Spec-Driven Development (SDD)

KI-assistierte Softwareentwicklung

Handreichung für Studium und Praxis

Worum es geht.

Diese Handreichung erklärt, wie sich Softwareentwicklung durch KI-Coding-Agenten verändert. Sie grenzt das fehleranfällige „Vibe Coding“ vom strukturierten Spec-Driven Development ab, ordnet SDD in bekannte Verfahren des Software-Engineerings ein, benennt seine Prinzipien und Grenzen und liefert ein durchgängiges Beispiel, einen praxistauglichen Workflow sowie einen Meta-Prompt zum schnellen Erstellen von Spezifikationen. Der zweite Teil ist ein Tutorial, das den Weg von der leeren Projektmappe bis zum laufenden Prototyp Schritt für Schritt zeigt.

Inhalt

Teil I — Die Methode

1. Was ist SDD — und woher es kommt

Spec-Driven Development stellt eine maschinenlesbare Spezifikation ins Zentrum des Entwicklungsprozesses. Zuerst wird beschrieben, *was* ein System leisten soll und unter welchen Bedingungen; aus dieser Beschreibung leitet ein KI-Agent Plan, Tests und Code ab. Die Spezifikation ist der verbindliche Vertrag — die „Source of Truth“. Code und Tests sind nachgelagerte Artefakte, die sich an ihr messen lassen müssen.

Der entscheidende Effekt: Der menschliche Aufwand verschiebt sich von der Implementierung hin zur präzisen Beschreibung von Anforderungen, Architektur und Akzeptanzkriterien — also dorthin, wo Fehler am billigsten zu beheben sind.

Vertiefung — Einordnung

SDD ist keine Erfindung von 2025, sondern eine Neuverpackung bewährter Ideen des Software-Engineerings:

- **Contract-First-Design** — die Schnittstelle (z. B. als OpenAPI/Swagger-Dokument) steht vor der Implementierung fest.
- **Design by Contract** (Bertrand Meyer, Eiffel) — Vor- und Nachbedingungen als Vertrag zwischen Komponenten.
- **Formale Spezifikation** (Z, VDM) und **Behaviour-Driven Development** (Gherkin/Cucumber) — Verhalten in prüfbarer Form.
- **Architecture Decision Records** — bewusst dokumentierte, nachvollziehbare Entwurfsentscheidungen.

Neu ist allein der Auslöser: Leistungsfähige Coding-Agenten machen es erstmals praktikabel, ausführbaren Code unmittelbar aus einer präzisen Spezifikation zu erzeugen. SDD ist damit weniger ein Bruch als die konsequente Rückkehr zum Requirements Engineering unter neuen Vorzeichen.

2. Das Problem: „Vibe Coding“

Der Begriff „Vibe Coding“ wurde Anfang 2025 geprägt[1]. Sein Kern ist nicht bloß „einfacher Prompt, dann iterieren“, sondern dass man den generierten Code akzeptiert, ohne ihn vollständig zu lesen oder zu verstehen — man gibt sich den „Vibes“ hin und vertraut dem Modell. Für schnelle Prototypen, Wegwerf-Skripte und Exploration ist das ein legitimes, oft sehr produktives Vorgehen.

Bei ernsthaften, langlebigen Systemen treten jedoch typische Nachteile auf: Der Agent trifft willkürliche architektonische Entscheidungen, die Ergebnisse sind schwer reproduzierbar, es fehlt die Nachvollziehbarkeit, und der Code „kompiliert zwar, trifft aber die eigentliche Absicht nicht“. Der klassische Planungs- und Reviewzyklus wird übersprungen, was Wartung und Teamarbeit erschwert.

Wichtig — keine Schwarz-Weiß-Frage

Vibe Coding ist nicht per se schlecht; es ist das richtige Werkzeug für Wegwerf-Code und schnelle Experimente. SDD lohnt sich, sobald Korrektheit, Wartbarkeit, Teamarbeit, Sicherheit oder rechtliche Nachvollziehbarkeit zählen. Die Kunst liegt darin, das passende Verfahren bewusst zu wählen.

[1] Begriff geprägt von Andrej Karpathy, Februar 2025.

3. Die Prinzipien des SDD

Die folgenden acht Prinzipien beschreiben die Denkweise hinter SDD:

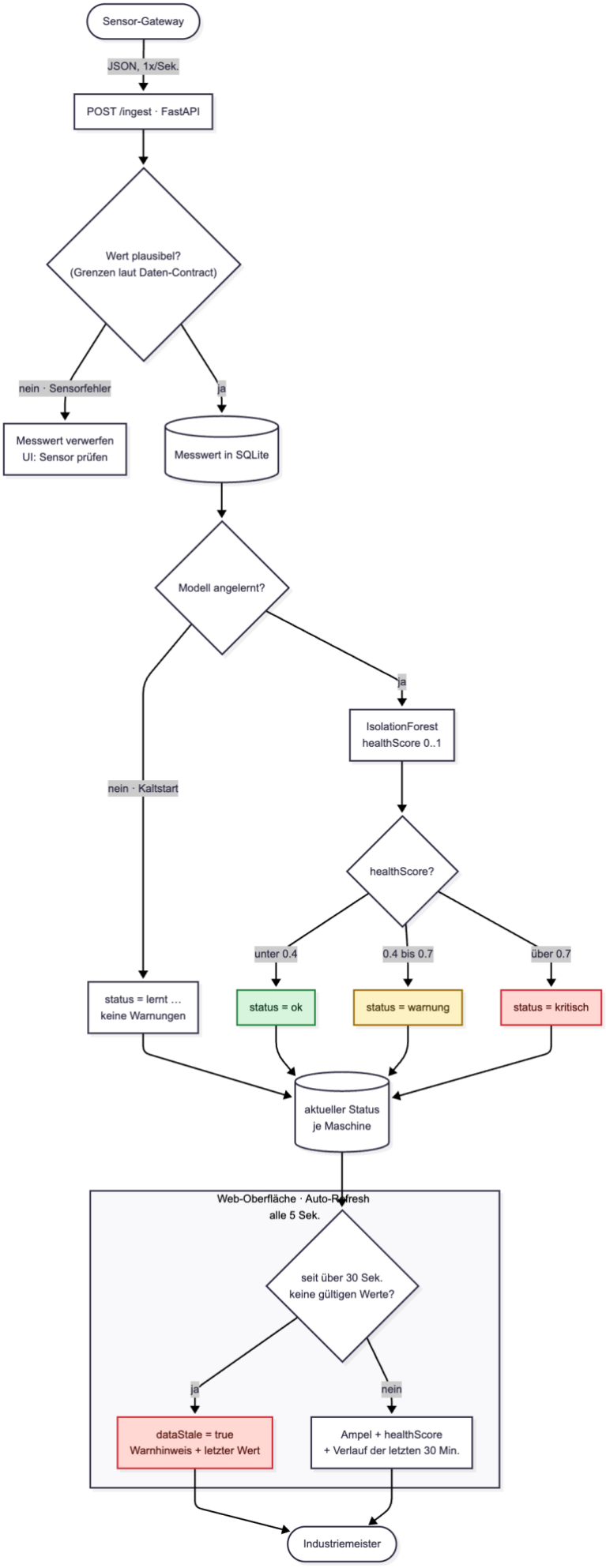
- **Spezifikation als Source of Truth.** Die Spec ist der maßgebliche Vertrag. Bei Abweichungen wird zuerst die Spec geprüft und angepasst, nicht ad hoc der Code.
- **Trennung von „Was“ und „Wie“.** Der Mensch definiert das Was (Geschäftslogik, Anforderungen), der Agent übernimmt das Wie (Syntax, Boilerplate, Implementierungsdetails).
- **Shift-Left der menschlichen Kontrolle.** Das Review findet bei Spezifikation und Architektur statt — bevor Code entsteht, wo Fehler am günstigsten zu korrigieren sind.
- **Leitplanken statt Determinismus.** Enge Vorgaben zu Bibliotheken, Fehlerbehandlung und Namenskonventionen verkleinern den Lösungsraum und reduzieren Halluzinationen. Sie machen die Generierung nicht deterministisch — LLMs bleiben stochastisch —, aber kalkulierbarer.
- **Verifizierbarkeit durch Design.** Eine Spec ist nur so gut, wie sie sich in messbare Akzeptanzkriterien (etwa Unit- und E2E-Tests) übersetzen lässt.
- **Nachvollziehbarkeit (Traceability).** Jedes Code-Artefakt lässt sich auf ein Spec-Element zurückführen. Dies ist der eigentliche Mehrwert gegenüber Vibe Coding und unverzichtbar in regulierten Umfeldern.
- **Lebendes Dokument.** Die Spec wird versioniert, lebt im Repository und wird über Reviews bzw. Pull Requests fortgeschrieben — nicht einmalig geschrieben und beiseitegelegt.
- **Werkzeug-Unabhängigkeit.** SDD ist eine Methodik, kein Produkt. Sie funktioniert mit verschiedenen Agenten und Stacks.

> **Vertiefung — „Source of Truth“ realistisch betrachtet**
>
> Das Ideal lautet: bei Fehlern stets die Spec ändern und den Code neu generieren, nie umgekehrt. In der Praxis ist das eine Zielvorstellung, kein Automatismus. Generierter Code braucht weiterhin Review; eine Spec erfasst selten 100 % des Verhaltens; und das verlustfreie Regenerieren von Code aus der Spec („Round-Tripping“) funktioniert heute nicht zuverlässig. Wer am Code vorbei arbeitet, erzeugt „Spec-Drift“ — Code und Spec laufen auseinander. Behandle das Prinzip daher als Disziplin, die das Team aktiv aufrechterhält.

4. Das Spezifikationsdokument

Eine Spec wird als maschinenlesbarer Vertrag (typischerweise Markdown) in das Projekt gelegt und dem Coding-Agenten als Kontext übergeben. Das folgende Beispiel stammt aus der Arbeitswelt vieler Kursteilnehmer: ein Prototyp für Predictive Maintenance. Eine Maschine wird mit Sensoren nachgerüstet, ein ML-Modell bewertet laufend ihren Zustand, und ein Industriemeister überwacht das Ergebnis über eine einfache Web-Oberfläche.

Das Beispiel ist bewusst nah am Proof of Concept gehalten — denn häufig entsteht zuerst ein Prozess-Prototyp, den später externe Fachleute zum produktiven System ausbauen. Gerade dann ist eine grundsolide Anfangs-Spezifikation Gold wert: Sie ist zugleich Bauplan und Übergabedokument. Beachte besonders die Abschnitte 7 und 8 — sie machen Annahmen und Scope explizit und sagen dem Folgeteam genau, wo es ansetzen muss.



Datenfluss des Prototyps: von der Sensorik über Plausibilitätsprüfung und Anomalie-Score bis zur Ampel-Oberfläche.

Beispiel: spec.md (Predictive-Maintenance-Prototyp)

```
# Spezifikation: Predictive-Maintenance-Prototyp für eine Produktionsmaschine

> Status: Prototyp / Proof of Concept · Source of Truth für dieses Projekt.
> Änderungen am Verhalten werden zuerst hier eingetragen, dann im Code umgesetzt.

## 0. Zweck & Reifegrad
PROTOTYP / Proof of Concept. Ziel ist NICHT der produktive Dauerbetrieb, sondern der
Nachweis, dass sich aus Sensordaten ein brauchbares Frühwarnsignal für den Wartungsbedarf
ableiten lässt. Die skalierte, produktive Umsetzung erfolgt anschließend mit externen
Fachleuten (siehe Abschnitt 8). Diese Spec ist zugleich Bauplan und Übergabedokument.

## 1. Daten-Contract (Sensorik)
- Quelle: Sensor-Gateway sendet Messwerte als JSON per HTTP POST /ingest.
- Abtastrate (Prototyp): 1 Messung pro Sekunde je Sensor.
- Felder je Nachricht:
  - machineId   (string)
  - timestamp   (ISO 8601, UTC)
  - vibration   (number, mm/s)
  - temperature (number, °C)
  - current     (number, A)
- Plausibilitätsgrenzen — Werte außerhalb gelten als SENSORFEHLER, nicht als Maschinenfehler:
  - temperature: -20 bis 150   vibration: 0 bis 50   current: 0 bis 100

## 2. ML-Aufgabe (bewusst einfach gehalten)
- Aufgabentyp: ANOMALIE-ERKENNUNG (unüberwacht). KEINE Ausfallvorhersage in Zeiteinheiten —
  im Prototyp liegen i. d. R. noch keine echten Ausfälle als Trainingslabel vor
  ("Cold-Start-Problem").
- Vorgehen: Das Modell lernt das Normalverhalten aus einer störungsfreien Referenzphase
  und meldet Abweichungen als Anomalie-Score.
- Ausgabe je Maschine, alle 5 Sekunden aktualisiert:
  - healthScore (number, 0..1)
  - status      ("ok" | "warnung" | "kritisch")
    Schwellen (Prototyp, später kalibrierbar): < 0.4 = ok, 0.4-0.7 = warnung, > 0.7 = kritisch
  - dataStale   (boolean) — true, wenn seit mehr als 30 s keine gültigen Werte kamen

## 3. Web-Oberfläche (für den Industriemeister)
- Eine Seite, Auto-Refresh alle 5 Sekunden.
- Zeigt je Maschine: Ampel (grün/gelb/rot), healthScore, Zeitpunkt der letzten Messung
  sowie den Verlauf der letzten 30 Minuten als einfaches Liniendiagramm.
- Bei Status "kritisch" oder dataStale: deutlich sichtbarer Warnhinweis.
- Verständlich ohne Schulung — der Meister muss den Zustand sofort deuten können.

## 4. Verhalten in Sonderfällen
- Datenlücke / Sensorausfall: Status NICHT auf "ok" setzen; dataStale = true und
  letzten bekannten Wert mit Zeitstempel anzeigen.
- Sensorfehler (Wert außerhalb der Plausibilitätsgrenzen): Messwert verwerfen, im UI als
  "Sensor prüfen" markieren, nicht in den healthScore einrechnen.
- Kaltstart (Modell noch nicht angelernt): Status "lernt ..." anzeigen, noch keine
  Warnungen erzeugen.

## 5. Technische Constraints (prototyp-tauglich)
- Sprache: Python.
- ML: scikit-learn, z. B. IsolationForest — KEIN Deep Learning im Prototyp.
- Backend: FastAPI. Datenhaltung: SQLite (eine Datei, kein DB-Server).
- Frontend: eine einfache HTML-Seite, die das Backend per fetch abfragt.
- Alles lokal auf einem Rechner lauffähig, keine Cloud-Abhängigkeit.

## 6. Akzeptanzkriterien (Prototyp gilt als erfolgreich, wenn ...)
1. Bei eingespielten Normaldaten bleibt der Status stabil auf "ok".
2. Eine künstlich erzeugte Auffälligkeit (z. B. steigende Vibration) hebt den Status
   innerhalb von 15 s auf "warnung"/"kritisch".
3. Bei abgeschaltetem Sensor zeigt das UI binnen 30 s dataStale an.
4. Der Industriemeister deutet den angezeigten Zustand ohne Erklärung richtig.
```

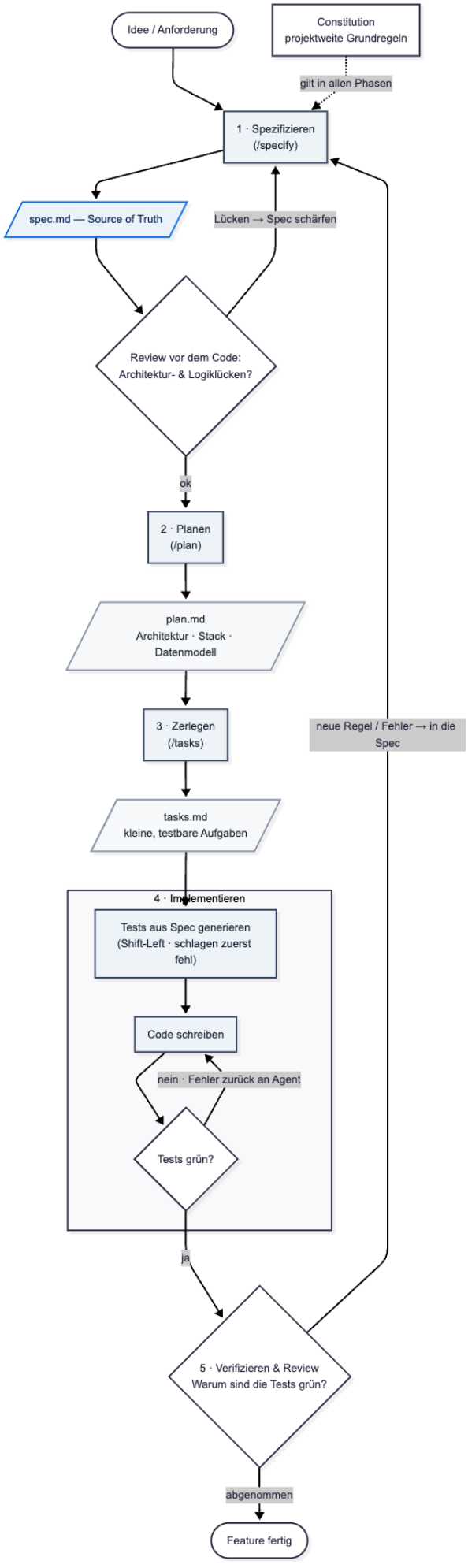
```
## 7. Annahmen (explizit, damit sie überprüfbar bleiben)
- Genau eine Maschine; Mehrmaschinen-Betrieb erst in der Produktivversion.
- Läuft im geschützten internen Netz → keine Auth/Verschlüsselung im PoC.
- Eine ausreichend lange störungsfreie Referenzphase zum Anlernen existiert.
- Datenmengen sind klein genug für SQLite und einen einzelnen Prozess.

## 8. Bewusst NICHT im Prototyp – Übergabe an Fachleute
- Skalierung auf viele Maschinen, Hochverfügbarkeit, Zeitreihen-Datenbank.
- Echte Restlebensdauer-Vorhersage auf Basis gelabelter Ausfälle.
- Authentifizierung, Rollen, Audit, Alarmierung per E-Mail/SMS.
- Modell-Monitoring, Drift-Erkennung, regelmäßiges Re-Training.
- OT-/Sicherheitsanforderungen (Trennung vom Produktionsnetz, einschlägige Normen).
```

Vertiefung — was eine gute KI-/ML-Prototyp-Spec ausmacht

Bei einem ML-Prototyp entscheiden andere Dinge über die Qualität als bei klassischer Software. Die wichtigsten:

- **ML-Aufgabe präzise benennen:** Anomalie-Erkennung ist nicht dasselbe wie Ausfallvorhersage. Wer das offenlässt, bekommt vom Agenten — oder später vom externen Team — womöglich etwas ganz anderes als gemeint.
- **Cold-Start ehrlich behandeln:** Predictive Maintenance bräuchte eigentlich echte Ausfalldaten zum Lernen, die im Prototyp fast nie vorliegen. Die Spec macht aus dieser Not eine bewusste Entscheidung: unüberwachte Anomalie-Erkennung auf Basis des Normalverhaltens.
- **Daten-Contract ist das Fundament:** Format, Abtastrate, Einheiten und Plausibilitätsgrenzen. Ein ML-System ist nur so gut wie die Klarheit über seine Eingangsdaten — „garbage in, garbage out“.
- **Sensorfehler ≠ Maschinenfehler:** Diese Unterscheidung entscheidet, ob das Frühwarnsystem vertrauenswürdig ist. Ein defekter Sensor darf keinen Maschinenalarm auslösen — und umgekehrt.
- **Verständlichkeit für den Nutzer:** Der Industriemeister ist kein Data Scientist. Eine Ampel, die er ohne Schulung deuten kann, ist mehr wert als ein präziser, aber unverständlicher Score.
- **Annahmen und Scope explizit machen:** Gerade weil später Fachleute übernehmen, ist die Spec die Übergabestelle. Die Abschnitte „Annahmen“ und „Bewusst NICHT im Prototyp“ sind kein Beiwerk, sondern der eigentliche Wert — sie verhindern, dass ein Proof of Concept versehentlich für ein fertiges Produkt gehalten wird.



5. Der Workflow in der Praxis

Der heute etablierte Ablauf gliedert sich in fünf Phasen, die jeweils ein eigenes, überprüfbares Artefakt erzeugen, das die nächste Phase speist.

Über allen Phasen liegt die **Constitution** — projektweite Grundregeln. Sie liegt als `constitution.md` im Projekt und gilt für jeden Durchlauf. Zwei Eigenschaften unterscheiden sie von allen anderen Artefakten:

Sie ist aus nichts abgeleitet, und sie wird im laufenden Auftrag nicht geändert. Änderungswünsche an ihr sind ein eigener Vorgang, kein Nebenprodukt der Implementierung.

Toolkits wie GitHub Spec Kit bilden den Ablauf über Befehle wie /specify, /plan und /tasks ab; diese Befehle stehen stellvertretend für die jeweilige Phase – die Methode selbst bleibt werkzeugunabhängig.

- **Spezifizieren.** Die Spec schreiben und den Agenten zunächst nur auf Architektur- und Logiklücken prüfen lassen — noch ohne Code.

- **Planen.** Einen technischen Plan ableiten: Architektur, Datenmodell, Schnittstellen, gewählte Bibliotheken und Begründungen.

- **Zerlegen.** Den Plan in kleine, einzeln testbare Aufgaben aufteilen.

- **Implementieren.** Zuerst Tests aus der Spec generieren (Shift-Left; sie schlagen anfangs fehl), dann Code schreiben, bis die Tests grün sind. Fehlermeldungen werden dem Agenten zur Selbstkorrektur zurückgegeben.

Verifizieren & Review. Prüfen, *warum* die Tests grün sind, nicht nur *dass* sie es sind. Neue Regeln wandern in die Spec — niemals als formloser Zuruf an den Code.

Im Diagramm wird der Zusammenhang sichtbar: Aus der Idee entsteht `spec.md` als Source of Truth. Vor der ersten Zeile Code steht ein menschliches Review (Shift-Left); gefundene Lücken fließen zurück in die Spec, nicht in eine spätere Phase. Erst nach Freigabe entstehen `plan.md` und `tasks.md`, dann die Tests und der Code.

Der SDD-Ablauf als Kette aus fünf Phasen mit ihren Artefakten und Rückkopplungen.

Entscheidend sind die **drei Rückkopplungen:** erstens das Review vor dem Code, zweitens die Test-Korrektur-Schleife innerhalb der Implementierung (die nur den Code betrifft, nicht die Anforderungen) und drittens die übergeordnete Schleife, die jede neue Regel und jeden gefundenen Fehler über die Spezifikation führt. Die Pfeile führen deshalb zurück zur Phase „Spezifizieren“, nie direkt zum Code. Erst wenn das Feature abgenommen ist, endet der Durchlauf.

> **Achtung — Grenzen der autonomen Fehlerbehebung**
> Ein Agent kann Tests auch „grün“ bekommen, indem er Assertions abschwächt oder Sonderfälle umgeht, statt das Problem zu lösen. Das Review muss daher prüfen, ob die Tests das richtige Verhalten erzwingen — nicht nur, ob sie bestehen. Begrenze zudem die Zahl der Selbstkorrektur-Schleifen, um Endlos-Iterationen zu vermeiden.

6. Werkzeuglandschaft

Den Begriff SDD und einen konkreten Prozess hat vor allem **GitHub Spec Kit** geprägt[2]: ein quelloffenes, bewusst werkzeug-unabhängiges Toolkit mit einer CLI (`specify`), Vorlagen und den Phasen Spec → Plan → Tasks → Implement. Eine projektweite „Constitution“ hält übergreifende Grundregeln fest. Es arbeitet mit zahlreichen Agenten zusammen.

Coding-Agenten, mit denen SDD praktiziert wird, sind unter anderem Kilo Code, Claude Code, GitHub Copilot, Gemini CLI usw. Daneben existieren weitere spec-orientierte Umgebungen. Entscheidend ist nicht das Produkt, sondern die Disziplin: Spezifikation zuerst, Code als Ableitung.

7. Grenzen — und wann SDD sich nicht lohnt

SDD ist kein Allheilmittel. Eine ehrliche Handreichung benennt die Kehrseiten, damit das Verfahren bewusst eingesetzt wird:

- **Overhead:** Eine gute Spec zu schreiben kostet Zeit. Bei Wegwerf-Code und Prototypen ist das unwirtschaftlich.
- **Fehlbare Spec:** Eine Spezifikation kann genauso unvollständig oder falsch sein wie Code. SDD verlagert das Denken, es ersetzt es nicht.
- **Restrisiko Halluzination:** Auch innerhalb enger Constraints kann ein Modell Fehler produzieren.
- **Spec-Drift & Round-Tripping:** Code und Spec laufen auseinander, wenn man am Code vorbei arbeitet; das verlustfreie Regenerieren ist unreif.
- **Falsches Sicherheitsgefühl:** Eine formal wirkende Spec verleitet dazu, Review und kritisches Denken zu vernachlässigen.

[2] GitHub Spec Kit, als Open-Source-Toolkit veröffentlicht am 2. September 2025 (MIT-Lizenz).

8. Spezifikationen schnell erstellen: der Meta-Prompt

Statt Spezifikationen mühsam selbst zu schreiben, dreht man den Spieß um und lässt sich von der KI interviewen. Den folgenden Prompt in einen LLM-Chat kopieren und den Platzhalter durch die eigene Idee ersetzen.

```
Du agierst ab sofort als sehr erfahrener Senior Software Architect und Product
Manager. Mein Ziel ist es, nach den Prinzipien des Spec-Driven Development (SDD)
zu arbeiten. Am Ende sollst du mir eine wasserdichte, maschinenlesbare Markdown-
Spezifikation (spec.md) für ein Feature/Projekt schreiben, die ich direkt an einen
KI-Coding-Agenten übergeben kann.

Hier ist meine noch sehr vage Idee:
[HIER DEINE IDEE EINTRAGEN, z. B. "Ich brauche ein internes Dashboard, in dem
Mitarbeiter Urlaubsanträge einreichen und Vorgesetzte sie genehmigen."]

WICHTIGE REGEL: Schreibe die Spezifikation noch NICHT.
Bevor wir sie erstellen, musst du alle Unklarheiten beseitigen. Stelle mir eine
strukturierte, nummerierte Liste der 10 bis 14 wichtigsten Fragen. Gruppiere sie
in folgende Kategorien:
  1. Geschäftslogik & Core-Feature-Verhalten
  2. Edge Cases (unerwartetes Verhalten, Fehler, Nebenläufigkeit)
  3. Security, Authentifizierung & Berechtigungen
  4. Technische Constraints (Stack, Datenbank, APIs)
  5. Nicht-funktionale Anforderungen (Performance, Skalierung, Logging, Datenschutz)
  6. Testing & Akzeptanzkriterien

Stelle präzise, technische Fragen. Biete bei komplexen Punkten direkt 2 bis 3
Best-Practice-Optionen (A, B, C) an, aus denen ich wählen kann.

Warte nach deinen Fragen auf meine Antworten. Wenn meine Antworten weitere Lücken
aufwerfen, hake nach. Deine Einschätzung "keine Lücken mehr" ist eine Heuristik,
keine Garantie – bleibe daher kritisch. Erst wenn du hinreichend sicher bist,
werde ich sagen: "Erstelle jetzt die spec.md".
```

So geht es nach dem Prompt weiter

- **Beantworten:** kurz und knapp auf antworten (z. B. „Zu 3: Option B. Zu 4: React und Firebase“).
- **Nachschärfen:** eventuelle Rückfragen der KI klären.
- **Generieren:** den finalen Befehl geben: „Perfekt. Erstelle nun die finale spec.md, strukturiert nach API-Contract, Validierung, Geschäftslogik, Fehlerbehandlung, Constraints und Akzeptanzkriterien.“

9. Glossar

Spec-Driven Development (SDD) — Methodik, die eine maschinenlesbare Spezifikation ins Zentrum stellt; Code wird daraus abgeleitet.

Vibe Coding — Generierten Code akzeptieren und über Folge-Prompts iterieren, ohne ihn vollständig zu durchdringen.

Source of Truth — Die Spezifikation als einzige verbindliche Referenz für das Soll-Verhalten.

Shift-Left — Qualitätssicherung früh im Prozess verankern — hier: Review bei Spec und Architektur statt erst am Code.

Constraint (Leitplanke) — Verbindliche Vorgabe (Stack, Bibliotheken, Konventionen), die den Lösungsraum des Agenten einengt.

Traceability — Nachvollziehbarkeit: jedes Code-Artefakt lässt sich auf ein Spec-Element zurückführen.

Akzeptanzkriterium — Messbare, prüfbare Bedingung, an der sich die Erfüllung einer Anforderung feststellen lässt.

Cold-Start-Problem (ML) — Schwierigkeit, ein Vorhersagemodell zu trainieren, wenn (noch) keine Beispiele für den seltenen Zielfall — etwa Maschinenausfälle — vorliegen.

Anomalie-Erkennung — Unüberwachtes ML-Verfahren, das aus dem Normalverhalten lernt und Abweichungen meldet — geeignet, wenn keine gelabelten Fehlerfälle vorliegen.

Constitution — Projektweite Grundregeln, die über den einzelnen Spezifikationen stehen und festlegen, wie gearbeitet wird: Vorrang bei Widersprüchen, Umgang mit Annahmen, Abbruchbedingungen, Grenzen. Bewusst projektunabhängig formuliert und damit wiederverwendbar; wird im laufenden Auftrag nicht geändert.

Round-Tripping — Idee, Code jederzeit verlustfrei aus der Spec neu zu generieren — aktuell technisch unreif.

Spec-Drift — Auseinanderlaufen von Spezifikation und Code, wenn am Code vorbei gearbeitet wird.

Vorrangregel — Festgelegte Reihenfolge, in der die Projektdateien bei Widersprüchen gelten: constitution.md vor spec.md vor plan.md vor tasks.md. Fehlt sie, entscheidet der Agent selbst, welcher Quelle er folgt – unsichtbar und in jeder Sitzung neu.

Teil II — In der Praxis: Tutorial am Beispiel Predictive Maintenance

Bevor du loslegst: iterativ statt Wasserfall

Vielleicht denkst du beim Blick auf das folgende Tutorial: *„Wie soll ich solche Dokumente erstellen, bevor ich überhaupt etwas programmiert habe?“* Das Tutorial zeigt einen idealen, vollständigen Durchlauf — das Ziel, nicht den Stand, mit dem du beginnst. Niemand schreibt vor dem ersten Tastendruck eine perfekte, fertige Spezifikation; genau das wäre nicht SDD, sondern Wasserfall. In Wirklichkeit wächst die Spec mit dem Projekt: Du startest mit einem Skelett und füllst es Schritt für Schritt. Die saubere Endfassung ist das Ergebnis mehrerer Runden, nicht ihre Voraussetzung.

Der häufigste Irrtum ist, SDD als „alles vorab durchdenken“ zu lesen. SDD ist das Gegenteil — die Spezifikation ist ein lebendes Dokument. Im Prozessdiagramm führt jede Korrektur zurück in die Spec; das ist bereits Iteration.

„Altes Software-Dev“ (Wasserfall)	So ist SDD gemeint (iterativ)
Erst das komplette Dokument, dann Code	Dünne Spec, kurzer Durchlauf, dann nachschärfen
Die Spec ist fertig und unveränderlich	Die Spec ist ein lebendes Dokument
Vollständigkeit vor dem ersten Schritt	Nur so viel wie für den nächsten Schritt nötig
Ein großer Wurf	Viele kleine, vollständige Schnitte

Merksatz — Just enough spec

So viel Spezifikation, wie für den nächsten kleinen Schritt nötig ist, nicht mehr. Ein SDD-Durchlauf (Spec → Plan → Tasks → Implement → Verify) ist kurz und wird wiederholt — im Grunde ein Sprint. Die Iterationseinheit ist die Spec, nicht der Code.

Faustregel für den Einstieg — Stabiles früh, Volatiles spät

Vorab gehört, was billig zu denken und teuer zu ändern ist. Wachsen darf, was man ohnehin erst später genauer weiß.

Früh festlegen (am Beispiel Predictive Maintenance): der ML-Aufgabentyp (Anomalie-Erkennung statt Ausfallvorhersage), der Daten-Contract und ein bis zwei Akzeptanzkriterien.

Später wachsen lassen: konkrete Schwellenwerte, Feinheiten und Sonderfälle.

So fängst du konkret an

1. Mit einem Skelett starten: Zweck, Daten-Contract und eine einzige Akzeptanzbedingung genügen.
2. Den ersten kleinen Schnitt bauen — ein vollständiger, kurzer Durchlauf wie unten im Tutorial.
3. Den laufenden Prototyp „fragen“, was als Nächstes in die Spec muss — und die Spec ergänzen.

Worum es im Tutorial geht.

Es zeigt ganz konkret, wie man ein KI-Projekt nach der SDD-Methode beginnt — von der leeren Projektmappe bis zum laufenden Prototyp. Als Beispiel dient die Predictive-Maintenance-Spezifikation aus Teil I, Abschnitt 4. Jeder Schritt ist für beide Werkzeuge beschrieben: die Kilo Code-Extension in VS Code und Claude Code.

Roter Faden: Spec → Plan → Tasks → Implementieren → Verifizieren. Der Code entsteht zuletzt — und immer aus der Spec.

0. Voraussetzungen & Setup

Du brauchst vorab drei Dinge: VS Code, Python und ein KI-Werkzeug. Lege dann das Projekt an und installiere die Bibliotheken aus der Spec.

Werkzeug einrichten — eine der beiden Varianten

So geht's in Kilo Code (VS Code)

- In VS Code links auf „Extensions“ gehen, nach „Kilo Code“ suchen, installieren. Danach erscheint links das Kilo Code-Symbol — darüber öffnet sich das Seitenpanel.
- Beim ersten Start anmelden bzw. ein Modell hinterlegen.
- Kilo Code kennt vier Funktionen: **Autocomplete, Chat, Edit** und **Agent**. Die Betriebsart wählst du im Auswahlfeld unter dem Eingabefeld; mit Strg/Cmd + . schaltest du durch.

So geht's in Claude Code

Claude Code installieren (aktuelle Schritte siehe Doku-Link am Ende). Empfohlen ist der native Installer; alternativ per npm (benötigt Node.js 18+):

```
# Variante A (empfohlen):
curl -fsSL https://claude.ai/install.sh | bash

# Variante B (npm):
npm install -g @anthropic-ai/claude-code

# danach im Projektordner starten und beim ersten Mal anmelden:
claude
```

Claude Code braucht einen kostenpflichtigen Zugang (Pro/Max) oder einen API-Schlüssel. Die VS-Code-Erweiterung „Claude Code“ zeigt Änderungen als Diff direkt im Editor an.

1. Die Spezifikation erstellen — die Planungsphase

Bevor die Spec ins Projekt wandert, muss sie erst einmal entstehen. Das ist die Phase **Spezifizieren** aus Teil I — der eigentliche Startpunkt jedes SDD-Durchlaufs. Statt die Spezifikation mühsam von Hand zu schreiben, lässt du dich von der KI interviewen: Du gibst eine grobe Idee vor, das Modell fragt so lange nach, bis die Anforderungen klar sind, und schreibt am Ende die `spec.md`. Den fertigen **Meta-Prompt** dafür findest du in Teil I, Abschnitt 8.

Ziel ist nicht die perfekte Spec im ersten Anlauf, sondern eine tragfähige erste Fassung — „just enough spec“ für den ersten Durchlauf. Sie wächst später mit dem Projekt.

> **Für dieses Tutorial**
> Die fertige Beispiel-Spec liegt bereits vor (Teil I, Abschnitt 4). Du hast zwei Möglichkeiten: Entweder du erzeugst mit dem Meta-Prompt eine **eigene** Spec für deine eigene Idee und folgst damit dem ganzen Ablauf — oder du übernimmst die fertige Predictive-Maintenance-Spec und konzentrierst dich auf die übrigen Schritte.

So läuft das Interview in VS Code

> **So geht's in Kilo Code (VS Code)**
> - Neuen Chat öffnen. Im Auswahlfeld den **Chat-** oder **Plan-Modus** wählen — beide denken nur, schreiben keinen Code.
> - Den Meta-Prompt aus Teil I (Abschnitt 8) einfügen und im Platzhalter deine Idee eintragen.
> - Die Rückfragen der KI knapp beantworten; bei Lücken hakt sie nach. Erst wenn alles klar ist, gibst du frei: „Erstelle jetzt die spec.md.“
> - Den erzeugten Text als Datei **spec.md** im Projektordner speichern. Im Chat-Modus kopierst du ihn dorthin; im **Agent-Modus** kann Kilo Code die Datei direkt anlegen.

> **So geht's in Claude Code**
> - Im Projektordner `claude` starten und mit Shift+Tab in den **Plan-Modus** wechseln (sicheres, nur lesendes Arbeiten).
> - Meta-Prompt einfügen, Idee eintragen, Rückfragen beantworten.
> - Mit „Erstelle jetzt die spec.md“ abschließen — Claude Code legt die Datei nach deiner Bestätigung an.

Am Ende dieser Phase hast du eine erste `spec.md`. Im nächsten Schritt verankerst du sie als verbindliche Source of Truth im Projekt.

2. Das Fundament: die Spec ins Projekt legen

Die im letzten Schritt erstellte Spezifikation (oder die fertige Beispiel-Spec aus Teil I, Abschnitt 4) ist die „Source of Truth“. Sie gehört als Datei ins Projekt — nicht in den Chatverlauf, denn der ist flüchtig. Speichere sie im Projektordner als `spec.md`.

Daneben braucht das Werkzeug dauerhafte Projektregeln - die „Constitution“. Sie legt fest, wie gearbeitet wird, und gilt unabhängig davon, was gebaut wird. So musst du dich nicht in jeder Sitzung wiederholen. Beide Werkzeuge nutzen dasselbe Prinzip:

> **So geht's in Claude und/ oder in Kilo-Code (VS Code)**
> Lege eine Datei **constitution.md** im Projekt-Stammverzeichnis an und hänge sie im Chat mit @constitution.md an.

Die Projektdateien und ihre Rollen

Ab hier arbeiten vier Dateien zusammen. Jede beantwortet genau eine Frage – und keine beantwortet die Frage einer anderen mit:

Datei	beantwortet die Frage	wer sie schreibt
`constitution.md`	Wie wird gearbeitet?	du – einmalig, projektübergreifend wiederverwendbar
`spec.md`	Was wird gebaut?	KI-Agent unter deiner Führung – nach Interview mit der KI
`plan.md`	Womit wird es gebaut?	der Agent – abgeleitet aus spec.md
`tasks.md`	In welcher Reihenfolge?	der Agent – abgeleitet aus plan.md

Die Reihenfolge der Zeilen ist kein Zufall. Sie ist zugleich die Vorrangregel: Widersprechen sich zwei Dateien, gilt die weiter oben stehende. Ohne eine solche Regel entscheidet der Agent selbst, welcher Quelle er folgt – und zwar in jeder Sitzung neu und ohne dass du es merkst.

Wichtig ist der Umkehrschluss: **Eine Datei, die es noch nicht gibt, ist kein Mangel.** Wenn du das Interview zur Erstellung der Spezifikation führst, existiert spec.md naturgemäß noch nicht – sie ist ja das Ergebnis dieses Schritts. Fehlende Dateien werden nicht durch den Auftragstext ersetzt; sie werden erarbeitet.

Dateien im Chat ausdrücklich anhängen

Moderne Werkzeuge laden Regeldateien inzwischen von allein. Hänge sie trotzdem ausdrücklich mit dem @-Zeichen an: `@constitution.md`, `@spec.md`. Zwei Gründe sprechen dafür:

- Du siehst, was tatsächlich im Kontext liegt – und merkst, wenn eine Datei fehlt.
- Du gewöhnst dir an, den Kontext bewusst zu wählen, statt ihn dem Werkzeug zu überlassen. Welche Datei wann gebraucht wird, ist eine fachliche Entscheidung, keine Einstellungssache.

Lässt sich ein Satz nicht unverändert in ein beliebiges anderes Projekt kopieren, gehört er nicht in die constitution.md.

Eine ausgereifte Constitution-Datei sieht etwa so aus:

```
# Arbeitsverfassung
> **Verwendung:** Diese Datei bei jedem Auftrag mit `@constitution.md` mitgeben.
> Sie gilt zusammen mit `@spec.md`, `@plan.md` und `@tasks.md`.

---
## 1. Vorrang bei Widerspruch
Widersprechen sich vorliegende Projektdateien, gilt:
`@constitution.md` -> `@spec.md` -> `@plan.md` -> `@tasks.md`.

Nur vorliegende Dateien zählen. Eine noch nicht erstellte Datei ist kein
Mangel, den es zu umgehen gilt - sie ist in der Regel das Ergebnis des
laufenden Arbeitsschritts.

Ausnahme: Weicht `@spec.md` oder `@plan.md` **ausdrücklich und begründet**
von einem Default dieser Verfassung ab, gilt die Abweichung. Stillschweigende
Abweichung ist unzulässig.

Lässt sich ein Widerspruch so nicht auflösen: nicht selbst entscheiden.
Arbeit anhalten, Widerspruch benennen, als ROT melden.

---
## 2. Annahmen und Rückfragen
- **Immer:** Annahmen ausschreiben, bevor Code entsteht. Keine stillen Entscheidungen.
- **In Spezifikation und Planung:** Bei Mehrdeutigkeit nachfragen. Sind
  mehrere Lesarten plausibel, alle nennen - nicht eine auswählen.
- **In der Implementierung:** Keine Rückfragen zu Punkten, die bereits
  geklärt sind. Rückfrage nur bei Widerspruch oder echter Lücke.
- Gibt es einen einfacheren Weg als den geplanten: sagen, nicht heimlich umsetzen.

---
## 3. Einfachheit ist der Default
- Nur umsetzen, was in `@spec.md` steht. Keine zusätzlichen Features, Optionen.
- Keine Abstraktion für Code mit genau einer Verwendungsstelle.
- Keine Fehlerbehandlung für Fälle, die die Spezifikation ausschließt.

Diese Defaults gelten, solange `@plan.md` nichts anderes vorschreibt.
`@plan.md` schlägt den Default - aber nur schriftlich.

---
## 4. Chirurgische Änderungen
- Jede geänderte Zeile muss auf **genau eine** benannte Aufgabe
  zurückführbar sein. Ist sie das nicht, gehört sie nicht in die Änderung.
- Benachbarten Code nicht „verbessern": keine Umformatierung, keine
  Kommentarpflege, kein Refactoring von Funktionierendem.
- Vorhandenen Stil übernehmen, auch wenn ein anderer besser wäre.
- Code, der **vorher schon** tot war: melden, nicht löschen. Importe und
  Variablen, die **durch diese Änderung** verwaist sind: entfernen.
- Ausnahme: Automatische Formatierer dürfen angrenzende Zeilen verändern.
  Im Bericht erwähnen.

---
## 5. Verifikation vor Fertigmeldung
- Prüfkriterium **vor** der Umsetzung benennen. „Funktioniert" ist kein Kriterium, „Test X läuft
grün" ist eines.
- Fehlerbehebung: erst ein Test, der den Fehler reproduziert, dann dieBehebung.
- Prüfbefehl des Projekts: siehe `@plan.md`.

Jede Task wird mit einer Ampel gemeldet:
| Bedeutung |
| **GRÜN**  | Kriterium erfüllt und nachgewiesen.
```

```
| **GELB**  | Umgesetzt, aber nicht nachgewiesen. Grund nennen.
| **ROT**   | Nicht erfüllt oder blockiert.

Nach drei erfolglosen Korrekturversuchen an derselben Task: abbrechen und ROT melden. Nicht weiter
probieren.

---
## 6. Keine Erfindungen
- Keine Bibliotheksfunktion, kein API-Aufruf, kein Parameter ohne Beleg.
  Existenz und Signatur vor Verwendung prüfen.
- Unsicherheit kennzeichnen statt plausibel klingend raten.
- Keine neue Abhängigkeit ohne Freigabe. Freigegebene Abhängigkeiten
  stehen mit fester Version in `@plan.md`.

---
## 7. Sprache, Daten, Reproduzierbarkeit
- Bezeichner, Kommentare und Docstrings auf Deutsch. Schlüsselwörter und
  fremde APIs bleiben englisch.
- Keine echten Personendaten in Beispiel-, Trainings- oder Testdaten.
  Herkunft verwendeter Datensätze dokumentieren.
- Zufall reproduzierbar machen: Seed setzen und im Code sichtbar lassen.

---
## 8. Grenzen
Ohne ausdrückliche Aufforderung nicht anfassen:
- Lock-Dateien
- Dateien außerhalb des Projektordners
Nicht ohne Aufforderung committen oder pushen.

---
## 9. Änderung dieser Verfassung
Diese Datei wird nicht im laufenden Auftrag geändert. Änderungswünsche
als Vorschlag melden.

Bei einer Änderung: Version erhöhen, Datum setzen, geänderte Abschnitte
in einer Zeile benennen
```

Tipp — von Anfang an versionieren

Initialisiere Git und committe nach jeder Phase. So hast du immer einen Rückfallpunkt, wenn der Agent etwas verschlimmbessert.

```
git init
git add spec.md constitution.md
git commit -m "chore: Spezifikation und Constitution"
```

3. Spec prüfen und Plan erzeugen — noch ohne Code

Die Spec steht — jetzt kommt der kritische Blick *vor* dem Code. Das ist der „Shift-Left"-Moment: Bevor eine Zeile Code entsteht, lässt du das Modell die Spezifikation auf Lücken prüfen. Arbeite dafür in einem Modus, der nur liest und nichts verändert.

> **Prompt-Beispiel**
> *Lies @spec.md und @constitution.md. Du bist erfahrener Senior-Entwickler. Prüfe die Spezifikation auf Architektur- und Logiklücken, BEVOR wir Code schreiben. Stelle mir gezielte Rückfragen zu allem, was unklar oder widersprüchlich ist. Schreibe noch keinen Code.*

> **So geht's in Kilo Code (VS Code)**
> - Neuen Chat öffnen. Im Auswahlfeld den Plan-Modus wählen (liest nur, ändert nichts).
> - Dateien als Kontext anhängen mit **@spec.md** und **@constitution.md**. Den Prompt oben einfügen und absenden.

Beantworte die Rückfragen knapp. Tauchen dabei echte Lücken auf, ergänze sie in `spec.md` — nicht im Chat. Danach lässt du den Plan erstellen:

> **Prompt-Beispiel**
> *Erstelle aus @spec.md einen technischen Plan und speichere ihn als plan.md: Projektstruktur (Dateien/Module), Datenfluss, gewählte Bibliotheken und kurze Begründungen. Halte dich an die Constraints in spec.md. Noch kein Implementierungscode.*

Lies `plan.md` durch. Stimmt etwas nicht, korrigierst du die Spec und lässt den Plan neu ableiten — das ist die SDD-Schleife in Aktion.

So sieht die fertige Projektmappe ungefähr aus

```
pdm-prototyp/
  spec.md              # Source of Truth
  plan.md  tasks.md    # abgeleitete Artefakte
  constitution.md      # globale Regeln
  main.py              # FastAPI: /ingest, Status, Web-Seite
  model.py             # Anlernen + Anomalie-Score (IsolationForest)
  simulate_sensor.py   # Sensor-Simulator (--anomalie)
  static/index.html    # Ampel-Oberfläche (Auto-Refresh)
  tests/               # pytest
  pdm.db               # SQLite (wird erzeugt)
```

Die genaue Struktur bestimmt der Agent — wichtig ist nur, dass sie zu `plan.md` passt.

4. In kleine Aufgaben zerlegen

Aus dem Plan wird eine Aufgabenliste. Kleine, einzeln testbare Schritte sind leichter zu prüfen und gehen seltener schief als ein einziger „Mach mal alles“-Auftrag.

> **Prompt-Beispiel**
> *Leite aus @plan.md eine Aufgabenliste ab und speichere sie als tasks.md: kleine, einzeln testbare Schritte in sinnvoller Reihenfolge, jede mit einem Satz Ziel.*

Eine typische Aufgabenliste für diesen Prototyp sieht etwa so aus:

- Datenmodell und Endpunkt `POST /ingest` (FastAPI) anlegen.
- Plausibilitätsprüfung der Messwerte (Sensorfehler erkennen).
- Messwerte in SQLite speichern.
- Modell aus einer Referenzphase anlernen (IsolationForest).
- `healthScore` berechnen und in Status (ok/warnung/kritisch) übersetzen.
- `dataStale`-Logik (seit mehr als 30 s keine Werte).
- Web-Oberfläche mit Ampel und Auto-Refresh.
- Sensor-Simulator zum Testen ohne echte Hardware.
- Tests zu den Akzeptanzkriterien aus `spec.md`.

5. Implementieren — Tests zuerst

Jetzt darf der Agent Dateien anlegen und Befehle ausführen. Die SDD-Reihenfolge lautet: erst die Tests aus den Akzeptanzkriterien, dann der Code, der sie grün macht.

> **So geht's in Kilo Code (VS Code)**
> In den **Agent-Modus** wechseln. Jetzt kann Kilo Code Dateien schreiben und das Terminal nutzen. **Wichtig:** Jede vorgeschlagene Änderung als Diff prüfen und erst dann übernehmen.

> **So geht's in Claude Code**
> Im normalen Modus arbeiten — Claude Code fragt vor Datei- und Terminal-Aktionen nach. Änderungen erscheinen als Diff (VS-Code-Erweiterung). Bestätigen oder ablehnen.

Schritt 1 — Tests, die zunächst fehlschlagen:

> **Prompt-Beispiel**
> *Wir arbeiten @tasks.md Schritt für Schritt ab. Beginne mit den Tests. Schreibe auf Basis der Akzeptanzkriterien in @spec.md (Abschnitt 6) pytest-Tests, die das Verhalten prüfen und zunächst fehlschlagen. Erkläre kurz, was jeder Test prüft. Schreibe noch keinen Implementierungscode.*

Schritt 2 — Code, bis die Tests grün sind:

> **Prompt-Beispiel**
> *Implementiere jetzt den Code, bis alle Tests grün sind. Halte dich strikt an die Constraints in @spec.md. Führe die Tests aus und zeig mir das Ergebnis. Schlagen Tests fehl, korrigiere und wiederhole.*

Der Agent führt nun selbst die Schleife „Test ausführen → Fehler lesen → Code anpassen“ aus. Lass ihn iterieren, aber sieh dir die Diffs an.

Schritt 3 — testen ohne echte Maschine:

> **Prompt-Beispiel**
> *Erzeuge ein Skript simulate_sensor.py, das jede Sekunde plausible Messwerte an /ingest sendet. Mit dem Schalter --anomalie steigt die Vibration langsam an, sodass der Status auf warnung/kritisch wechselt. So kann ich die Akzeptanzkriterien von Hand prüfen.*

6. Verifizieren & weiterentwickeln

Grüne Tests sind notwendig, aber nicht alles. Prüfe, ob sie das richtige Verhalten erzwingen — nicht nur, dass sie bestehen. Gleiche das laufende Verhalten gegen die Akzeptanzkriterien aus `spec.md` ab. Neue Wünsche oder gefundene Fehler kommen ZUERST in spec.md — nie als formloser Zuruf an den Code. Erst danach lässt du den Agenten anpassen. Sonst entsteht „Spec-Drift“: Code und Spezifikation laufen auseinander.

7. Spickzettel

Aufgabe	Kilo-Code (VS Code)
Datei als Kontext geben	@dateiname im Chat
Nur lesen / planen (sicher)	Plan-Modus wählen
Dateien anlegen, Befehle ausführen	Agent-Modus
Projektregeln dauerhaft	constitution.md + @-Verweis
Änderungen prüfen	Diff ansehen, dann „Accept“

8. Häufige Stolpersteine

- **Blind „Accept“ klicken.** Der Agent ist nicht unfehlbar. Diffs immer lesen, bevor du sie übernimmst.
- **Grün ≠ richtig.** Tests können bestehen, weil sie abgeschwächt wurden. Gegen die Akzeptanzkriterien der Spec gegenprüfen.
- **Kontext überladen.** Nur die relevanten Dateien anhängen; bei Themenwechsel eine neue Sitzung starten. Das spart Tokens und verbessert die Antworten.
- **Sensible Daten in die Cloud.** Keine echten Betriebs- oder Maschinendaten an Cloud-Modelle geben, solange das nicht freigegeben ist. Für den Prototyp reichen simulierte Daten.
- **Prototyp mit Produkt verwechseln.** Abschnitt 8 der Spec sagt, was bewusst offenbleibt. Genau dort setzen später die externen Fachleute an.

Quellen & weiterführende Links

- Claude Code — Dokumentation: https://docs.claude.com/en/docs/claude-code/overview
- GitHub Spec Kit (optionales SDD-Toolkit mit /specify, /plan, /tasks): https://github.com/github/spec-kit
- Constitution-Entwurf: https://github.com/multica-ai/andrej-karpathy-skills/tree/main