XBtech

SOFTWARE FACTORY

Wie unsere Software Factory arbeitet

Fabio Möller, Manfred Wolff, xB Technologies GmbH

01. Unsere Software Factory auf einen Blick

Unsere Software Factory macht aus einer rohen Idee ein Release, und das Modell erledigt dabei fast alles. Fast. Drei Handgriffe gehören uns: das Ready, der Push und der Merge.

Unsere Software Factory ist kein Produkt mit Login. Sie besteht aus zwei Werkzeugen und den Leitplanken im Build. Das KI-Workflow-Kit, als Open Source unter dem Namen claude-workflow-kit erhältlich, ist eine Sammlung von Skills für Claude Code, jeder Skill ein Arbeitsschritt. Der KI-Leitstand ist unser selbst gehostetes Board, auf dem der Status jeder Karte steht, für uns und für die Maschine.

Im Kopf haben wir dafür das Bild einer Fertigungsstraße. Das Board ist das Fließband, jede Karte ist ein Werkstück, und jeder Skill ist eine Station, an der genau eine Sache passiert. Oben in der Straße wird gedacht, unten wird gebaut.

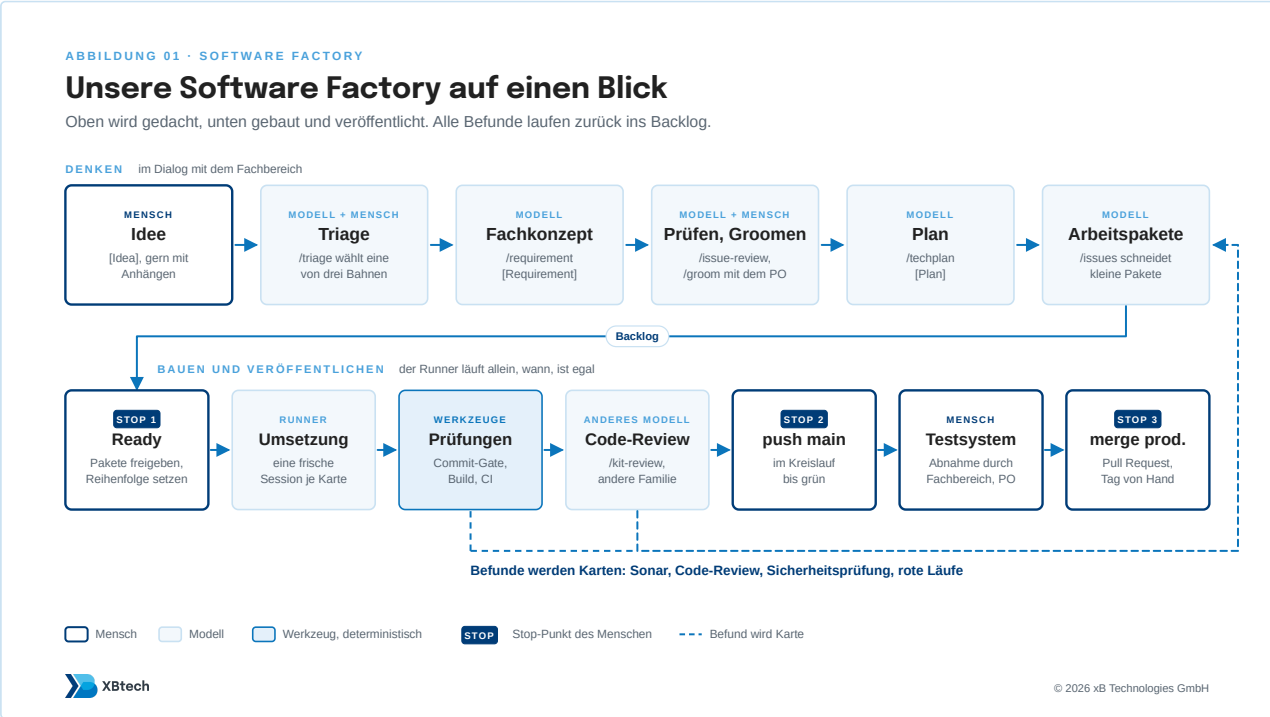

Abbildung 1: Unsere Software Factory auf einen Blick. Oben wird gedacht, unten gebaut, und alle Befunde laufen zurück ins Backlog.

© 2026 xB Technologies GmbH

Der Grund für diesen Aufbau ist eine einfache Beobachtung. Die Implementierung ist billig geworden, ein Arbeitspaket setzt das Modell bei uns in etwa zehn Minuten um. Billig geworden sind das Verstehen und das Prüfen nicht. Deshalb haben wir die Factory vor und nach der Umsetzung am dichtesten ausgebaut.

Wir gehen die Straße jetzt einmal ab, mit einer Karte, von der Idee bis in die Produktion.

02. Am Anfang steht Gebrabbel

Jede Anforderung beginnt bei uns als Idee. Ein paar diktierte Sätze, eine weitergeleitete Mail, ein Screenshot. Das legen wir als Karte mit dem Präfix [Idea] aufs Board, die Anhänge gleich mit. Eine Idee hat kein Format, und sie wird nie umgesetzt.

Wenn wir Zeit dafür haben, rufen wir `/triage` auf. Der Skill liest die Idee samt Anhängen, legt uns alle unklaren Punkte auf einmal vor, jeden mit einer Empfehlung, und schlägt eine Bahn vor. Wir widersprechen nur dort, wo es nötig ist. Entscheiden tun wir.

Bahn 1 ist die Kleinigkeit: eine Datei, ein Config-Wert, kein Datenmodell, keine Sicherheitslogik. Die setzt das Modell direkt um, ein Commit, fertig. Bahn 3 ist der klare Weg, der nur länger ist als ein Commit. Daraus macht `/task` genau ein Arbeitspaket. Bahn 2 ist alles, bei dem es mehr als einen vertretbaren Weg gibt, und dort beginnt die eigentliche Software Factory.

Im Zweifel nehmen wir Bahn 2. Der Fehler in diese Richtung kostet einen Plan, den wir nicht gebraucht hätten. Der Fehler in die andere Richtung kostet einen Nachmittag.

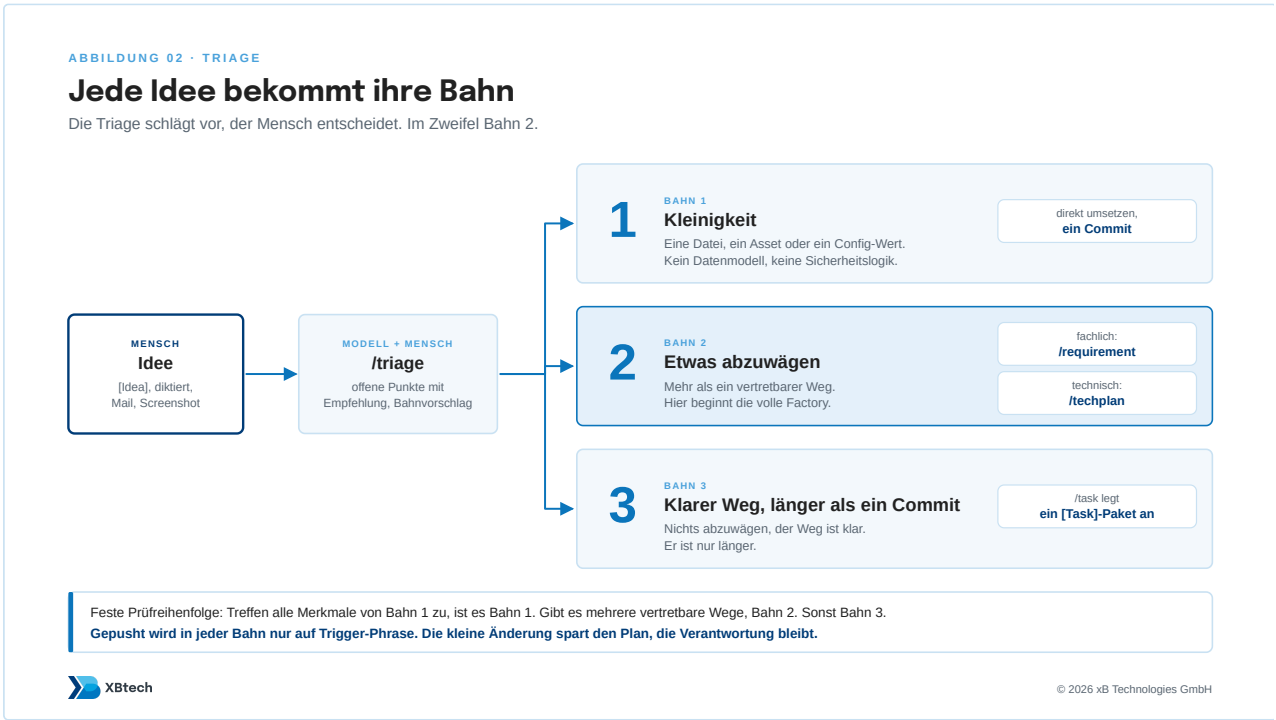

Abbildung 2: Die Triage bringt jede Idee auf eine von drei Bahnen. Gepusht wird in allen dreien nur auf unser Wort.

03. Die Denkarbeit wandert nach vorn

In Bahn 2 schreibt `/requirement` aus der Idee ein Fachkonzept mit dem Präfix [Requirement]. Es ist technikfrei und in der Sprache des Product Owners geschrieben: Ziel, fachliche Akzeptanzkriterien, Szenarien nach dem Muster Angenommen, Wenn, Dann, dazu Nicht-Ziele und offene Fragen. Bevor der Skill schreibt, sucht das Kit

frühere Fachkonzepte zum selben Thema. Widerspricht die neue Anforderung einer alten Entscheidung, fällt das auf, bevor es gebaut wird.

Danach kommt der Schritt, den wir am längsten unterschätzt haben: Wir lassen das Dokument prüfen, nicht den Code. Ein Fehler im Fachkonzept pflanzt sich in den Plan fort, von dort in jedes Arbeitspaket und am Ende in den Code. Im Fachkonzept gefunden, kostet er einen Satz. Deshalb prüfen wir oben am gründlichsten.

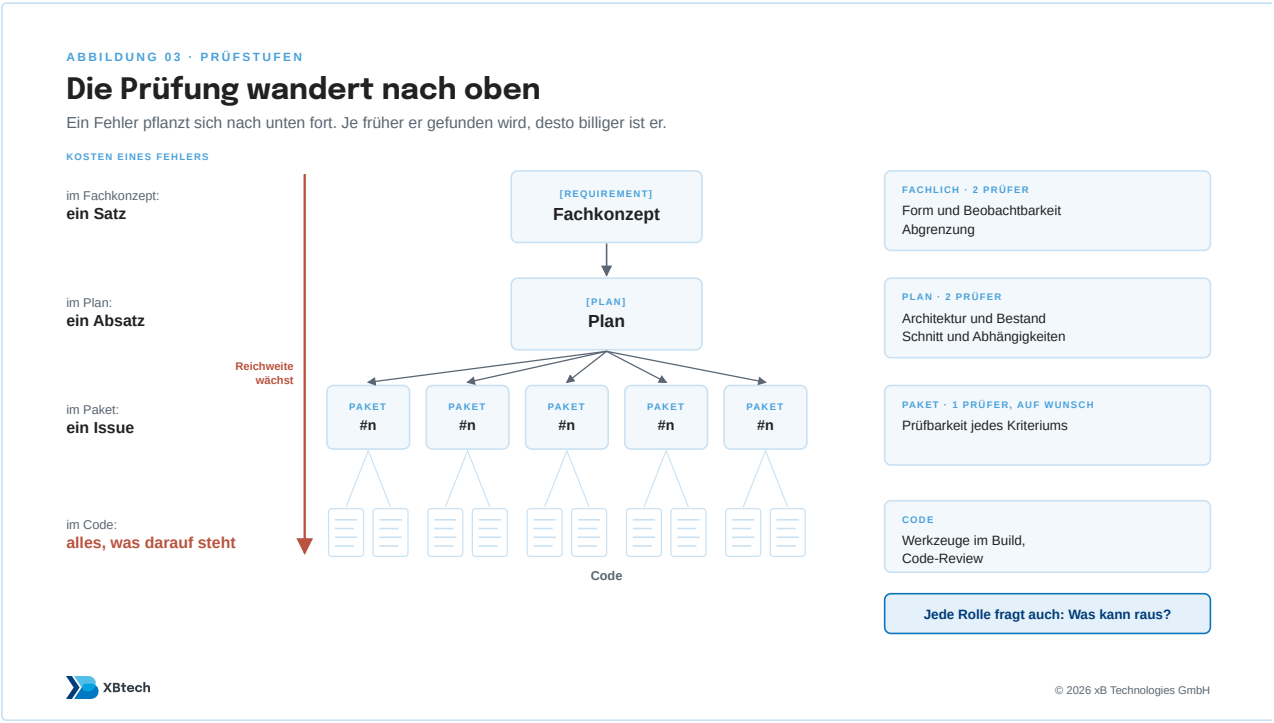

Abbildung 3: Die Prüfung wandert nach oben, weil die Reichweite eines Fehlers nach unten wächst.

Geprüft wird von Modellen, die das Dokument nicht geschrieben haben, und mindestens eines kommt aus einer anderen Modellfamilie. Zwei Modelle mit identischem Auftrag sind derselbe Blick zweimal. Deshalb hat jeder Prüfer eine eigene Rolle: beim Fachkonzept Form und Beobachtbarkeit und die Abgrenzung, beim Plan die Architektur gegen den Bestand und den Schnitt in Pakete. Jede Rolle stellt außerdem eine Frage, die sonst niemand stellt: Was kann raus?

© 2026 xB Technologies GmbH

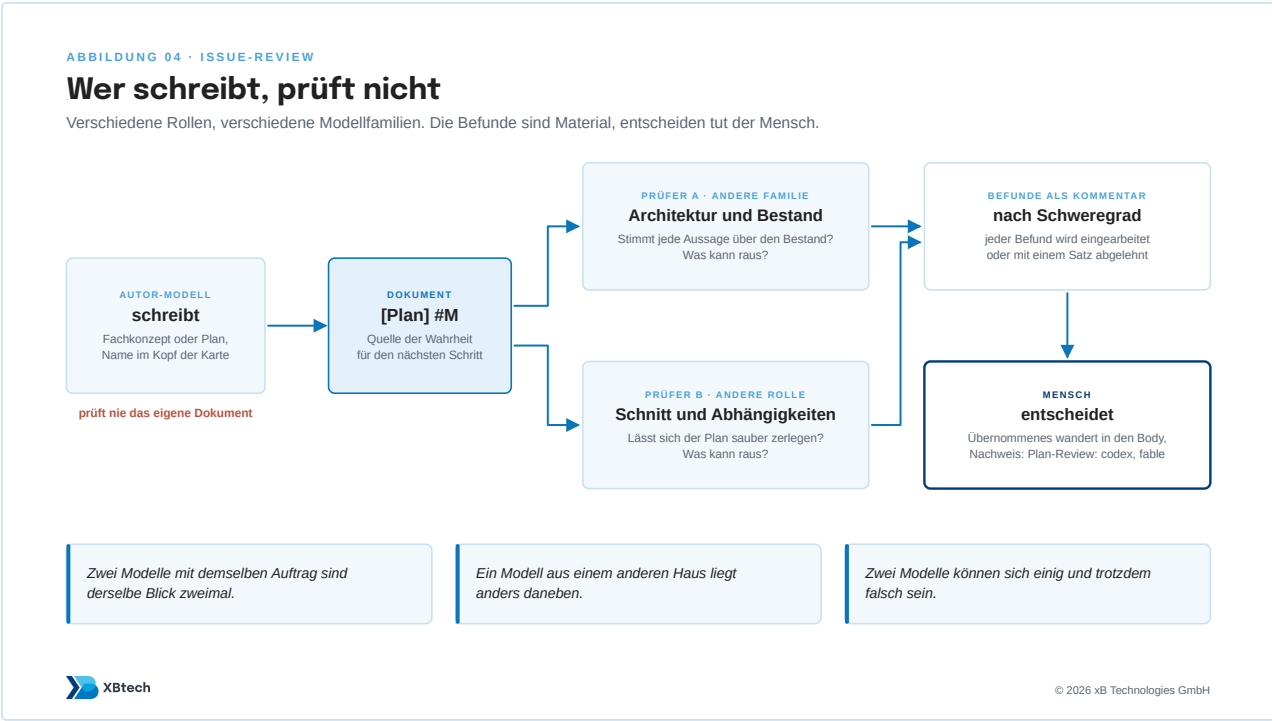

Abbildung 4: Wer schreibt, prüft nicht. Die Befunde sind Material, die Entscheidung bleibt beim Menschen.

Was danach offen bleibt, entscheiden wir mit `/groom`. Die Fragen kommen alle auf einmal, jede mit Empfehlung. Unsere Antworten wandern in den Body der Karte, nicht in Kommentare darunter, denn was nicht im Body steht, existiert für die Implementierung nicht. Erst wenn unter den offenen Fragen „Keine“ steht, schreibt `/techplan` den technischen Plan mit dem Präfix [Plan], und `/issues` schneidet ihn in Arbeitspakete.

Ein Arbeitspaket passt auf eine Seite und hat vier Abschnitte: Kontext, Aufgabe, Akzeptanzkriterium, Abhängigkeiten. Ab hier ist das Issue die Quelle der Wahrheit. Für den Runner gilt dabei eine strengere Regel als für die Arbeit im Dialog: Ein Paket muss ohne Rückfrage umsetzbar sein. Sitzen wir daneben, klären wir eine Unklarheit im Gespräch. Läuft der Runner allein, wird sie geraten.

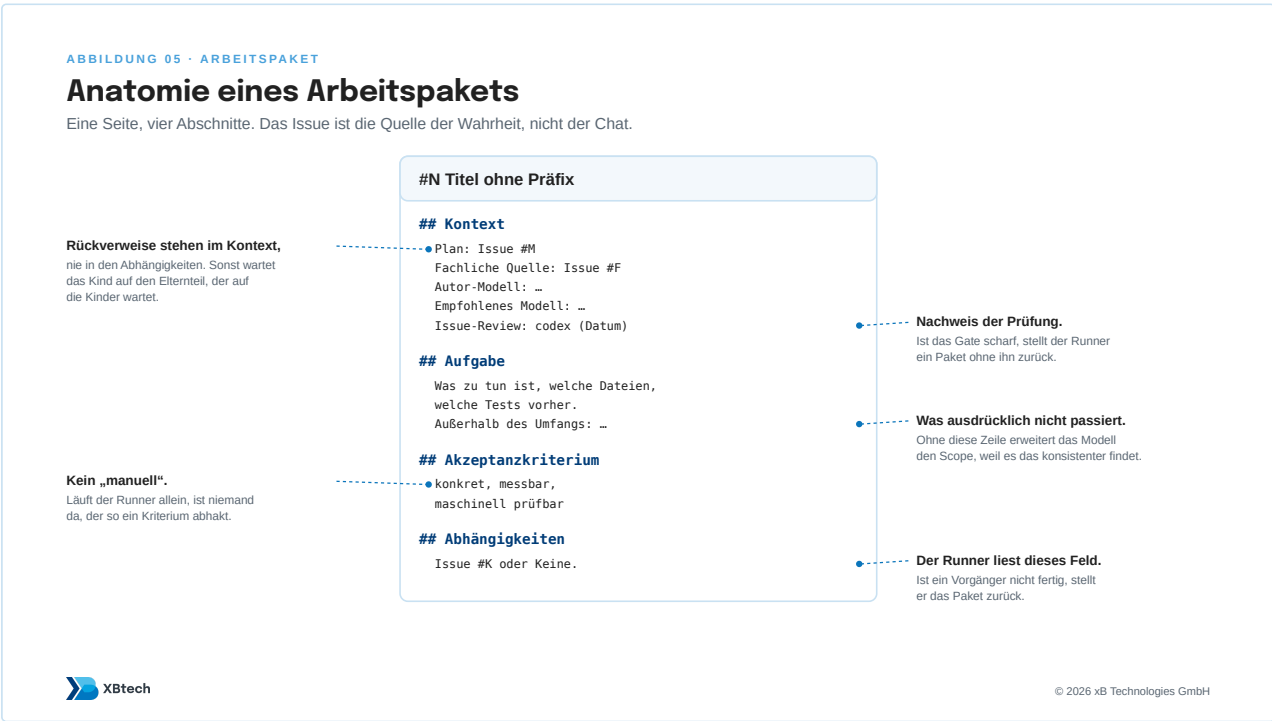

Abbildung 5: Anatomie eines Arbeitspakets. Die Rückverweise stehen im Kontext, nie in den Abhängigkeiten.

04. Das Ready: der Mensch gibt frei

Das Ready ist der erste Stop-Punkt. Wir ziehen die Pakete von Backlog nach Ready und sortieren sie, denn der Runner arbeitet die Spalte von oben nach unten ab. Das Label `kit:nightrun` markiert, was er anfassen darf. Der Name stammt noch aus der Zeit, als er nur nachts lief. In diesem Handgriff steckt unsere ganze Planung: wie viel Arbeit der nächste Lauf bekommt, in welcher Reihenfolge, mit welchen Abhängigkeiten.

Die Menge setzen wir so, dass wir das Ergebnis am selben Tag noch prüfen können. Den Takt bestimmt der langsamste menschliche Schritt, nicht der schnellste der Maschine.

Damit beim Ready nichts durchrutscht, steht vor der Umsetzung eine Schranke. Ideen, Fachkonzepte, Plandokumente und Menschenschritte mit dem Präfix [Human] sind nicht umsetzbar. Landet so eine Karte trotzdem in Ready, stellen die Skills und der Runner sie kommentiert ins Backlog zurück, ohne überhaupt eine Session zu starten. Wir verlassen uns nicht darauf, dass wir beim Ready aufpassen.

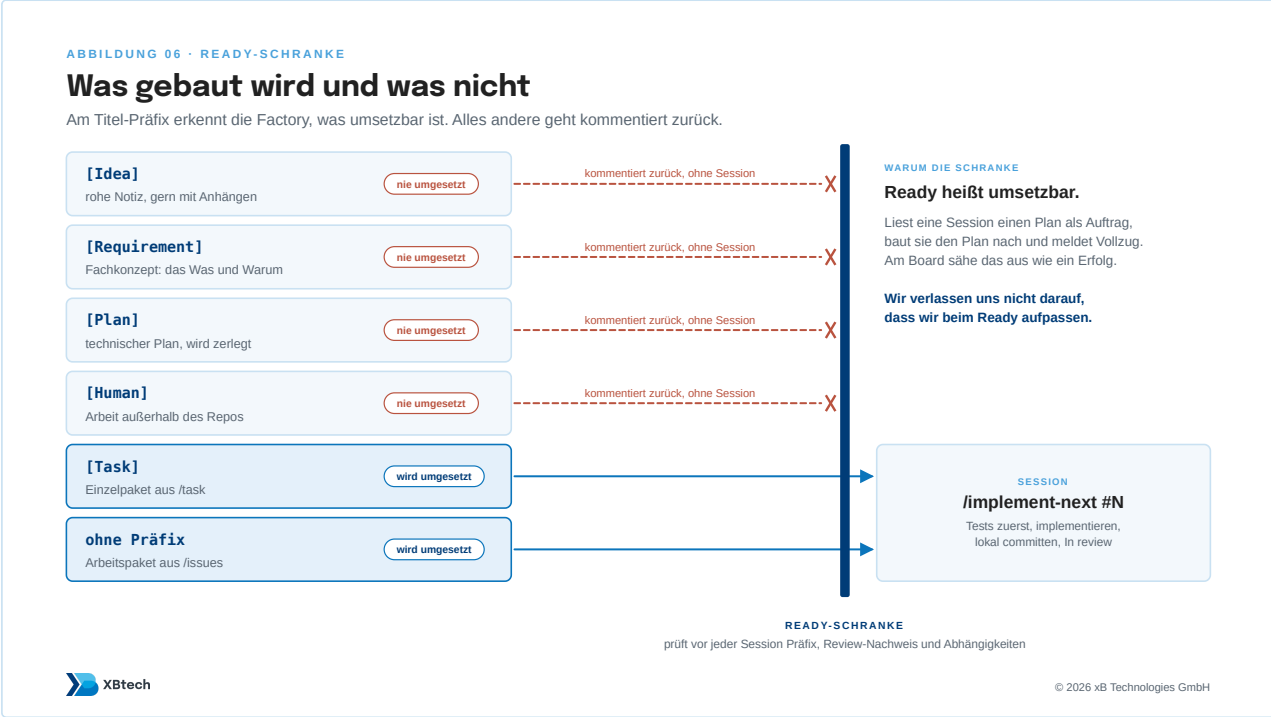

Abbildung 6: Die Ready-Schranke. Am Präfix erkennt die Factory, was gebaut werden darf.

05. Der Runner: eine Karte, eine Session

Der Runner arbeitet allein, und wann er läuft, ist egal. Wir starten ihn immer zuerst mit --dry-run. Der zeigt uns in zehn Sekunden, welche Karten er anfassen würde, welche er mangels Label überspringt und welche er wegen offener Abhängigkeiten zurückstellt.

Für jede Karte startet der Runner eine frische Session mit `/implement-next`. Darauf sind wir nicht am Reißbrett gekommen. Unser erster Versuch war ein einziger Chat für zwanzig Issues, und der scheiterte am Kontextfenster: Issue 17 entstand auf dem zusammengefassten Rest von 16 Vorgängern, und die Qualität sank genau dann, als niemand zuschaute. Heute ist jede Karte die erste und einzige ihrer Session. Was das Modell wissen muss, steht im Issue, in den Konventionsdateien und im Vault.

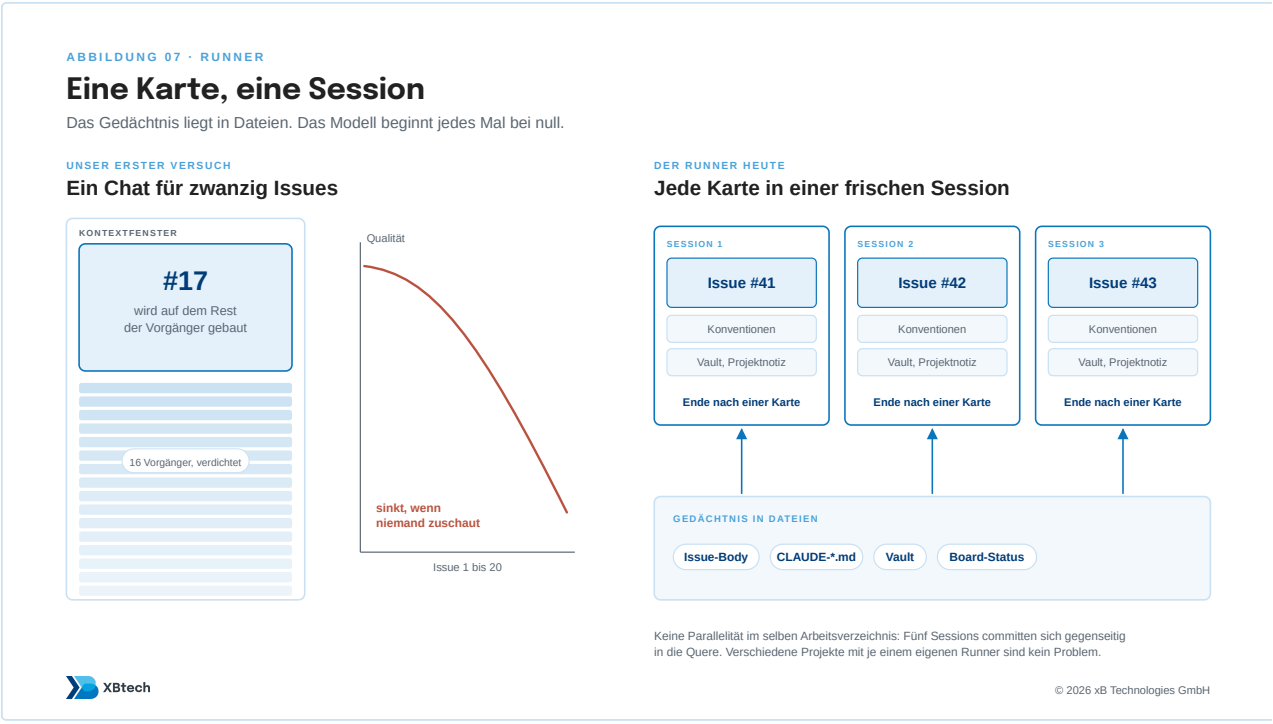

Abbildung 7: Eine Karte, eine Session. Das Gedächtnis liegt in Dateien, das Modell beginnt jedes Mal bei null.

Pro Karte läuft dasselbe ab. Karte nach In progress, Issue lesen, Tests zuerst, implementieren bis grün, lokal committen, Karte nach In review. Den Erfolg liest der Runner am Board ab und nicht an dem, was die Session über sich selbst erzählt.

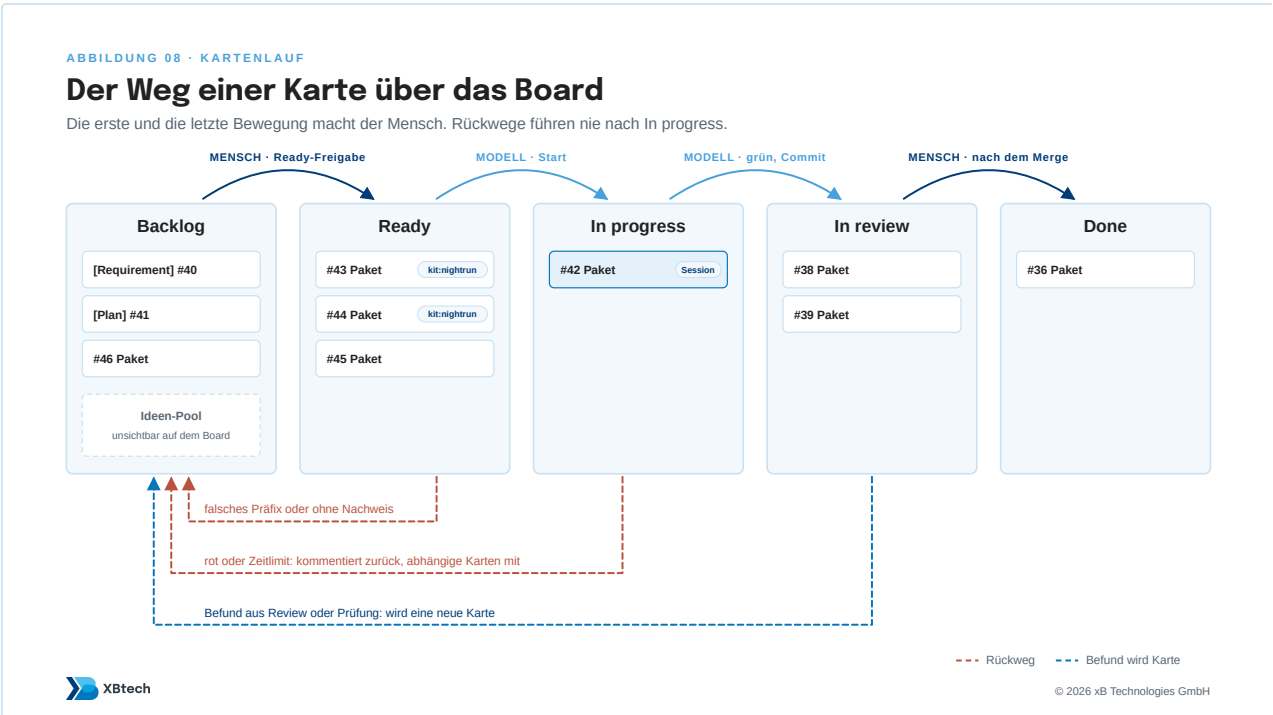

Abbildung 8: Der Kartenlauf. Die erste und die letzte Bewegung macht der Mensch, die beiden in der Mitte das Modell.

Geht etwas schief, unterscheidet der Runner genau. Ist die Arbeit gescheitert oder das Zeitlimit gerissen, wandert die Karte kommentiert zurück ins Backlog, und der Lauf geht mit der nächsten weiter. Ist die Umgebung kaputt, etwa weil eine Anmeldung abgelaufen ist, stoppt er hart und lässt die Karte in Ready liegen. Mit ihr ist ja nichts

falsch. Hinterlässt eine Session einen unsauberen Arbeitsbaum, fährt er die Checks selbst, und sind sie grün, bekommt eine Rettungssession zehn Minuten zum Committen. Pushen tut der Runner nie, mit keinem Flag.

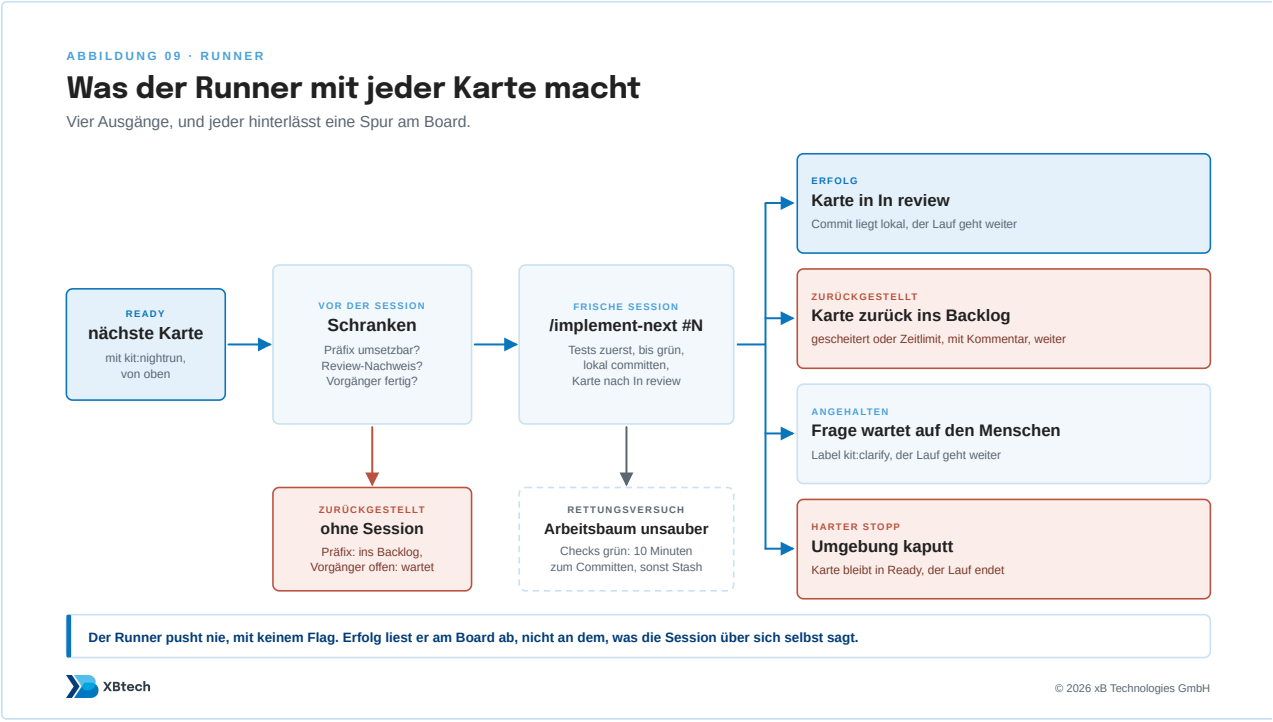

Abbildung 9: Was der Runner mit jeder Karte macht. Vier Ausgänge, und jeder hinterlässt eine Spur am Board.

Manchmal scheitert die erste Karte, und zehn andere, die auf ihr aufbauen, landen mit im Backlog. Auf den ersten Blick sieht das aus wie ein verlorener Lauf. Wir sehen das anders: Zehn Pakete auf einem gebrochenen Fundament hätten uns den ganzen nächsten Tag gekostet.

Wer will, überlässt dem Runner auch die Schritte vor den Paketen. Das Label `kit:night` an einem gegroomten und geprüften Fachkonzept startet eine Kette. Sie schreibt den Plan, prüft ihn, schneidet ihn in Pakete und hält die Pakete gegen die Anforderung. Mit `kit:carry-through` geben wir das Ready vorab, dann setzt die Kette die Pakete auch gleich um. Bleibt eine Frage offen, hält sie an und wartet auf uns. Am Ende hängt ein Bericht an der Karte.



Abbildung 10: Die Kette. Ein Label, und am Ende liegen geprüfter Plan, Pakete und Bericht am Board.

06. Leitplanken, die scheitern können

Ein Sprachmodell kann kein Gate sein. Derselbe Code, zweimal geprüft, ergibt nicht zwangsläufig denselben Befund, das Modell ist gegenüber der eigenen Lösung unkritisch, und es lässt mit sich reden. Ein fehlschlagender Build lässt sich nicht überreden. Deshalb sitzen die harten Prüfungen bei uns in Werkzeugen, die den Build scheitern lassen.

Die Schichten liegen hintereinander. Zuerst Test-First: Der Test bindet das Modell an etwas, das nicht aus dem Modell selbst kommt. Dann das Commit-Gate, das jeden Commit abweist, dem kein grüner Prüflauf auf demselben Stand vorausging. Im Build stecken 100 Prozent Coverage oder ein begründeter Ausschluss, Mutationstests, ArchUnit für die Architekturregeln und statische Analyse ohne offene Findings. Im CI laufen SonarQube, gitleaks und Semgrep noch einmal, und diese Instanz überredet niemand.

Das Code-Review durch ein Modell aus einer anderen Familie ersetzt keine dieser Schichten. Es beantwortet Urteilsfragen, die kein Werkzeug stellt: Löst der Code das Problem aus dem Issue? Fehlt ein Randfall? Führt ein Name in die Irre? Und am Ende entscheidet ein Mensch.

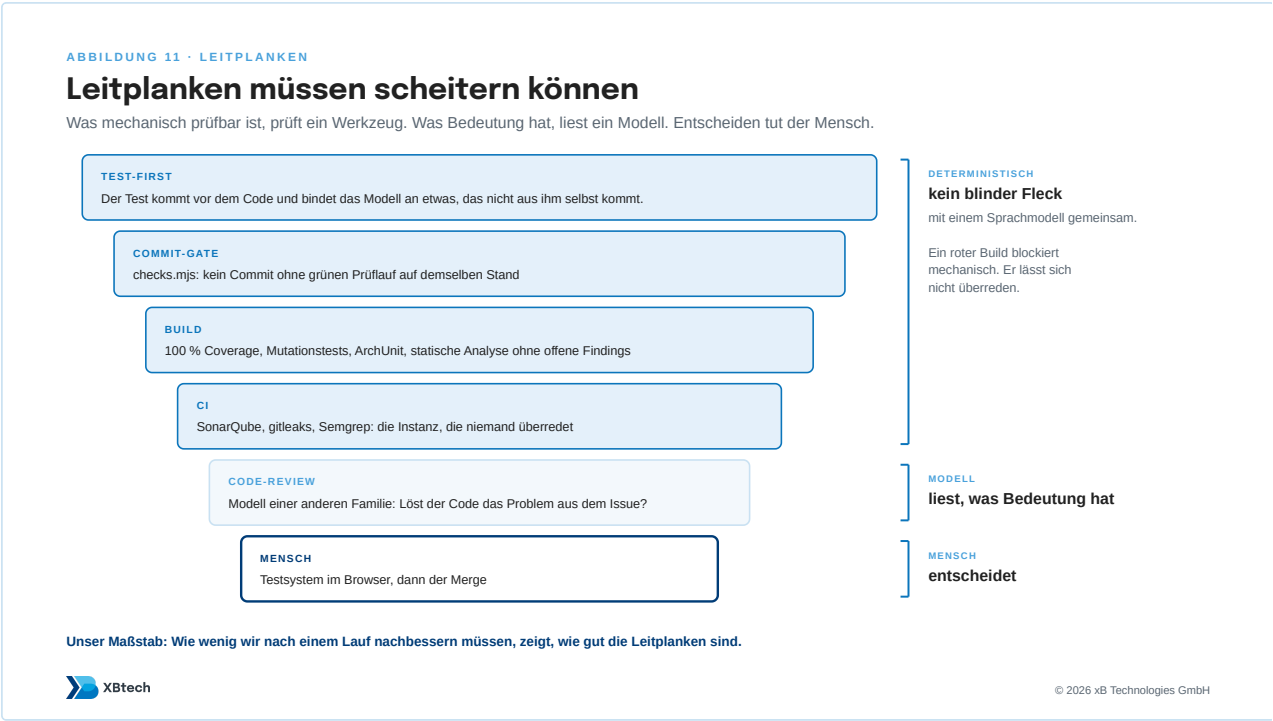

Abbildung 11: Leitplanken müssen scheitern können, nicht argumentieren. Deterministische Werkzeuge teilen mit keinem Modell einen blinden Fleck.

Unser Maßstab für die Leitplanken ist einfach. Wie wenig wir nach einem Lauf nachbessern müssen, zeigt, wie gut sie sind.

07. Befunde fließen zurück ins Backlog

Am meisten verändert hat unsere Arbeit eine Rückkopplung: Jeder Befund wird eine Karte. Meldet SonarQube nach einem Lauf neunzehn Punkte, liegen danach neunzehn Karten im Backlog. Ein Dashboard reicht uns dafür nicht, weil wir über ein Dashboard nicht stolpern. Als Karte konkurriert technische Schuld ganz normal mit den Features, und manchmal gewinnt sie.

Die Befunde kommen aus mehreren Quellen. Die Sicherheitsprüfung vor jedem push main legt für jeden Fund der Stufe KRITISCH oder WICHTIG eine Karte an. Ein roter Integrationslauf erzeugt je Prüfung eine Reparaturkarte mit dem Fund im Wortlaut und dem Paket, das ihn verursacht hat. Ein Review-Befund, der sich als Designfrage entpuppt, wird ein neues Paket oder ein neues Fachkonzept. Und was im wöchentlichen Stakeholder-Review auffällt, landet ebenfalls als Karte im Backlog.

Dort entscheiden wir. Ist es ein Fehlalarm, weisen wir den Befund ab. Ist es echte Arbeit, schneiden wir sie wie jede andere Karte und geben sie frei. Es gibt Läufe, in denen der Runner ausschließlich Sonar-Karten abarbeitet. Die sind unspektakulär und gehören zu den produktivsten, die wir haben.

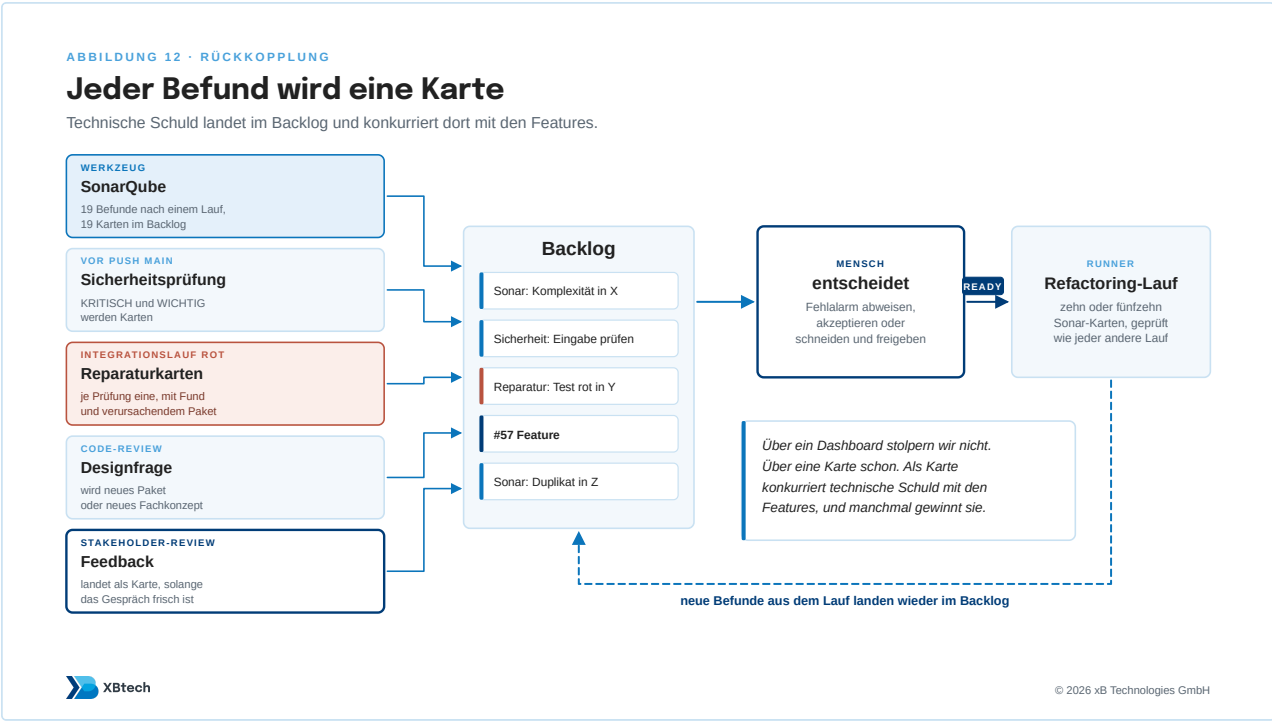

Abbildung 12: Technische Schuld wird zur Karte. Sie konkurriert mit den Features um das nächste Ready.

In unserer Software Factory schließt sich der Kreis ohne unser Zutun. Jede fertig committete Karte löst eine Prüfung aus, im Umfang der Push-Stufe samt Sicherheitsprüfung und Code-Review. Ist sie rot, entsteht je Ursache eine Reparaturkarte mit dem Label `kit:repair`. Diese Karten zieht die Prüfung selbst nach Ready, setzt sie in einem eigenen Worktree um und prüft nach, höchstens dreimal je Ursache. Danach nennt eine [Human]-Karte die offenen Funde, und wir sind dran.

Ohne unsere Freigabe nach Ready gehen nur zwei Sorten Karten: die Reparaturkarten und, am Ende eines Laufs, den wir selbst gestartet haben, die WICHTIG-Funde aus dem Code-Review. Ein neuer Fund aus deren Umsetzung wartet schon wieder auf uns, genau wie jede andere Karte.

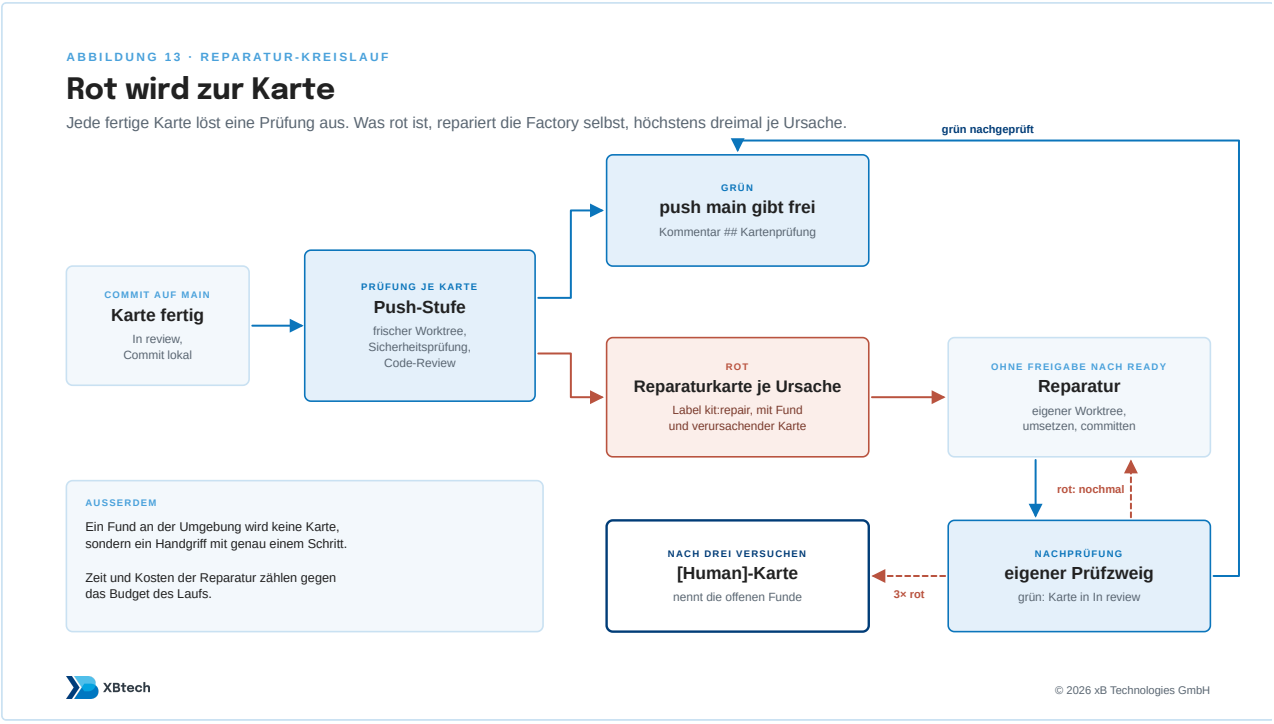

Abbildung 13: Der Reparatur-Kreislauf. Rot wird zur Karte, die Karte wird repariert und nachgeprüft, nach drei Versuchen übernimmt der Mensch.

08. Veröffentlichen: push main und merge production

Bevor etwas veröffentlicht wird, liest ein zweites Modell den Code. Was der Runner allein gebaut hat, sehen wir im Code-Review zum ersten Mal richtig, und wir nehmen die Befunde entsprechend ernst.

push main tippen wir selbst, und jeder Batch braucht seine eigene Freigabe. Danach läuft der Schritt allein, und zwar im Kreislauf, bis der Stand grün ist: prüfen, aus jedem Rot eine Reparaturkarte machen, reparieren, nachprüfen. Veröffentlicht wird nur ein Stand, der grün geprüft ist. Dann zieht das Testsystem nach.

Auf dem Testsystem prüft ein Mensch im Browser: Golden Path, kritische Randfälle, keine sichtbaren Regressionen. Dafür gibt es bewusst keinen Skill. Drei Fehler, die uns Produktionszeit gekostet haben, waren erst dort sichtbar, alle drei bei grünen Tests. Erst danach kommt merge production. Der Skill legt den Pull Request an, mergen tut, wer geprüft hat. Den Git-Tag gibt er als fertige Zeile aus, abschicken müssen wir sie selbst.

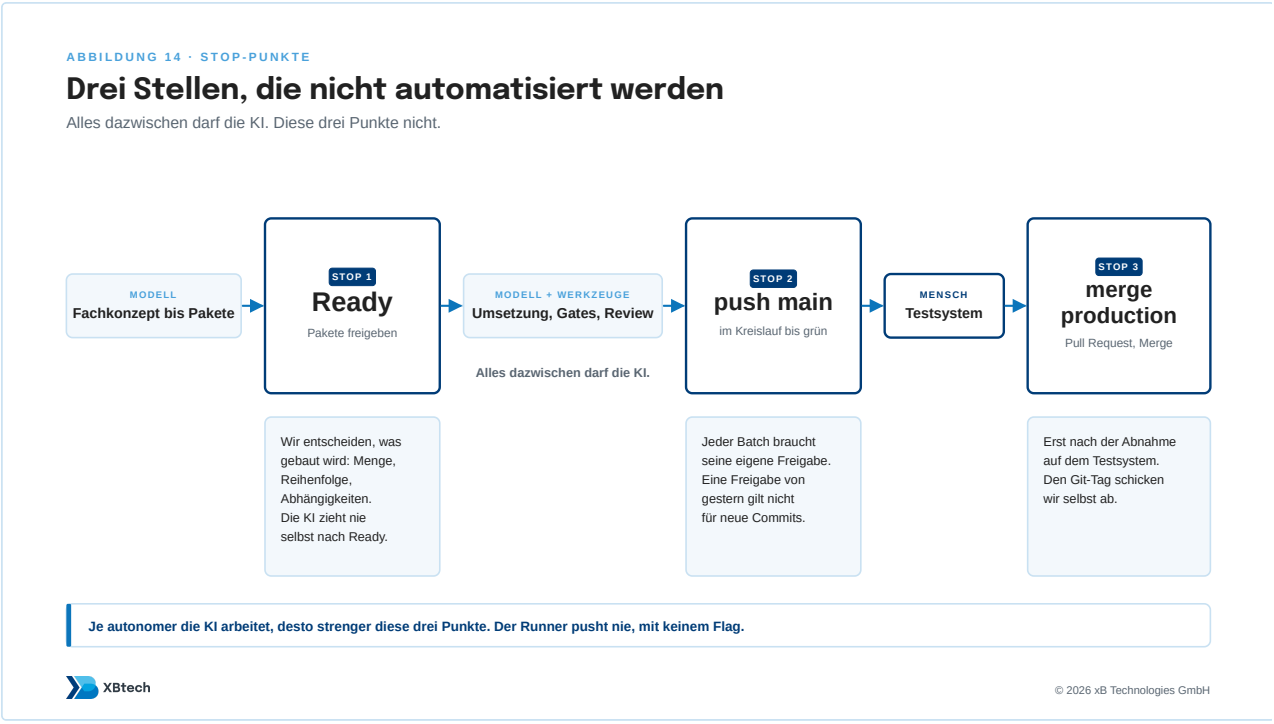

Abbildung 14: Alles dazwischen darf die KI. Diese drei Punkte nicht.

09. Unser Tag im Moment

Unser Lieblingsablauf sieht im Moment so aus. Morgens klären wir mit dem Fachbereich, was die nächsten Funktionen sein sollen: Fachkonzept, Prüfung, Grooming, Plan, Pakete, Ready. Ungefähr von 11 bis 16 Uhr läuft dann der Runner und setzt alles um, ohne dass wir danebensitzen.

Abends sehen wir nach, ob Karten übrig geblieben sind, Reparaturen oder Funde aus dem Code-Review. Sind die erledigt, tippen wir push main, und der läuft die ganze Nacht im Kreislauf, bis er grün ist. Morgens steht alles auf dem Testsystem. Der Fachbereich oder der PO sieht nach, ob alles passt, wir mergen nach production, und dann beginnt die nächste Runde.

Dass der Runner auch nachts bauen kann, ist dabei eine Binse. Er läuft allein, und wann, ist egal.

Der Tagestakt ist ein Beispiel. Den Takt stimmen wir mit dem Fachbereich ab, vom einzelnen Requirement bis zur Woche. In seinen eigenen Projekten geht Manfred Wolff nach jedem Requirement aufs Testsystem, vier- bis fünfmal am Tag.

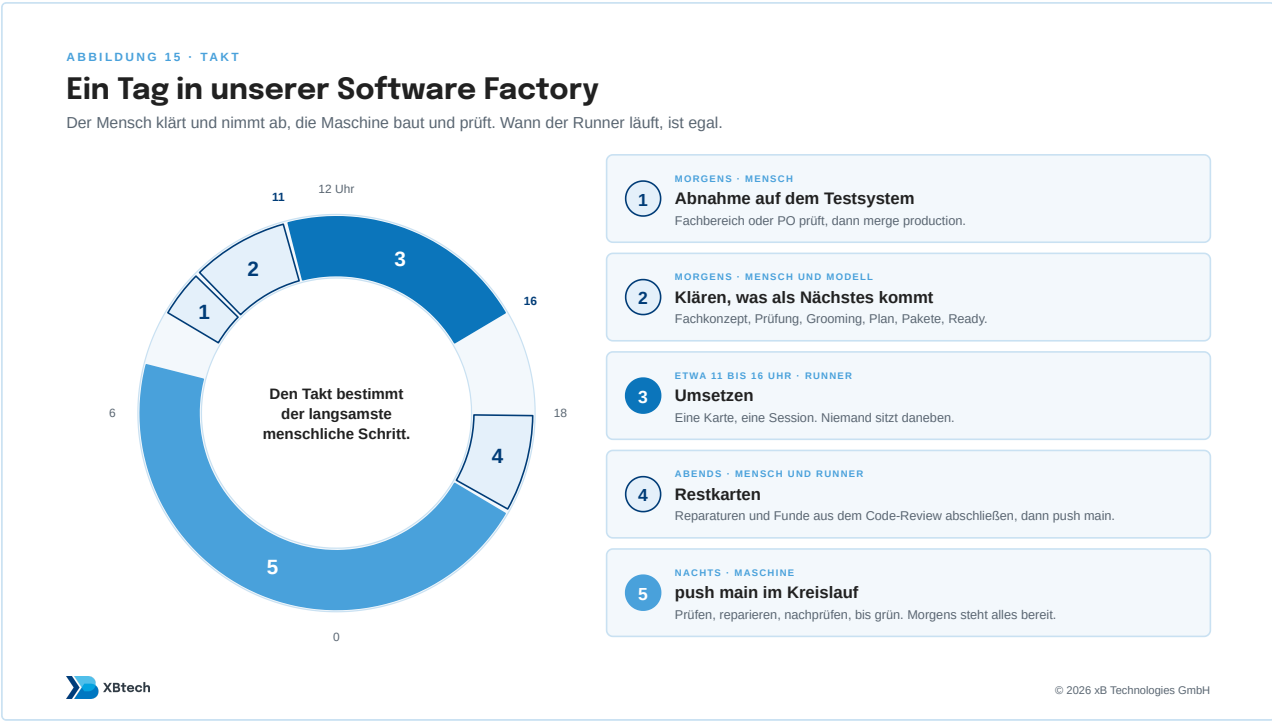

Abbildung 15: Ein Tag in unserer Software Factory. Der Mensch klärt und nimmt ab, die Maschine baut und prüft.

10. Was unsere Software Factory nicht kann

Eines kann unsere Software Factory nicht: aus einer falschen Anforderung eine richtige machen. Ein Paket, das präzise das Falsche verlangt, ist nach drei Modellen und einem Lauf präzise falsch gebaut. Deshalb stehen die Stop-Punkte da, wo sie stehen, und in keinem Umbau des Kits haben sie sich verändert. Je autonomer die KI arbeitet, desto strenger müssen sie sein.

Der Mensch bleibt am Steuer. Auch wenn er nicht danebensitzt.

Mehr zum Prozess steht im Buch „Software mit KI entwickeln“ von Manfred Wolff und in der Dokumentation unter docs.mwolff.org.