösung.

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.

Gepusht wurde nichts. 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". Diese Härte ist der Grund, warum die Sache funktioniert. Eine Leitplanke, mit der sich verhandeln lässt, ist keine.

Deshalb verweigert der Nacht-Runner auch den Start, wenn in der Config keine `buildChecks` stehen. 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 es richtig, dass man das ausdrücklich sagen muss.

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.

Der Grund ist einfach: 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. Das ist derselbe Effekt, den alle kennen, die je den eigenen Text korrekturlesen wollten.

Nach einem Nachtlauf 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 ist kein nächtlicher Automatismus. Sie bleibt ein bewusster Akt am Morgen.

> 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 skaliert.

10:30 Uhr: der Push

Der zweite Stop-Punkt. Ich tippe die Trigger-Phrase, sonst passiert nichts. `/push-main` ist gegen selbsttätigen Aufruf gesperrt, 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, und hier habe ich Beispiele, warum das wichtig ist. 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, keiner davon vorher, 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.

`/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. Kein Plan, kein Code, keine technischen Issues.

Das Issue ist das Übergabe-Artefakt. Es geht an die PO-Rolle, dort wird daran gegroomt, und zwar im Body, nicht in Kommentaren. Bis zum

Nachmittag kommt es zurück. Die Regeln dahinter und die Leitplanken, die verhindern, dass so ein Issue versehentlich implementiert wird, stehen in Kapitel 11.

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. Der Plan ist der Schritt, den ich am ernstesten nehme, und er ist die Stelle, an der die Denkarbeit nach vorn wandert. Was ich hier nicht kläre, kläre ich nachts nicht, denn nachts ist niemand da, der etwas klären könnte. 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.

17:00 Uhr: das GO

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. Nicht, weil die KI müde wird. Am Morgen muss ich jedes Ergebnis reviewen. Der Takt wird vom langsamsten menschlichen Schritt bestimmt, nicht vom schnellsten 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 nutze 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

Der beschriebene Tag lief durch. Die meisten Tage tun das nicht.

Am Anfang habe ich den Nachtlauf tagsüber laufen lassen, unter Aufsicht, weil immer irgendwelche Checks fehlten. Man muss sich da ein bisschen eingrooven. In der settings.json werden die Permissions gesetzt, und ich setze Permissions grundsätzlich explizit, statt dem Modell zu erlauben, alles zu machen, was es will. Es hat bestimmt eine Woche gebraucht, bis der erste Nachtlauf 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 kein Versagen, 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.

Der Tag, vor dem ich mich am meisten hüte, ist aber ein anderer: der, 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. Man merkt es nur nicht am selben Morgen.

KAPITEL 11

Die PO-Schleife: fachliche und technische Issues

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

Die PO-Schleife löst das, indem sie zwei Sorten von Issues trennt. Wer keinen PO hat, überspringt dieses Kapitel. `/plan` direkt aufzurufen bleibt der Normalweg.

Zwei Sorten Issues

Fachliche Issues tragen das Präfix `[Fachlich]` im Titel. Sie beschreiben in PO-Sprache das Was und das Warum: Ziel, fachliche Akzeptanzkriterien, Nicht-Ziele, offene Fragen an die PO-Rolle. Strikt technikfrei. Sie werden am Board gegroomt und **nie implementiert**.

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, und sie hat einen praktischen Kern: Die PO-Rolle soll über den fachlichen Inhalt streiten können, ohne über Dateinamen zu streiten. Solange beides in einem Dokument steht, wird über beides gestritten oder über keines.

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` schneidet daraus die technischen Issues. Jedes trägt im Kontext-Abschnitt den Rückverweis “Fachliche Quelle: Issue #N”.
5. Ab hier der normale Weg: GO, Implementierung, Review, Push.

Im kanban-kit landen neue fachliche Issues zunächst im Ideen-Speicher. Die drei Stufen haben dort eine klare Bedeutung: Ideen-Speicher heißt ungesichtete Rohanforderung. Das Einplanen ins Backlog heißt fachlich in Arbeit, ab da ist das Issue adressierbar und groombar. Der Aufruf von `/plan #N` heißt fachlich freigegeben.

Die Verhandlung läuft im Body

Diese Regel wird oft gebrochen, und sie hat einen einfachen Grund: Stünden die Antworten als Kommentare unter dem Issue, wäre die fachliche Dokumentation zerstückelt, ein Teil im Body, ein Teil in den Kommentaren. Das ist nicht sinnvoll.

Deshalb wandern PO-Antworten und Ergänzungen direkt in den Text des Issues. Das fühlt sich zuerst falsch an, weil man den Diskussionsverlauf verliert. Genau das ist beabsichtigt. Am Ende soll kein Protokoll einer Diskussion dastehen, sondern eine Anforderung.

Am meisten Wert haben dabei die **Nicht-Ziele**. Sie sind die Stelle, an der die PO-Rolle sagt, 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

Fachliche Issues gehen nie nach Ready. Ready heißt implementierbar, und ein fachliches Issue ist es nicht. Diese Regel steht nicht nur in der Dokumentation, sie ist mechanisch abgesichert: Landet doch eines in Ready, stellen die Implementierungs-Skills und der Nacht-Runner es

kommentiert zurück ins Backlog, ohne überhaupt eine Session zu starten. Ich verlasse mich nicht darauf, dass ich abends aufpasse.

Der Rückverweis steht 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 Elternteil.

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 eine ehrliche Trennung von Verantwortung: Die PO-Rolle verantwortet das Was und kann es abnehmen, ohne über Technik zu urteilen, für die sie nicht zuständig ist. Zweitens bessere technische Issues, 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, ist die PO-Rolle oft auf zwei Personen verteilt, und die PO-Schleife bildet genau diese Teilung 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 daraus gebaut werden kann. Sie ist die Übersetzung zwischen zwei

Sprachen, und sie ist der Grund, warum es diese Rolle in vielen Organisationen überhaupt gibt.

Genau hier setzt das fachliche Issue an. Es ist das Artefakt, an dem beide arbeiten können, 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 auffallen, wenn man an die Umsetzung denkt. Erst wenn beide zufrieden sind, entsteht mit `/plan #N` der technische Plan.

Ohne diese Trennung passiert regelmäßig eines von zwei Dingen. Entweder der fachliche PO bekommt ein Dokument mit Dateinamen und Schichten vorgelegt 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 durch Sprint Planning, Daily, Review und Retrospektive geleistet werden.

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 sind heute zu treffen?

Das Modell nimmt am Daily nicht teil. Das ist keine Bevormundung, es ist eine pragmatische Entscheidung. Das Daily koordiniert zwischen Menschen, die Verantwortung tragen. Das Modell hat keine eigene Ta-

geslage, 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 die Synchronisation ohnehin über Board, Pull Requests und Pipelines läuft, 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

Kein eigenes Meeting, eine kontinuierliche Tätigkeit. Mindestens einmal täglich, typischerweise als Teil der Morgenroutine, geht die PO-Rolle durch den Backlog. Welche Issues haben offene Designfragen, die heute geklärt werden können? Welche sind reif für Ready? Welche sind veraltet und sollten geschlossen werden?

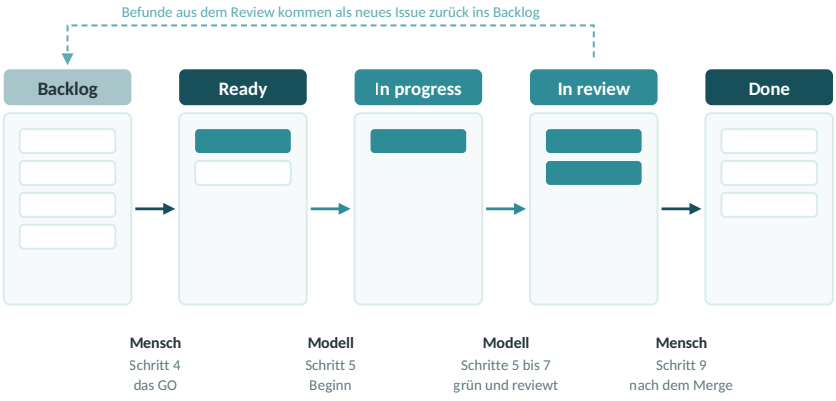
Die dritte Frage stelle ich mir erst, seit ein Backlog von mir zur Ablage geworden war. Ein Backlog, den niemand aufräumt, ist kein Vorrat, er ist Schuldenlast. Der Ideen-Speicher aus Kapitel 5 hilft dabei, weil ich Dinge parken kann, ohne sie wegzuwerfen und ohne dass sie das Board vermüllen.

Das Modell ist beim Grooming nützlich: Issues nach Diktat formulieren, Akzeptanzkriterien vorschlagen, Designfragen aufdecken, die nicht offensichtlich sind. Was es nicht tut, ist die Bewegung nach Ready. Das ist der Moment, in dem ein Issue zum verbindlichen Auftrag wird.

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.

Ihr Output sind 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: Gezeigt wird kein Foliensatz. Gezeigt wird die laufende Anwendung.

Ich halte an einem festen wöchentlichen Termin fest, obwohl der Takt der Arbeit viel kürzer ist. Der Grund ist nicht technisch. Stakeholder brauchen Verlässlichkeit, und ein Termin, den es immer gibt, senkt die Schwelle für Kritik, die sonst vielleicht nicht geäußert wird.

Aus dem Review entstehen neue Backlog-Einträge, manchmal Korrekturen an bestehenden Issues. Das Modell kann diese Anpassungen unmit-

telbar nach dem Review mitformulieren, solange das Gespräch noch frisch ist. Möglich ist auch, schon im Review Notizen zu diktieren, die später geordnet und den einzelnen Issues zugeordnet werden.

KI-Retrospektive

Das einzige Event, das es in Scrum so nicht gibt, und deshalb hat es mit Kapitel 23 ein eigenes Kapitel. 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?

Ihr Output sind keine Erkenntnisse, sondern Änderungen an Dateien. Eine Retrospektive, die keine Datei verändert, war zu abstrakt.

Was sich gegenüber Scrum verschoben hat

Drei Beobachtungen, die ich für belastbar halte.

Weniger Synchronisation, mehr Asynchronität. Der Status steht am Board und ist jederzeit lesbar. Ein Treffen, dessen Zweck das Verlesen dieses Status ist, hat seinen Zweck verloren.

Kürzere Takte bei den Arbeits-Events, gleiche Takte bei den menschlichen. 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 braucht eine Stelle, an der er sich selbst überarbeitet, 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

Die Wahl der Iterationslänge ist keine triviale Designentscheidung. Sie bestimmt, wie schnell ein Team auf Feedback reagieren kann, wie viel Arbeit zwischen zwei Reviews liegt und wie häufig Fehler in Produktion landen können.

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

Ein Tag ist für die meisten Teams der realistische Default. Morgens das Grooming, tagsüber die Implementierung, abends der Push und gegebenenfalls der Merge.

Dieser Takt hat zwei Vorteile. Er ist überschaubar, ein Mensch kann am Abend ehrlich beurteilen, was am Morgen versprochen wurde. Und er produziert täglich Output auf dem Test-Server, was Vertrauen bei Stakeholdern aufbaut. Wer mit weniger Takt arbeitet, läuft Gefahr, dass 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 nicht alle vier Stunden ein Zwischenstand erzwungen wird. Wer hier in zu enge Takte zwingt, 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, und ich behandle ihn deshalb nicht als Exotik. Abends wird freigegeben, nachts wird gebaut, morgens wird geprüft und ausgeliefert. Kapitel 10 beschreibt den Tag, der daraus entsteht, Kapitel 19 die Technik dahinter.

Der Reiz liegt nicht darin, dass nachts gearbeitet wird. Er liegt darin, dass die Nacht eine harte Grenze setzt. 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.

Die Begrenzung ist dabei nicht die Nacht, sondern der Morgen. 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 und für keinen Ersatz. 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 im Moment der Entstehung braucht. 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 Zeitperioden mit definiertem Anfang und Ende. Kanban arbeitet im Flow.

Für KI-augmentierte Entwicklung ist der Flow aus meiner Sicht passender. Issues sind kleinteilig, ihre Bearbeitung dauert Stunden statt Tage. Ein Zwei-Wochen-Sprint, der mit dreißig Issues geplant wird, 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 fixierter Stakeholder-Rhythmus, ein Vertrag, der Zwischenstände vorschreibt. Das ist kein Rückfall, das ist Anpassung an eine Umgebung, die man nicht allein kontrolliert.

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 nicht in der Taktlänge. Er 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, wenn etwas eingerichtet werden soll.

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, und es ist auch dann zu lesen, wenn sonst nichts aus Teil IV gebraucht wird. 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

Dieses Kapitel beschreibt den Betrieb des kanban-kit. Wer GitHub Projects, GitLab oder den lokalen Modus benutzt, kann es 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` und `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

Vorausgesetzt wird eine Docker-Laufzeit. 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

ist es nötig, sonst startet ein reines `docker compose up -d` das alte Image, und man sucht 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 wird `MANBAN_DOMAIN` gesetzt, dann läuft Let's Encrypt automatisch.

Die Umgebungsvariablen, die zählen

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

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

Zwei davon gehören in Produktion zwingend gesetzt: `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 werden stattdessen ins Log geschrieben:

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

Den geloggten 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 einem echten Account verliehen wird. 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 man auf der Login-Seite. Der Token wirkt nur, solange kein Admin existiert, man kann sich damit also nicht aussperren. Danach gehört er aus der `.env` entfernt.

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, man muss nicht Traefik nehmen. Die dafür nötigen Schritte stehen in der Online-Doku unter “Produktions-Deployment”, weil sie stark von der Hosting-Umgebung abhängen.

Was unabhängig davon gilt: 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. Kein Hintergrundprozess, kein Service, keine 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 braucht es gar nichts davon.

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 werden sie über Labels abgebildet. Die Namen sind case-sensitiv, was 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 man leer lassen kann.

Die erste Frage ist die einzige, bei der ich eine klare Empfehlung habe. **Projektlokal** legt die Skills in `.claude/skills/` ab, sie gehören damit zum Repo und werden über git geteilt. Für einen teamverbindlichen Prozess ist das die richtige Wahl, weil jeder, der klont, dieselbe Prozessgrundlage hat. Global ist für die persönliche Nutzung über viele Projekte hinweg gedacht.

Danach Claude Code neu starten. Die Skills werden beim Start geladen; sichtbar werden sie in der Befehlsliste, die aufklappt, sobald man `/` eintippt. Das 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 ausschließlich aus dieser Datei. Ein Update an einem Skill gilt damit in allen Projekten, ohne dass in jedem Repo etwas angepasst werden muss.

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

Zwei Felder verdienen Aufmerksamkeit.

`buildChecks` ist die Liste der Kommandos, die `/local-check` nacheinander ausführt. Alle müssen grün sein, bevor der Skill Vollzug meldet. Dieses Feld ist die Stelle, an der die Qualitätssicherung im Prozess verankert wird, 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 man es manchmal separat ziehen will. Ein leerer String deaktiviert es.

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

`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 GO, 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.

Aktualisieren und in weitere Projekte übernehmen

Weil die Skills projekt-unabhängig sind und nur die Config projektlokal ist, aktualisiert man das Kit, indem man den Installer erneut laufen lässt. Die Config 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.

Was der Installer nicht tut

Er legt keine `kontext.config.json` an. Die ist personenbezogen und wird von Hand erstellt, Kapitel 17 beschreibt sie. 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. Das ist die Konstruktionsregel, und sie ist der Grund, warum der Prozess hält: Ein scharf geschnittener Skill sagt, was in diesem Schritt passieren soll, und genauso, was in diesem Schritt nicht passieren darf.

Dieses Kapitel geht sie der Reihe nach durch. Claude ruft Skills zwar auch von sich aus auf, wenn der Kontext passt, aber auf der sicheren Seite ist, wer den Skill ausdrücklich selbst aufruft.

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 nach `{vault}/Log/YYYY-MM-DD.md`, ohne Vault nach `docs/session-log/YYYY-MM-DD.md` im Projekt. Dokumentation entsteht damit nicht als nachträgliche Pflicht, sondern als Abschluss der Arbeitseinheit.

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

Planung

`/fachplan` (Schritt 1.5, optional) überführt eine rohe Anforderung in 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 ein fachliches Issue als Anforderungsquelle. *Grenze:* Er implementiert nichts und legt keine Issues an. Der Plan ist Diskussionsgrundlage, nicht Freigabe.

`/issues` (Schritt 3) schneidet den freigegebenen Plan in kleinteilige Issues mit den vier Abschnitten aus Kapitel 7. 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 lässt sich vor dem GO entscheiden, womit jedes Issue umgesetzt wird, ohne den Plan noch einmal zu lesen. Im Nachtbetrieb gilt das nicht pro Issue: Dem Nacht-Runner sagt man beim Start, welches Modell er nehmen soll, und das gilt dann für den ganzen Lauf.

`/issue-review` steht zwischen Schritt 3 und dem GO und ist freiwillig. Er lässt ein Issue von zwei Modellen prüfen, die es nicht geschrieben haben, und schlägt einen geschärften Body vor. Ohne Argumente nimmt er das ganze Backlog, mit Nummern genau die genannten. *Grenze:* Er überschreibt den Body nie ohne Zustimmung. Kapitel 9 beschreibt das Verfahren.

Implementierung

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

Skill	Umfang	Wofür
`/implement-ready`	Ganze Ready-Spalte in einer Session	Der Normalfall bei der Tagesarbeit

Skill	Umfang	Wofür
`/implement-test`	Nur die Tests zu einem Issue, rot	Wenn der Testvertrag sichtbar sein soll
`/implement-done`	Gegen die roten Tests bis grün	Fortsetzung von `/implement-test`
`/implement-next`	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.

Zwei Grenzen gelten für alle vier: Sie pushen nie, und sie ziehen keine Backlog-Issues eigenmächtig nach Ready.

Das Paar `/implement-test` und `/implement-done` ist der granulare Einstieg. 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

`/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, was die Prüfung erleichtert und die menschliche Abnahme nicht ersetzt. 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 `reviewScope` den Diff oder alle Dateien lesen. Die Befunde landen im Issue oder im Pull Request.

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

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.

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 ist keine Schwäche, die sich wegprompten lässt, das ist der Stand der Modelle. Ärgern hilft nicht, Vorkehrungen schon.

Die Vorkehrung besteht darin, die Erinnerung aus dem Modell herauszuziehen und in Dateien zu legen. 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 werkzeugagnostisch, der Dateiname ist Werkzeugsache. Ich beschreibe darin, was in diesem Projekt anders ist als in anderen.

Die Workflow-Konventionsdatei, im Beispiel `CLAUDE-workflow.md`, beschreibt den Prozess selbst. Sie ist bewusst projekt-unabhängig gehalten, damit sie in andere Projekte kopiert und dort minimal angepasst werden kann. Sie wird vom Installer angelegt.

Tech-spezifische Konventionsdateien, etwa `CLAUDE-java.md` oder `CLAUDE-react.md`, sammeln Regeln pro Sprache und Framework: Hier sind Dinge festgelegt, die sehr stark mit Softwareentwicklung zu tun haben, wie Testpyramide, Code-Styling oder Architektur-Muster, die zu befolgen sind.

Eine Security-Konventionsdatei, etwa `CLAUDE-security.md`, bündelt die Checks vor Commit und Push. Sie beschreibt, worauf zu achten ist. In Kapitel 20 gehen wir näher darauf ein, dass es wichtig ist, gerade in diesem Bereich Werkzeuge außerhalb des Modells zu nutzen.

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

Alle diese Dateien werden gepflegt 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:

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

Konfiguriert wird das in `kontext.config.json`. Diese Datei legt der Installer bewusst nicht an, denn sie ist personenbezogen.

```
{
  "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 au-

tomatisch am Repo-Namen erkannt. Weicht der Vault-Ordnername davon ab, setzt man `project` in der lokalen Config.

Warum zwei Config-Dateien

Das verwirrt am Anfang, ist aber richtig. `workflow.config.json` ist repospezifisch: Build-Kommandos, Branch-Namen, Review-Modell. Sie gehört ins Repo und wird geteilt, 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 — genau den hält diese Datei, und deshalb gehört sie nicht ins Repo.

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 ohne persistentes Memory gearbeitet wird.

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. Nicht jeder soll seine persönlichen Erfahrungen für sich machen — Wissen und Projektfortschritt sollen über alle geteilt werden.

Memory-Drift

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

Wer Memory-Dateien nur ergänzt, ohne sie zu pflegen, baut eine wachsende Schuldenlast auf. Das Ergebnis sind alte Regeln, neue Realität und wachsende Reibung. Die KI-Retrospektive aus Kapitel 23 ist die Stelle, an

der das geprüft wird, und sie ist der Grund, warum es sie als eigenes Format gibt.

Ein Punkt, den ich nicht beschönigen möchte: Dieser Aufwand liegt beim Menschen. Die Zeit, die das Modell beim Implementieren spart, fließt zu einem Teil in die Pflege des Kontexts zurück. Das ist ein Grund, warum viele, die mit KI-gestützter Softwareentwicklung arbeiten, meinen, dass es nicht schneller geht. 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 kleine Schnittstelle, und die ist bewusst klein: Issue anlegen, auflisten, lesen, verschieben, kommentieren.

Diese Schmalheit ist der Grund, warum ein Plattformwechsel nur eine Zeile in der Konfiguration ist 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.

Diese Trennung war nicht von Anfang an da. Als ich mit der Implementierung begonnen habe, habe ich beide Aspekte immer zusammen behandelt. Später habe ich festgestellt, dass sich Repository und Issue-Tracker sehr gut trennen lassen.

Die praktisch nützlichste Kombination ist `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
`github`	`gh`, authentifiziert	Spalten in GitHub Projects
`gitlab`	`glab`, authentifiziert	Labels `~Backlog` bis `~Done`
`local`	nichts	YAML-Frontmatter in `issues/`
kanban-kit	Instanz, Token, `tbx.mjs`	Spalten im Board

Was bei GitLab anders ist

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

Wichtiger: 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 werden deshalb als Labels abgebildet. Die Bewegung zwischen den Status bleibt dieselbe. Sie ist nur nicht in der Spalte sichtbar, sondern als Label an der Karte.

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

Ein Detail mit Folgen: 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` mit Namespace-Präfix erfüllt das.

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 tatsächlich für unterschätzt. Er hat drei Eigenschaften, die kein gehostetes Board hat. Die Issues sind versioniert, man sieht also, wann eine Anforderung geändert wurde und von wem. Sie sind Teil des Diffs, laufen also durch dasselbe Review wie der Code. Und sie funktionieren ohne alles, auch im Zug ohne Netz.

Was fehlt, ist die Übersicht. 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, man zieht ein Issue von Backlog nach Ready, und danach sieht man zu, wie das Kit die Datei selbst nach In progress und nach In review schiebt. `--help` zeigt den Rest. Mehr kann es nicht.

Sobald mehrere Menschen den Stand sehen wollen, reicht das nicht mehr. Das ist die Stelle, an der ein richtiges Board dazukommt, und dann ist es eine Frage des Geschmacks, ob GitHub Projects, GitLab oder das kanban-kit.

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 ist überhaupt kein Problem, weil der Adapter alle konkreten Details abschirmt. Was der Wechsel nicht mitbringt, ist die Migration der vorhandenen Issues. Der Adapter übersetzt Operationen, er transportiert keine Daten.

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

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 mehreren Formaten im Repo. Das ist machbar, und es ist nicht Be-

standteil des Kits. Der Grund ist derselbe wie bei Jira: Ich habe es nicht gebraucht. To be fair: Das ist eine typische Aufgabe, die ein Sprachmodell wunderbar und akkurat lösen kann.

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 liegt als `.claude/kit/night.mjs` im Repo und kommt mit dem Installer. 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.

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. Auch hier kann der Runner nur dann entscheiden, ob das Fundament trägt, wenn dieser Abschnitt ordentlich gepflegt wird.

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.

Starten

```
node .claude/kit/night.mjs --dry-run   # zeigt an, was liefe,
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
`--dry-run`	Nur anzeigen, nichts starten	aus
`--max <N>`	Session-Limit pro Nacht	10
`--model <id>`	Modell für den Lauf	stärkstes verfügbares
`--timeout-min <N>`	Zeitlimit pro Runde	60
`--label <name>`	Routing-Label, `none` schaltet den Filter ab	`kit:nightrun`
`--verbose`	Live-Verlaufsprotokoll	aus
`--no-checks-ok`	Start trotz leerer `buildChecks`	aus

Ohne `--verbose` protokolliert der Runner pro Runde nur Start und Ende. Bei einer langen Session sieht man dann nicht, woran sie gerade arbeitet, weil die Abschlussnachricht erst am Ende 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.

Das löst ein reales Problem. 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, was der Default mit seinem Namespace-Präfix erfüllt.

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
`buildChecks` leer	Kein Start, außer mit `--no-checks-ok`
Checks rot	Issue zurück ins Backlog, nicht nach In review
Issue hängt beim Start in In progress	Runner stoppt, statt zu raten
Unerfüllte `#N`-Abhängigkeit	Issue wird zurückgestellt
Titel mit `[Fachlich]`	Kommentiert zurück ins Backlog, ohne Session
Session-Limit erreicht	Lauf endet, Rest bleibt in Ready

Die vorletzte Zeile ist die mechanische Absicherung der PO-Schleife aus Kapitel 11. Ready heißt implementierbar, und ein fachliches Issue ist es nicht.

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 eingecheckt und wird geteilt. Der Adapter bricht ab, wenn er dort eines findet, statt das Secret still zu benutzen.

Berechtigungen

Ein Punkt, der beim ersten Einrichten Zeit kostet. Eine unbeaufsichtigte Session kann nicht zurückfragen, ob sie ein Kommando ausführen darf. Damit der Lauf nicht an der ersten Rückfrage hängen bleibt, müssen die benötigten Kommandos vorab erlaubt sein, und Kommandos, die aus der Sandbox ausbrechen müssen, etwa weil sie einen Docker-Daemon brauchen, davon ausgenommen werden.

Die genaue Schreibweise dieser Einträge hängt von der Claude-Code-Version ab und steht in der Online-Doku. Wer beim ersten Nachtlauf morgens ein Protokoll findet, in dem nichts passiert ist, 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:*)"
    ]
  }
}
```

Was der Nachtbetrieb nicht ist

Der Nachtbetrieb ist keine Verdopplung der Arbeitszeit. Das reine Implementieren von Code kann aber vollständig automatisch laufen, wenn die Leitplanken stimmen. Von daher spricht nichts dagegen, diese Zeit zu sparen. Sie wird an anderen Stellen dringend benötigt.

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. Es ist eine Aussage darüber, 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 außerdem dazu 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 dieselben Vereinfachungen noch einmal vornehmen. Ein zweites Modell hilft, weil es andere blinde Flecken hat. Es hilft nicht, weil es zuverlässiger wäre. Und 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. Zwei Modelle haben deshalb wahrscheinlich eine große Schnittmenge, was blinde Flecken angeht.

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. Man 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 ihnen 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" ist keine Aussage über den Code, sondern über die Ähnlichkeit zu bekannten Mustern.

Dazu kommt ein Befund, der die Sache verschärft: **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. Man lässt ein Modell Code erzeugen, nimmt diesen Code als Lernmaterial für die nächste Version desselben Modells, und wiederholt das. Ohne Prüfung sinkt die Trefferquote am schnellsten. Mit Compiler und statischen Prüfregeln wird der Verfall langsamer, er hört aber nicht auf. Der Code kompiliert weiterhin und wird trotzdem schlechter.

Der dritte Befund ist der interessante: Lässt man 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.

Aus der Bildforschung kennt man 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. Pull Requests von KI-Agenten sind von rund 4 Millionen im September 2025 auf über 17 Millionen im März 2026 gestiegen. Diese Repositories sind das Lernmaterial für die nächste Modellgeneration. Was darin steckt, misst GitClear an 200 Millionen geänderten Zeilen: 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 ist von 9,4 auf 15,7 Prozent gestiegen.

Was in so einem Kreislauf zuerst verschwindet, ist nicht die Korrektheit, es ist die Bandbreite: die ungewöhnliche Lösung, der elegante Sonderweg, das Stück Code, von dem man beim Lesen etwas lernt. Übrig bleibt der Durchschnitt. Das Fazit der Autoren, sinngemäß: Stabiles Weitertrainieren braucht eine Prüfinstanz, die nicht am Modell hängt. Es 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. Was ich hier nenne, ist der Stand 2026 und als Ausgangspunkt gedacht, nicht als Rangliste.

Kategorie	Wogegen	Werkzeuge
Gesamtqualität, zentral	Code Smells, Duplikate, Trends über Zeit	SonarQube, SonarCloud
Secrets		gitleaks, TruffleHog

Kategorie	Wogegen	Werkzeuge
	API-Keys, Tokens, Zertifikate im Diff	
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

Von diesen ist SonarQube das Werkzeug mit dem breitesten Nutzen, 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. Wer das einsetzt, bekommt vom Modell keinen Code mehr, der Schichtengrenzen verletzt. Der Build schlägt fehl, das Modell muss es richtig machen. Das ist der Unterschied zwischen einer Regel im Kopf und einer Regel im

Build, und es ist der Unterschied zwischen Wunsch und Wirkung. 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

Eines meiner wichtigsten Werkzeuge ist SonarQube. Mit SonarQube mache ich tatsächlich eine Rückkopplung: Alle Befunde, die SonarQube findet, werden zurück ins Backlog gespiegelt. Dort kann ich sie priorisieren und entscheiden, ob ein Befund ein Fehlalarm ist, ob ich ihn akzeptieren kann oder ob er im nächsten Schritt mitbearbeitet werden muss.

Wenn SonarQube in einem Lauf neunzehn Issues findet, dann finde ich diese neunzehn Punkte anschließend als Karten auf meinem Board wieder. Damit werden sie zu dem, was jede andere Arbeit auch ist: etwas, das ich priorisieren, schneiden und freigeben kann. Wichtig ist, dass sie nicht nur in einem Dashboard angezeigt werden, sondern tatsächlich ins Backlog wandern, wo man immer wieder über sie stolpert.

Der Effekt darauf, wie ich arbeite, ist größer, als es klingt. Technische Schuld wird sonst zu einer Sache, die man irgendwann mal angeht, 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 ausschließlich Refactorings und Bugfindings aus SonarQube abgearbeitet werden. 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.

Ein Wort zur Vorsicht: 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 wird morgens besprochen.

Schwellenwerte

Meine Zielwerte sind unbequem, und ich vertrete sie trotzdem.

Coverage: 100 Prozent oder ein begründeter, dokumentierter Ausschluss. Der Ausschluss ist eine Entscheidung, kein Kompromiss. Ein unbegründeter Ausschluss ist ein Leck.

Mutation Score: so hoch wie möglich, und wenn priorisiert werden 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: Sie werden gesehen und im SonarQube wissentlich akzeptiert. Was nicht geht, ist die offene Liste, denn die wird innerhalb von Wochen zur Liste mit ignorierten 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, statische Analyse werden im Maven- oder npm-Build als verbindliche Gates eingerichtet. Ein Build, der nicht grün ist, blockiert. Diese Verdrahtung ist die eine Investition, die am Anfang Zeit kostet und über Monate Zeit spart.

In der Config. Die Kommandos wandern in `buildChecks`. `/local-check` führt sie nacheinander aus, ein roter Check blockiert den weiteren Prozess, und `/push-main` ist mechanisch blockiert, solange kein grüner Check vorliegt. Der Nacht-Runner startet ohne `buildChecks` gar nicht erst.

Im CI. Dieselben Checks laufen noch einmal auf dem Server, plus die, die lokal zu lange dauern. Der Grund ist einfach: 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

Nachdem ich beschrieben habe, was es nicht kann, gehört die andere Hälfte dazu, denn sonst klingt es so, als sei das Review 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 Arbeitsteilung ist also klar: 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, und das ist eine echte Verbesserung, weil sich damit ein Golden Path automatisiert durchlaufen lässt.

Was sich dadurch nicht ändert, ist die Zuständigkeit. 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, eine Eins-zu-eins-Abbildung der Produktionsumgebung, auf der ich alles testen kann. Und zum Schluss geht es auf Produktion.

Wenn diese Verifikation nicht möglich ist, weil kein Docker läuft oder kein lokaler Build erreichbar ist, wird das im Abschlussbericht vermerkt. 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, ist nach einem halben Jahr eine Beschreibung dessen, was einmal galt. Dieser Teil handelt davon, wie er am Leben bleibt.

Kapitel 21 zeigt, wie sich der Prozess in einem neuen Team einführen lässt, 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, aber es macht den Unterschied zwischen einem Team, das mit KI arbeitet, und einem Team, das von der KI bearbeitet wird.

Klein anfangen

Mein erster Rat widerspricht dem Rest dieses Buches, und deshalb steht er zuerst: Fang ohne Infrastruktur an.

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 dem Aufbau der Infrastruktur beginnt, 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, wer sie einmal sorgfältig erstellt hat, übernimmt sie ins nächste.

2. Ein Board mit fünf Spalten. Backlog, Ready, In progress, In review, Done. Wichtig ist nicht das Werkzeug, wichtig ist die Konvention dahinter: 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 werden in der Workflow-Datei festgehalten und sind für alle verbindlich. 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`. Das ist die eine Investition, die am Anfang Zeit kostet und über Monate Zeit spart. Kapitel 20 sagt, warum sie nicht optional ist.

5. Ein zweites Modell für Reviews festlegen. Welches, ist zweitrangig. Dass es ein anderes ist als das implementierende, ist der Punkt. 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, und das ist es auch. Es ist zugleich der erste Praxistest des Prozesses: Plan, Issue, GO, Implementierung, Review, Push. Wer hier einmal durchläuft, kennt den Prozess 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

Tag	Was
Danach	Nachtbetrieb, wenn die Issues sauber genug geschnitten sind

Der Nachtbetrieb steht bewusst am Ende. Er verzeiht keine unklaren Issues, und die Fähigkeit, Issues eindeutig zu schneiden, entsteht 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 werden als Bremse empfunden und weggelassen. Meistens beginnt es beim GO, weil der Plan ja schon abgenommen war. Kapitel 22 handelt davon.

Es gibt keinen Ort, an dem der Prozess sich ändern darf. Dann wird er entweder starr befolgt, bis er nicht mehr passt, oder still aufgegeben. 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 dadurch konkret, dass er die typischen Verletzungen benennt. Die folgenden Anti-Patterns sind alle real beobachtet, die meisten bei mir selbst. Wer sich beim Arbeiten dabei ertappt, einen dieser Gedanken zu denken, sollte anhalten.

“Plan-Approval reicht, ich fange einfach an”

Die häufigste Verletzung überhaupt. Ein Plan wird akzeptiert, das Modell interpretiert das als Implementierungsfreigabe und beginnt zu coden.

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

Der Grund für die Trennung ist nicht Förmlichkeit. Zwischen Plan und GO 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 ist menschlich und trotzdem falsch. 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.

“Tests folgen später”

Sie folgen nicht. Was später erledigt werden soll, wird nicht erledigt. 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. Der Test ist die Stelle, an der das Modell an die Realität gebunden wird.

“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, und nicht gut darin, seine eigenen Fehler zu erkennen. Verifikation ist Menschenpflicht.

“Bei komplexen Aufgaben gebe ich mehr ab”

Der verbreitetste Denkfehler und der teuerste. Er klingt vernünftig: Je schwerer die Aufgabe, desto mehr Hilfe.

Er ist falsch herum, und Kapitel 3 führt aus, warum. Woran man merkt, dass man gerade dabei ist: Man liest einen Vorschlag und nickt, statt ihn nachzurechnen. Man fragt das Modell, wie etwas funktioniert, statt es zu prüfen. Man akzeptiert einen Plan, dessen Begründung man 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 GO. Jeder einzelne Schritt wirkt sinnvoll, und in der Summe ist der Prozess weg.

Wichtig ist die Reaktion darauf. Der Code war beide Male in Ordnung. Es war kein technischer Bug, es war ein Prozess-Bug. Zurückgerollt gehört nicht der Code, sondern der Workflow.

Memory-Drift

Memory-Dateien werden ergänzt, aber nie gepflegt. Das Ergebnis sind alte Regeln bei neuer Realität. Kapitel 17 beschreibt den Mechanismus, Kapitel 23 den Ort, an dem er geprüft wird.

Halluzinationen als verifizierte Wahrheit

Das Modell erfindet plausibel klingende Bibliotheksnamen, Methodensignaturen, Konfigurationsoptionen. Wer eine solche Erfindung ungeprüft übernimmt, baut Phantomschulden ein, die irgendwann mit Zinsen zurückgezahlt werden.

Halluzinationen sind keine Ausnahme, sie sind eine Betriebsbedingung. Jede Behauptung mit externem Bezug gehört geprüft.

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. Verschiedene Modelle haben verschiedene 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.

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 skaliert die Konsequenzen jeder Entscheidung, die niemand getroffen hat. Stop-Punkte sind keine Bequemlichkeitseinschränkung, sie sind Strukturelement.

Das gefährlichste Muster

Zum Schluss das, vor dem ich mich am meisten hüte, weil es sich nicht wie ein Fehler anfühlt: der Tag, an dem alles glatt läuft.

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

KAPITEL 23

Die KI-Retrospektive

Die KI-Retrospektive ist die Neuerung im Eventkanon, und sie ist die einzige Veranstaltung, in der sich der Prozess selbst überarbeitet.

Sie ist nicht ein Tagesordnungspunkt der allgemeinen Retrospektive. Das ist keine Förmlichkeit. Die klassische Retro spricht über Menschen und Zusammenarbeit, und wenn die Mensch-Modell-Fragen dort als letzter Punkt auftauchen, fallen sie der Zeit zum Opfer oder werden zu allgemein beantwortet.

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 wurde 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 unterschiedlich ausgelegt werden kann?

Der Output sind Dateien, keine Erkenntnisse

Das ist die Regel, an der sich entscheidet, ob die Retrospektive etwas taugt.

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 unterstützt das, indem er die drei Fragen stellt und die vorgeschlagenen Änderungen als Diff formuliert. Was er nicht tut, ist entscheiden, welche davon übernommen wird.

Was regelmäßig herauskommt

Nach einiger Zeit kristallisieren sich Muster heraus, und 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 ersetzt wurde. Solche Einträge zu streichen ist genauso wichtig wie neue zu schreiben, denn jede Zeile in einer Konventionsdatei kostet Kontextfenster.

Regeln, die ins Werkzeug gehören statt in eine Datei. Das ist das Ergebnis mit der größten Wirkung. Wenn dieselbe Regel dreimal gebrochen wurde, ist sie am falschen Ort. Eine Architekturregel gehört nicht in eine Konventionsdatei, sondern in einen ArchUnit-Test, der den Build bricht. Die Retrospektive ist die Stelle, an der diese Verschiebung auffällt.

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, und das macht es stärker, nicht komplizierter.

Der praktische Grund ist einfach: Die Werkzeuge verschieben sich monatlich. Ein Prozess, der sich nicht in demselben Takt überarbeiten darf, ist nach einem halben Jahr eine Beschreibung dessen, was einmal galt. Wer in der KI-Ära nicht regelmäßig reflektiert, sammelt Memory-Drift an: alte Regeln, neue Realität, wachsende Reibung.

Deshalb ist auch dieses Buch versioniert, und deshalb sagt Anhang E, wie es entstanden ist und nach welchen Regeln es überarbeitet wird. Es wäre inkonsequent, einen Prozess zu beschreiben, der sich selbst überarbeitet, und den Text darüber unverändert stehen zu lassen.

KAPITEL 24

Ausblick

Alles in diesem Buch ist eine Momentaufnahme. Die Werkzeuge ändern sich monatlich, die Best Practices entsprechend. Vier Entwicklungen halte ich für absehbar.

Mehr Autonomie

Was heute als Implementations-Engine arbeitet, wird morgen in Mehr-Agenten-Aufstellungen arbeiten: ein planender Agent, ein implementierender, ein testender, ein dokumentierender, koordiniert von einem Menschen.

Der hier beschriebene Prozess ist auch in einer solchen Welt anwendbar, er muss nur in einigen Rollen anders besetzt werden. Was sich nicht ändert, ist die Grundregel: 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, und die Erfahrung daraus ist eindeutig: Der Gewinn kam nicht davon, dass mehr gebaut wurde. Er kam davon, dass ich abends sauberer schneiden musste.

Continuous AI wird Normalität

Was heute als Vision formuliert wird, ist in zwei Jahren selbstverständlich. Repositories werden Hintergrund-Loops haben, in denen kleine Verbesserungen kontinuierlich entstehen.

Das macht die Trennung zwischen menschlich getakteter Iteration und KI-getakteter Hintergrundarbeit wichtiger, nicht unwichtiger. In den Hintergrund gehört, was kein menschliches Urteil im Moment der Entstehung braucht. 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 etwa neun von zehn Fällen 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 Kundenauflagen.

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 ist ein Befund mit Verfallsdatum, und ich rechne damit, dass er als Erstes von allem in diesem Buch überholt sein wird.

Die Nachwuchsfrage bleibt ungelöst

Die Entwicklung, über die am wenigsten gesprochen wird und die mich am meisten beschäftigt.

Die Karrierepyramide bricht an der Basis weg. Wenn die Nachwuchspipeline jetzt austrocknet, fehlt in fünf Jahren nicht der Nachwuchs, sondern die Erfahrung. Und Senior-Kompetenz braucht Jahre, die kann man nicht zur nächsten Hauptversammlung nachbestellen.

Ich habe darauf keine vollständige Antwort. Was ich habe, sind 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 eigenen Einschätzung kommt von Heiko Dietz, dessen Einwand in der Einleitung steht. Seine These reicht deutlich weiter als meine: Verdrängt werde nicht nur der Nachwuchs, 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 orchestrieren, 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 zu nehmen ist 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 ist der wirtschaftliche Teil aus zwei Gründen. Erstens setzt die These voraus, dass die Nachfrage nach Software ungefähr konstant bleibt und der Zuwachs an Produktivität deshalb Stellen freisetzt. Das war in der Geschichte dieses Berufs noch nie so. Bisher hat jede Senkung der Herstellungskosten dazu geführt, dass mehr Software gebaut wurde, nicht dass dieselbe Menge billiger entstand.

Zweitens setzt sie voraus, dass die zwei oder drei orchestrierenden Personen einfach da sind. Sie sind aber genau die Senioren, die aus der Breite hervorgehen, die verschwinden soll. Eine Branche, die nur noch Spitzenkräfte beschäftigt, muss erklären, woher die nächste Generation Spitzenkräfte kommt. Das ist die Nachwuchsfrage aus dem vorigen Abschnitt, nur eine Etage höher.

Wer recht behält, entscheidet keine Prognose. Das entscheiden die nächsten Jahre. Ich stelle seine Position hierhin, weil sie von jemandem kommt, der beide Seiten kennt: 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, und die folgenden Fragen sind offen:

Wie balanciert man KI-Autonomie und menschliche Aufsicht in Echtzeit, nicht nur an Stop-Punkten? Wer trägt die Verantwortung, wenn der Code von einem Modell stammt: die reviewende Person, die PO-Rolle, das Un-

ternehmen? 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 hier beschriebene Prozess 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 niemand verantwortete Software hat eine Lebenserwartung, die in Wochen zu rechnen ist.

Was bleiben wird

Zum Schluss die drei Sätze, auf die sich alles zusammenziehen lässt.

KI braucht Leitplanken, und diese Leitplanken sind in vierzig Jahren Software Engineering entstanden. 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 sehr mächtiges Werkzeug, und genau 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`	1.5	Fachliches Issue im Story-Format, technikfrei	Kein technischer Plan, keine technischen Issues
`/plan`	2	Plan aus Anforderung oder aus `#N`	Implementiert nichts, legt keine 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
`/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
`/local-check`	6	`buildChecks` und `mutationCommand` ausführen	Übergeht keinen roten Check
`/review`	7	Review in frischer Session, zweites Modell	Ersetzt keine Security-Werkzeuge im CI
`/push-main`	8	Commit-Batch auf main pushen	Startet nie selbsttätig

Skill	Schritt	Was er tut	Was er nicht tut
`/merge-production`	9	PR von main nach production	Merged nicht selbst

Querschnitt

Skill	Wann	Was er tut
`/issue-review`	Zwischen 3 und dem GO	Issue von zwei Modellen prüfen lassen, geschärften Body vorschlagen
`/kontext`	Session-Start	Vault, Projektnotiz und offene Issues laden, Lagebericht ausgeben
`/document`	Session-Ende	Tageslog schreiben, Projektnotiz aktualisieren
`/retro`	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	**GO: Issues nach Ready**	Am Board
Zwischen 8 und 9	Test-Server prüfen	Im Browser
9	**Merge**	Im Pull Request

Der Nacht-Runner

Kein Skill, ein Node-Skript: `.claude/kit/night.mjs`. Startet pro Issue eine frische Session mit `/implement-next`. Details in Kapitel 19.

Flag	Wirkung
`--dry-run`	Zeigt an, was liefe, startet nichts
`--max <N>`	Session-Limit pro Nacht
`--model <id>`	Modell für den Lauf
`--timeout-min <N>`	Zeitlimit pro Runde
`--label <name>`	Routing-Label, `none` schaltet den Filter ab
`--verbose`	Live-Verlaufsprotokoll
`--no-checks-ok`	Start trotz leerer `buildChecks`

ANHANG B

Config-Referenz

Zwei Config-Dateien, zwei Zuständigkeiten. `workflow.config.json` ist repo-spezifisch und wird geteilt. `kontext.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
`codeHost`	`github`, `gitlab`, `local`	Wo Repository und Pull Requests liegen
`issueTracker`	`github`, `gitlab`, `local`, `toolbox`	Wo Issues und Board-Bewegungen liegen. `toolbox` ist der historische Name für das kanban-kit, siehe Kapitel 5
`buildChecks`	Liste von Kommandos	Was `/local-check` nacheinander ausführt
`mutationCommand`	Kommando oder `""`	Mutation Testing, separat wegen Laufzeit
`mainBranch`	Branch-Name	Ziel von `/push-main`
`productionBranch`	Branch-Name	

Feld	Werte	Bedeutung
		Ziel von `/merge-production`
`reviewScope`	`diff`, `full`	Umfang, den `/review` zu sehen bekommt
`reviewModel`	Modell-ID	Modell für die Review-Session
`triggers.go`	Phrase	Freigabe zur Implementierung
`triggers.push`	Phrase	Freigabe zum Push
`triggers.merge`	Phrase	Freigabe für den PR nach production
`issueReview.rounds`	Zahl, Default 1	Prüfrunden je Issue
`issueReview.requiredBeforeReady`	`true`, `false` (Default)	Ready-Issues ohne Review-Marker zurück ins Backlog
`issueReview.reviewers`	Liste aus `name`, `kind`, `model` oder `cmd`	Prüfer. `kind: claude` läuft im Modell, `kind: command` als CLI über Standardein- und -ausgabe
`issueReview.pairs`	Autor-Name auf zwei Prüfer-Namen	Wer wen prüft. Ohne Eintrag greifen die ersten zwei Prüfer, die nicht der Autor sind

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": { "go": "GO", "push": "push main", "merge": "merge
production" }
}
```

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 eingecheckt und wird geteilt.

kontext.config.json

Global in `~/.claude/`, optional lokal in `.claude/`. Beide werden gemergt, lokale Werte gewinnen.

Feld	Typ	Bedeutung
`vault`	Pfad	Absoluter Pfad zum Memory-Vault. Fehlt er, läuft der Degraded Mode
`always`	Liste	Dateien relativ zum Vault, die immer gelesen werden
`projectDocs`	Liste	Dateien oder Glob-Muster relativ zum Projekt
`project`	Text	Override, wenn Repo-Name und Vault-Ordner abweichen

```
{
  "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
```

kanban-kit: Umgebungsvariablen

In einer `.env` neben der `docker-compose.yml`. Details in Kapitel 14.

Variable	Default	Bedeutung
`MANBAN_BASE_URL`	`https://localhost`	Basis-URL für Links in E-Mails
`MANBAN_BOOTSTRAP_ADMIN_TOKEN`	leer	Einmal-Token für den ersten Admin
`MANBAN_MAIL_ENABLED`	`false`	Echten Mailversand aktivieren
`MANBAN_CLEANUP_ENABLED`	`true`	Done-Archivierung und Papierkorb-Leerung
`MANBAN_DONE_RETENTION_DAYS`	`30`	Tage bis Done-Karten archiviert werden
`MANBAN_SESSION_SECRET`	Dev-Default	In Produktion zwingend setzen
`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.

Board-Adapter `board.mjs`. Schirmt die Skills von der konkreten Plattform ab. Kennt Issue anlegen, auflisten, lesen, verschieben, kommentieren. 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. Kein Vergessen, sondern eine Umgewichtung der Aufmerksamkeit. 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 konfigurierten Vault. Funktioniert vollständig, nur ohne projektübergreifendes Memory.

GO Der erste Stop-Punkt, Schritt 4. Die menschliche Freigabe zur Implementierung, am Board sichtbar als Bewegung nach Ready. Nicht zu verwechseln mit der Zustimmung zu einem Plan.

Fachliches Issue Issue mit `[Fachlich]`-Präfix. Beschreibt in PO-Sprache das Was und Warum, technikfrei. Wird gegroomt, nie implementiert. Siehe PO-Schleife.

Ideen-Speicher Zone im kanban-kit für Karten, die noch nicht auf dem Board sichtbar sein sollen. Ungesichtete Rohanforderung.

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.

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 die Ready-Spalte unbeaufsichtigt ab, mit einer frischen Session pro Issue. Pusht nie.

PO-Schleife Optionale Trennung in fachliche und technische Issues, damit ein Product Owner fachlich abnehmen kann, bevor Technik entsteht.

Prompt-Bibliothek Im Repository gepflegte, wiederverwendbare Anweisungen an das Modell. Veraltet schnell und gehört regelmäßig inspiziert.

Ready Die Spalte, in der ein Issue implementierbar ist. Der Kern des Boards. Bewegungen dorthin macht ausschließlich der Mensch.

Routing-Label Standardmäßig `kit:nightrun`. Markiert auf einem Board die Teilmenge der Ready-Issues, die der Nacht-Runner bearbeiten darf.

Stop-Punkt Eine der drei Stellen, die bewusst nicht automatisiert sind: das GO, der Push, der Merge.

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 Persönlicher Memory-Speicher außerhalb des Repos für projektübergreifendes Wissen. Optional.

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` Quellcode: `github.com/mannewolff/claude-workflow-kit`
- **kanban-kit**, Dokumentation: `kanban.mwolff.org/docs` Quellcode: `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` 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`
- Blog: `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`
- 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` 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` 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, `npm audit`

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 Prüfregeln 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` Quelle der Zahlen zu verschobenem und kopiertem Code, gemessen an über 200 Millionen geänderten Zeilen.
- Die Zahlen zu Pull Requests von KI-Agenten, von rund 4 Millionen im September 2025 auf über 17 Millionen im März 2026, stammen aus der Berichterstattung von The Information, 2026.

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` 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` 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.

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/`

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

Autor und inhaltlich verantwortlich Manfred Wolff Butjadinger Straße 34a 28197 Bremen

info@mwolff.org mwolff.org · 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 und kanban.mwolff.org Quellcode: 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.