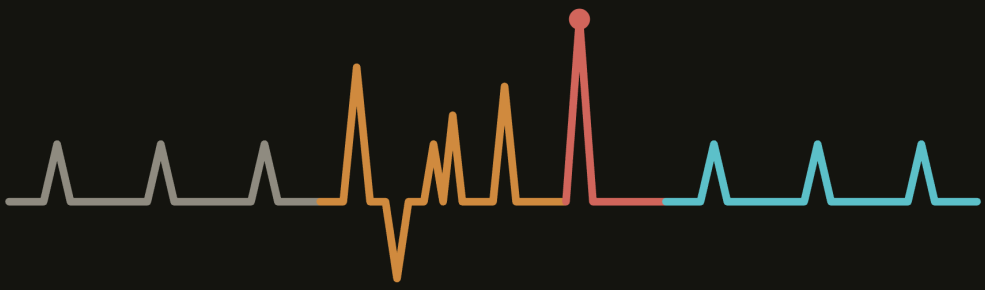
SOFTWAREARCHITEKTUR • CODING-AGENTEN

Der Agent tickt anders



Was Claude Shannon Softwarearchitekten über Coding-Agenten lehrt

INGO EICHHORST

LESEPROBE

Der Agent tickt anders

Leseprobe: Was Claude Shannon Softwarearchitekten über Coding-Agenten lehrt

Ingo Eichhorst

Inhalt

Über diese Leseprobe

Diese Leseprobe enthält die ersten drei Kapitel von „Der Agent tickt anders“ und dazu das vollständige Glossar. Das reicht, um das Denkmodell des Buches kennenzulernen und in der eigenen Arbeit auszuprobieren.

Kapitel 1, Einleitung, beschreibt das Problem: Agentischer Code ist kein Spaghetti-Code mehr, sondern Halluzinations-Lasagne. Sauber geschichtet, appetitlich angerichtet, mit einer Schicht, die wie Bolognese aussieht, aber keine ist. Das neue Problem ist nicht sichtbares Chaos, sondern trügerische Ordnung.

Kapitel 2, Das Shannon-Prinzip, legt die Begriffe frei, mit denen sich diese Ordnung prüfen lässt: Entropie, verrauschter Kanal, Kanalkapazität, Fehlerkorrektur. Claude Shannon hat 1948 gezeigt, wie sich Nachrichten trotz Rauschen zuverlässig übertragen lassen. Sein Modell trägt erstaunlich weit, wenn man einen Coding-Agenten als Kanal liest.

Kapitel 3, Quellencodierung am Sender, setzt das Modell zum ersten Mal praktisch ein. Es geht um die Frage, wie eine vage Absicht zu einer Spezifikation wird, die ein Agent nicht mehr frei interpretieren muss, und darum, welche Bestandteile einer Anweisung messbar etwas bewirken.

Nicht enthalten sind die fünf Kapitel, in denen aus dem Modell ein Arbeitsverfahren wird: der verrauschte Kanal und seine Kapazitätsgrenze, der prüfende Empfänger mit seinen Gates, die Regelkreise und Autonomiestufen, die Übertragung der klassischen Praktiken sowie der Ausblick. Ebenfalls nicht enthalten sind die Anhänge mit den Messdaten der Experimente und das Referenzverzeichnis.

Verweise auf spätere Kapitel bleiben in dieser Leseprobe stehen. Sie zeigen ins vollständige Buch: https://leanpub.com/der-agent-tickt-anders

Vorwort

Ein Vorwort von Ralf D. Müller.

[FOLGT]

1 Einleitung

Mein erstes längeres Programm war ein Desaster, und ich habe es geliebt. Es lief auf einem C64, war in BASIC geschrieben und bestand im Wesentlichen aus GOTO-Sprüngen kreuz und quer durch dreihundert Zeilen. Spaghetti-Code im Wortsinn: ein Teller voller Fäden, von denen sich keiner herausziehen ließ, ohne den Rest mitzubewegen. Trotzdem war dieses Chaos zumindest prinzipiell nachvollziehbar. Hinter jeder unnötigen Verzweigung und jeder globalen Variable stand ein Mensch, der sie absichtlich oder irrtümlich eingebaut hatte. Es gab ein mentales Modell, so schlecht es auch gewesen sein mag.

Dreißig Jahre später sitze ich vor einem Diff, das ein Coding-Agent in vier Sekunden erzeugt hat. Sauber eingerückt, konsistent benannt und kommentiert mit Docstrings, die klingen, als hätte sie eine erfahrene Kollegin geschrieben. Mittendrin steht der Aufruf einer Bibliotheksfunktion, die es nicht gibt und nie gegeben hat. Das ist kein Spaghetti-Code mehr. Es ist Halluzinations-Lasagne: ordentlich geschichtet, appetitlich angerichtet und mit einer Schicht, die wie Bolognese aussieht, aber keine ist. Die Oberfläche ist makellos, und genau das macht den Defekt so schwer erkennbar. Das neue Problem ist nicht sichtbares Chaos, sondern trügerische Ordnung. Hinter dem Code steht kein belastbares mentales Modell, sondern eine statistisch plausible Ausgabe.

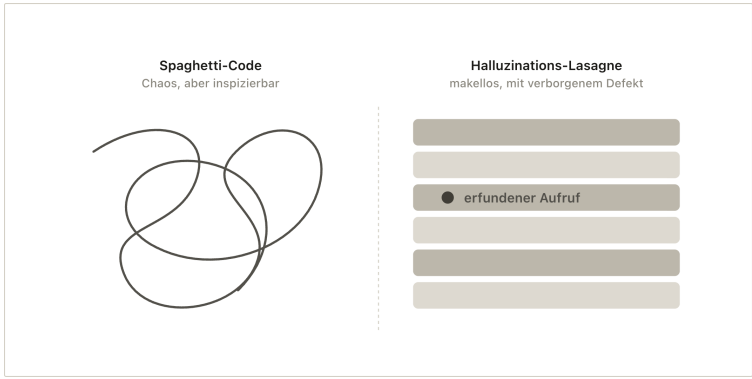


Abbildung 1.1: Der Spaghetti-Code zeigt sein Chaos offen. Die Halluzinations-Lasagne tarnt ihren Defekt als Ordnung.

Gefährlich ist nicht der einzelne erfundene Funktionsaufruf. Menschen irren ebenfalls. Sie können einen Fehler jedoch verstehen und ihr Verhalten dauerhaft ändern. Eine Korrektur im Chat verändert die Gewichte eines Sprachmodells nicht. Im nächsten Lauf kann derselbe Fehler wiederkehren. Daraus folgt eine unbequeme Asymmetrie: Wer falsche Ausgaben nicht sicher erkennt, muss jede folgenreiche Ausgabe prüfen. Die saubere Fassade verschärft das Problem. Sie lässt eine Prüfung gerade dann entbehrlich wirken, wenn sie am nötigsten wäre. Jede ungeprüfte Generierung trägt weitere Unsicherheit ins System.

Deshalb in Panik zu verfallen und jeden Einsatz von KI-Modellen abzulehnen, hilft wenig. Ebenso wenig hilft die Hoffnung auf das nächste Wundermodell, das alle grundlegenden Probleme lösen soll. Aus Sicht der First Principles, also von den Grundannahmen her, lohnt sich eine andere Frage: „Besteht die Aufgabe von Ingenieuren nicht seit jeher darin, aus unsicherem Input verlässlichen Output zu erzeugen?“ Eine Brücke trägt, obwohl Werkstoffkennwerte streuen. Ein Funknetz überträgt Daten durch ein Medium voller Störungen. Ein Prozessor liefert stabile Ergebnisse, obwohl seine Bauteile thermischem und elektrischem Rauschen ausgesetzt sind. Verlässlichkeit entsteht nicht durch die Abwesenheit von Unsicherheit, sondern durch ihre systematische Beherrschung.

Claude Shannon gab dieser Frage 1948 eine mathematisch präzise Form. Seine Informationstheorie beschreibt, wie sich Nachrichten trotz eines verrauschten Kanals zuverlässig übertragen lassen. Ursprünglich ging es um

Telefonleitungen und Funkstrecken.

Als Denkmodell für Coding-Agenten erklärt Shannons Kanal, warum plausible Ausgaben noch keine verlässlichen Ausgaben sind. Das Wort Rauschen bezeichnet dabei jede Abweichung zwischen beabsichtigtem und erzeugtem Ergebnis. Dazu gehören Sampling, Mehrdeutigkeit, fehlender Kontext und Grenzen des Modells.

Der Weg dieses Buches folgt dem Signal. Kapitel 2 legt Shannons Begriffe frei. Kapitel 3 verdichtet die Absicht am Sender. Kapitel 4 hält Aufgabe und Kontext im beherrschbaren Bereich. Kapitel 5 baut den prüfenden Empfänger. Kapitel 6 schließt daraus Regelkreise. Erst danach kehrt das Buch zu den vertrauten Praktiken der Softwareentwicklung zurück. Die Werkzeuge entlang dieses Weges werden wechseln. Die Aufgaben bleiben: Absicht klären, Komplexität begrenzen und Fehler erkennen.

2 Das Shannon-Prinzip

Bevor Shannons Modell auf Coding-Agenten übertragen wird, müssen seine Begriffe sitzen. Zwei Gedanken tragen das Buch. Erstens misst Information nicht Bedeutung, sondern statistische Überraschung. Zweitens kann ein geeigneter Code Nachrichten auch durch einen verrauschten Kanal zuverlässig übertragen, sofern Rate und Kanalkapazität zueinander passen. Der erste Gedanke erklärt, was am Sender zählt. Der zweite ordnet Kanal und Empfänger.

2.1 Information ist Überraschung, nicht Bedeutung

Im Alltag liegen Information und Bedeutung eng beieinander. Was wichtig klingt, gilt als informativ, was banal klingt, als leer. Shannon trennte diese Begriffe 1948 bewusst voneinander. Für seine Theorie zählt nicht, was eine Nachricht bedeutet, sondern wie wahrscheinlich sie vor ihrem Eintreffen war. Eine erwartbare Nachricht kann semantisch wichtig sein und trotzdem wenig neue Information tragen. Eine völlig belanglose Nachricht kann dagegen hochinformativ sein, wenn niemand mit ihr gerechnet hat.

Ein Monitoring-System macht den Unterschied greifbar. Die Meldung „Alle 10.000 Systeme laufen“ ist für den Betrieb enorm wichtig. Erscheint sie jedoch seit Monaten jede Minute, ist sie vollkommen erwartbar und trägt im Sinne Shannons nur wenig neue Information. Meldet dasselbe System plötzlich „Im Logfile steht das Wort Schweinssülze“, ist das betrieblich vermutlich bedeutungslos, aber statistisch höchst überraschend. Shannons Theorie weist der Schweinssülze deshalb mehr Bits zu, nicht weil sie wichtiger wäre, sondern weil sie unwahrscheinlicher war.

Diese Trennung war der entscheidende Schritt. Bedeutung hängt von Menschen, Situationen und Vorwissen ab. Statistische Überraschung lässt sich

dagegen messen. Für ein Ereignis mit der Wahrscheinlichkeit p beträgt der Informationsgehalt

$$I(x) = -\log_2 p(x).$$

Je unwahrscheinlicher ein Ereignis ist, desto mehr Bits trägt sein Eintreten. Ein Bit entspricht dabei der Information aus einer Entscheidung zwischen zwei gleich wahrscheinlichen Möglichkeiten. Die Entropie H ist der erwartete Informationsgehalt einer Quelle:

$$H(X) = -\sum_{x} p(x) \log_2 p(x).$$

Sie beschreibt, wie viele Bits pro Symbol bei einer optimalen, verlustfreien Codierung langer Symbolfolgen im Mittel mindestens nötig sind. Entropie misst damit Unsicherheit, nicht Textmenge und nicht Bedeutung. Eine gut vorhersagbare Quelle hat niedrige Entropie. Eine schwer vorhersagbare Quelle hat hohe.

Drei Beispiele machen den Unterschied sichtbar:

Quelle	Entropie pro Wurf
faire Münze	1 Bit
fairer Würfel	$\log_2 6 \approx 2{,}585$ Bit
Münze mit 99 Prozent Kopf	ungefähr $0{,}081$ Bit

Bei der fairen Münze sind beide Ausgänge gleich wahrscheinlich. Ein Wurf liefert deshalb genau ein Bit. Ein fairer Würfel besitzt sechs gleich wahrscheinliche Ausgänge. Eine feste Binärcodierung benötigt dafür drei Bits pro einzelnem Wurf, weil sechs Zustände in zwei Bits nicht hineinpassen. Codiert man jedoch lange Folgen von Würfen gemeinsam, nähert sich der durchschnittliche Bedarf dem theoretischen Minimum von etwa 2,585 Bits pro Wurf.

Eine stark verzerrte Münze sieht äußerlich wie dieselbe binäre Quelle aus, ist aber fast vollständig vorhersagbar. Lange Kopfserien lassen sich sehr

kompakt codieren, weshalb im Mittel nur etwa 0,081 Bit pro Wurf übrig bleibt.

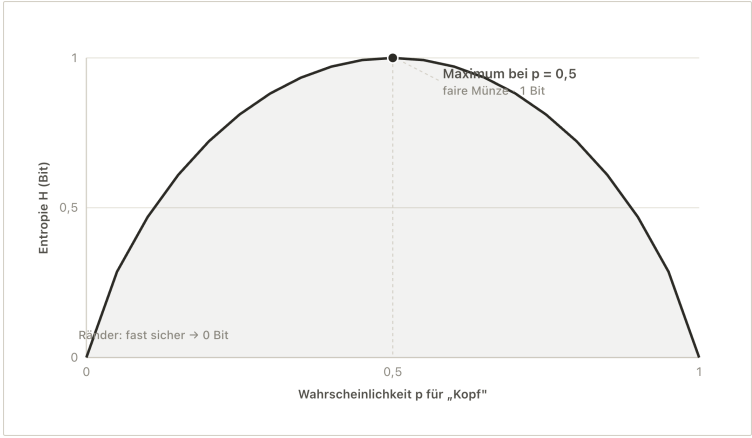


Abbildung 1.2: Die binäre Entropie erreicht beim fairen Münzwurf ihr Maximum von einem Bit. Genau dort ist der Ausgang am unsichersten. Daraus ergibt sich die Leitfrage dieses Buches: Wie wird aus unsicherem Input verlässlicher Output?

Der Informationsgehalt ist also keine feste Eigenschaft eines Symbols. Er hängt davon ab, wie wahrscheinlich dieses Symbol in der jeweiligen Situation war. Je weniger das zugrunde gelegte Wahrscheinlichkeitsmodell damit gerechnet hat, desto mehr Information trägt es.

Genau darin lag die Revolution. Shannon machte aus dem unscharfen Alltagsbegriff Information eine messbare Größe. Warum dafür ausgerechnet ein Logarithmus nötig ist, zeigt ein Gedankenexperiment.

Vor einem fairen Münzwurf gibt es zwei gleich wahrscheinliche Möglichkeiten. Sobald das Ergebnis feststeht, ist eine Frage mit zwei möglichen Antworten geklärt: Kopf oder Zahl. Die dadurch verschwundene Unsicherheit entspricht einem Bit. Bei zwei unabhängigen Würfen gibt es bereits vier gleich wahrscheinliche Folgen: Kopf-Kopf, Kopf-Zahl, Zahl-Kopf und Zahl-Zahl. Um eine davon eindeutig zu bestimmen, müssen zwei solcher Fragen beantwortet werden, eine pro Wurf. Das Ergebnis trägt deshalb zwei Bits.

Die Wahrscheinlichkeit jeder konkreten Folge beträgt jedoch nur $1/4$, denn $1/2 \cdot 1/2 = 1/4$. Hier treffen zwei Rechenarten aufeinander. Die Wahr-

scheinlichkeiten unabhängiger Ereignisse werden multipliziert, ihre Informationsgehalte sollen sich dagegen addieren. Gesucht ist also eine Funktion I, für die $I(p \cdot q) = I(p) + I(q)$ gilt. Genau das leistet der Logarithmus, denn $\log(p \cdot q) = \log p + \log q$. Weil der Logarithmus einer Wahrscheinlichkeit zwischen null und eins negativ ist, sorgt das Minuszeichen für einen positiven Informationsgehalt. So entsteht Shannons Formel $I(x) = -\log_2 p(x)$.

Die Basis 2 macht aus diesem Maß die Einheit Bit. Ein Ereignis mit der Wahrscheinlichkeit $1/2$ liefert ein Bit, eines mit der Wahrscheinlichkeit $1/4$ zwei Bits und eines mit der Wahrscheinlichkeit $1/8$ drei Bits. Anschaulich zählt der Wert, wie viele Entscheidungen zwischen jeweils zwei gleich wahrscheinlichen Möglichkeiten nötig sind, um ein Ergebnis festzulegen. Andere Logarithmusbasen funktionieren ebenfalls, erzeugen aber andere Einheiten. Die Basis 2 passt sowohl zu dieser Folge binärer Entscheidungen als auch zur Arbeitsweise digitaler Rechner.

Ein Bit bezeichnet daher nicht einfach die Ziffer 0 oder 1. Diese Ziffern stellen lediglich die beiden möglichen Zustände dar. Das Bit misst, wie viel Unsicherheit durch die Festlegung auf einen dieser Zustände verschwindet. Mit dieser Einheit lassen sich ansonsten völlig verschiedene Dinge wie Texte, Fotos, Musik und Programmcode quantitativ beschreiben. Ihre Bedeutung bleibt verschieden, ihre Informationsmenge wird vergleichbar. Diese Abstraktion bildet eine Grundlage der digitalen Welt. Alles lässt sich als Folge von Bits darstellen, speichern und verarbeiten. Erst die spätere Interpretation macht daraus Buchstaben, Bildpunkte, Töne oder Maschinenbefehle.

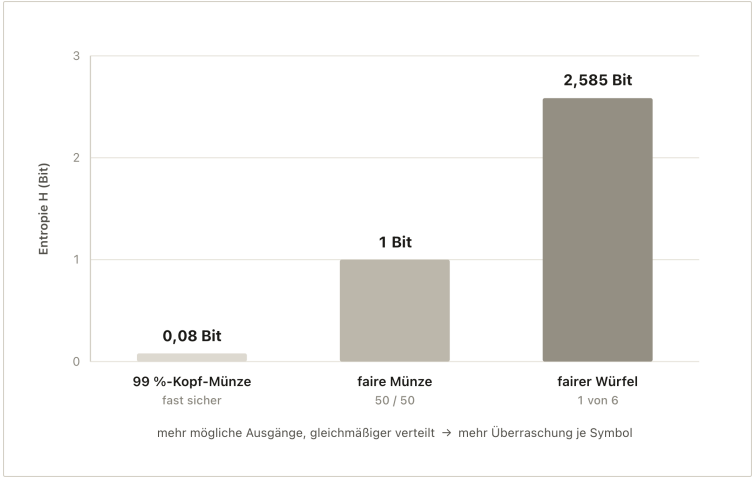


Abbildung 2.1: Entropie misst, wie überrascht ein Empfänger im Schnitt vom nächsten Symbol ist. Eine Münze, die fast immer Kopf zeigt, trägt kaum Information (0,08 Bit): das Ergebnis steht praktisch schon fest. Eine faire Münze trägt genau 1 Bit, ein fairer Würfel mit sechs gleich wahrscheinlichen Seiten 2,585 Bit. Je mehr gleichverteilte Ausgänge, desto höher die Entropie und desto mehr Information steckt in jedem einzelnen Zeichen.

2.2 Kanal, Rauschen und Kapazität

Shannons zweiter großer Gedanke betrifft die Übertragung. Sein Modell besteht aus fünf Hauptblöcken und einer Rauschquelle. Eine Informationsquelle erzeugt die Nachricht. Ein Sender übersetzt sie in ein Signal. Der Kanal transportiert dieses Signal und wird dabei von einer Rauschquelle gestört. Ein Empfänger rekonstruiert aus dem angekommenen Signal die Nachricht. Die Senke ist ihr Ziel.

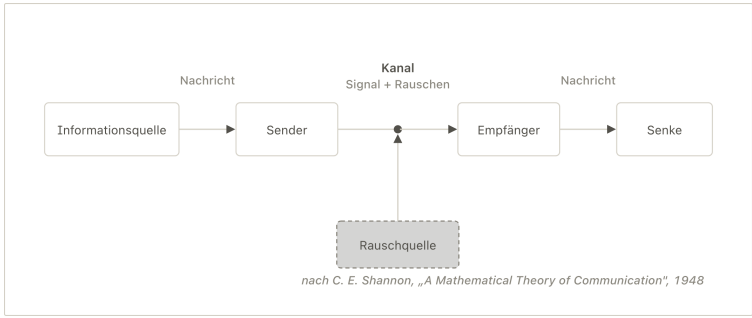


Abbildung 2.2: Shannons Modell kennt fünf Bauteile: eine Informationsquelle, die eine Nachricht erzeugt, einen Sender, der sie in ein Signal übersetzt, einen Kanal, der das Signal überträgt und dabei durch eine Rauschquelle gestört wird, einen Empfänger, der aus dem gestörten Signal die Nachricht rekonstruiert, und eine Senke als Ziel. Das Rauschen ist dabei keine Panne, sondern eine Eigenschaft des Mediums.

Rauschen ist in diesem Modell keine überraschende Panne, sondern Teil der Kanalbeschreibung. Ein Kupferkabel rauscht thermisch, eine Funkstrecke fängt fremde Signale ein. Eine geeignete Codierung und Redundanz können solche Störungen beherrschbar machen. Sie schaffen das Rauschen nicht ab.

Jeder Kanal besitzt eine Kapazität C. Je nach Modell wird sie in Bits pro Kanalnutzung oder in Bits pro Sekunde angegeben. Das Noisy-Channel-Coding-Theorem, also Shannons Satz über die Codierung verrauschter Kanäle, zieht eine präzise Grenze.

Liegt die Übertragungsrate R unterhalb von C, kann die Fehlerwahrscheinlichkeit mit geeigneten Codes und hinreichend langen Blöcken beliebig klein werden. Bei einer endlichen Übertragung wird sie dadurch nicht automatisch null, aber sie lässt sich kontrollieren. Liegt R oberhalb von C, kann kein Code die Fehlerwahrscheinlichkeit dauerhaft gegen null drücken. Die Kapazität ist damit keine bloße Qualitätsstufe, sondern die Grenze zuverlässiger Kommunikation.

Für Coding-Agenten ist das eine Analogie, kein Naturgesetz. Für Sprachmodelle ist keine Shannon-Kapazität bekannt, die sich in „Bits Absicht pro Generierung“ angeben ließe. Ebenso wenig ist eine scharfe Schwelle nachgewiesen. Empirisch zeigt sich meist eine ansteigende Fehlerwahrscheinlichkeit, keine sauber vermessene Klippe. Kapitel 4 behandelt diese Gren-

ze deshalb als Betriebsmodell, das für ein bestimmtes Modell, mit einem bestimmten Agenten-Harness und eine Aufgabenklasse kalibriert werden muss.

Im Buch übernimmt die Modellgenerierung die Rolle des Kanals. Ihr Rauschen ist jede Abweichung zwischen Absicht und Ergebnis. Durch das unterschiedliche Sampling aus den Trainingsdaten des Modells kann solche Abweichungen streuen. Mehrdeutigkeit, unvollständiger Kontext und Modellgrenzen können auch eine an sich deterministische Ausgabe zuverlässig in die falsche Richtung lenken.

Das folgende Beispiel macht diese Abhängigkeit vom Kontext unmittelbar sichtbar. Nach dem Prompt

```
1 + 1 =
```

ist die Zahl 2 mit hoher Wahrscheinlichkeit die nächste Ausgabe. Ein vorangestellter Kontext verschiebt jedoch die Verteilung:

```
Team Synergien
1 + 1 =
```

Nun kann 3 zur wahrscheinlichsten Fortsetzung werden. Die Arithmetik hat sich nicht verändert. Verändert hat sich die Wahrscheinlichkeitsverteilung des Modells, weil es nun eher das bekannte Bild aufgreift, nach dem ein gutes Team gemeinsam mehr erreicht als die Summe seiner Teile. Information entsteht damit nie aus einem Zeichen allein, sondern aus dessen Wahrscheinlichkeit unter einem bestimmten Kontext.

„Intentions-Bits“ sind daher keine Messeinheit. Der Ausdruck dient nur als Bild für unabhängige Entscheidungen, die eine Generierung bewahren muss. Je mehr davon gleichzeitig offen sind, desto größer wird das Risiko, dass lokal plausibler, aber global falscher Code entsteht.

2.3 Der QR-Code als gelöste Kanalcodierung

Wie kontrollierte Redundanz funktioniert, zeigt ein vertrauter Alltagsgegenstand: der QR-Code. Reed-Solomon-Codes ergänzen seine Nutzdaten

um zusätzliche Prüfsymbole. Der Empfänger kann damit fehlende oder beschädigte Codewörter rekonstruieren, solange genügend unverfälschte Information übrig bleibt.

QR-Codes kennen vier Fehlerkorrekturstufen. Stufe L kann nominell ungefähr sieben Prozent beschädigter Codewörter wiederherstellen, M etwa fünfzehn, Q etwa fünfundzwanzig und H etwa dreißig Prozent. Die Werte beziehen sich weder pauschal auf eine überklebte Fläche noch garantieren sie Erfolg bei jedem Schadensmuster. Ein kleiner Aufkleber über einem Suchmuster kann den Code unlesbar machen. Verteilte Schäden in Datenbereichen verkraftet er oft erstaunlich gut.

Die Zuverlässigkeit entsteht also nicht durch Magie und auch nicht durch bloße Wiederholung. Der Sender fügt gezielt Redundanz nach einem bekannten Schema hinzu, und der Empfänger weiß, wie er sie zur Rekonstruktion nutzen muss. Genau diese Arbeitsteilung ist für Coding-Agenten interessant. Eine Absicht kann sich gleichzeitig in einer typisierten Signatur, einem Schema, einem Beispiel und einem Akzeptanztest ausdrücken. Jede Darstellung trägt einen Teil derselben Entscheidung. Erst ein Empfänger, der diese Spuren prüft und Abweichungen zurückweist, macht aus zusätzlichem Text tatsächlich Fehlerkorrektur.

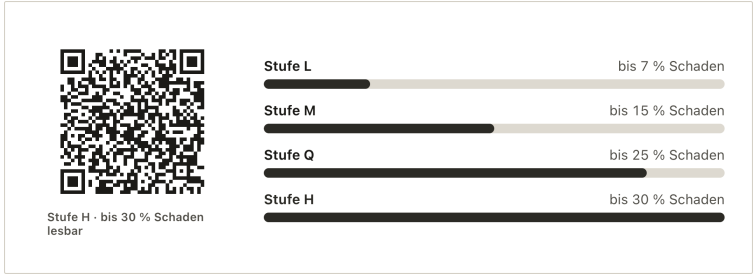


Abbildung 2.3: Die Daten eines QR-Codes sind mit Reed-Solomon-Codes abgesichert, jedes Datenbyte trägt Information über die anderen mit. Vier Fehlerkorrekturstufen geben eine harte Garantie: Ein zu einem Drittel überklebter oder zerkratzter Code lässt sich auf Stufe H immer noch fehlerfrei lesen. Genau diese Garantie fehlt, wenn man einen Agenten einfach drauflos generieren lässt. Wer agentischen Code so verlässlich haben will, braucht explizite Redundanz und einen Empfänger, der sie zur Korrektur nutzt. QR: github.com/ingo-eichhorst/agent-readyness

2.4 Entropie als Arbeitsmodell für Software

Hier ist eine begriffliche Trennung wichtig. In Shannons Theorie misst Entropie nicht, wie chaotisch etwas aussieht. Sie misst die Unsicherheit vor einem möglichen Ergebnis: Welche Ausgaben sind möglich, und wie wahrscheinlich ist jede davon? Eine Codebasis, die bereits auf der Festplatte liegt, ist dagegen ein konkretes Ergebnis und keine Wahrscheinlichkeitsverteilung. Sie besitzt deshalb nicht automatisch Shannon-Entropie, nur weil sie inkonsistent oder unübersichtlich ist.

Der Begriff „strukturelle Entropie“ wird im Folgenden bewusst als Arbeitsmodell verwendet. Er beschreibt, wie offen der Lösungsraum für eine Änderung ist. Je mehr unterschiedliche und untereinander widersprüchliche Implementierungen im Repository plausibel erscheinen, desto höher ist diese strukturelle Entropie.

Diese Unsicherheit ist niedrig, wenn das Repository Entscheidungen klar bindet. Dasselbe Problem wird überall nach demselben Muster gelöst, Schnittstellen sind eindeutig, Namen folgen einer Konvention und Tests markieren die zulässigen Verhaltensweisen. Sie ist hoch, wenn für dieselbe Aufgabe viele lokal plausible, aber untereinander widersprüchliche Lösungen offenstehen.

Davon zu unterscheiden ist die semantische Entropie. Sie misst, wie stark mehrere Generierungen bei unveränderter Spec in ihren Bedeutungen und Designentscheidungen streuen. Zwei unterschiedlich formulierte Lösungen mit demselben Verhalten gehören dabei zusammen. Strukturelle Entropie fragt dagegen, wie viele untereinander widersprüchliche Implementierungswege das Repository plausibel erscheinen lässt. Zwei Patches können semantisch dasselbe leisten und trotzdem strukturell weit auseinanderliegen, etwa weil einer eine vorhandene Domänenschnittstelle nutzt und der andere die Logik in einem Controller dupliziert. Präzise Specs binden die semantische Auswahl, klare Modulgrenzen und Architekturregeln die strukturelle.

Ein Sprachmodell erzeugt Code nicht, indem es einen fertigen Plan aus einem Speicher abruft. Es berechnet für den bisherigen Kontext zunächst, welche Tokens als Nächstes möglich sind und wie wahrscheinlich jedes davon ist. Ein Auswahlverfahren entscheidet sich für eines dieser Tokens, fügt

es dem Kontext hinzu und startet dieselbe Rechnung erneut. So entsteht die Ausgabe Schritt für Schritt. Eine frühe Auswahl verändert dabei, welche Fortsetzungen später noch wahrscheinlich sind. Die Generierung folgt deshalb einem Pfad durch viele aufeinanderfolgende Wahrscheinlichkeitsverteilungen.

Für Software hat das eine konkrete Folge. Eine Anforderung wie „Füge einen Endpunkt für Benutzerprofile hinzu“ lässt zahlreiche Entscheidungen offen: Liegt die Validierung im Controller oder im Service? Werden Fehler als Exception oder als Rückgabewert behandelt? Folgt der Endpunkt dem Namensschema des Repositorys oder einem verbreiteten Standard?

Das Modell trifft diese Entscheidungen nicht im luftleeren Raum. Der Modell-Prior ist seine Ausgangserwartung: das, was es aufgrund seiner Trainingsdaten für wahrscheinlich hält, bevor der projektspezifische Kontext eine genauere Richtung vorgibt. Dieser Prior ist kein gespeicherter Katalog von Regeln. Er entsteht daraus, welche Muster das Modell während des Trainings wie häufig und in welchen Zusammenhängen gesehen hat. Tauchte Validierung dort überwiegend in Controllern auf, erhält diese Variante zunächst mehr Wahrscheinlichkeit als eine ungewöhnliche Projektkonvention, die Validierung in einen Service zu legen.

Prompt und Repository-Kontext können diese Ausgangsverteilung verschieben. Wo beide schweigen, setzt sich jedoch meist der Prior durch. Verbreitete Framework-Konventionen, bekannte Architekturmuster und übliche Namen liegen deshalb näher als exotische Varianten. Passt der Prior zum Repository, füllt das Modell offene Stellen oft sinnvoll aus. Weichen die projektspezifischen Regeln vom Mainstream ab oder konkurrieren mehrere gelernte Standards miteinander, kann derselbe Prior plausible Lösungen hervorbringen, die lokal funktionieren und dennoch nicht zum Projekt passen.

Fehlen klare Architekturentscheidungen, Konventionen und Akzeptanzkriterien, entscheidet somit der Modell-Prior, welches Muster sich in einem Lauf durchsetzt. Verschiedene Läufe können dabei unterschiedliche, jeweils plausible Antworten wählen und die strukturelle Entropie erhöhen.

Die Wahrscheinlichkeitsnatur des Modells bedeutet allerdings nicht, dass jede Generierung die Codebasis zwangsläufig unordentlicher macht. Kla-

re Constraints, also bindende Vorgaben, engen die möglichen Lösungen ein. Compiler, Schemata und Tests prüfen anschließend, ob die gewählte Lösung diese Vorgaben einhält. Agenten können dadurch bestehende Muster fortsetzen, Abweichungen reparieren und Code sogar vereinheitlichen. Die belastbare Aussage ist daher enger: Ohne bindende Vorgaben und einen prüfenden Empfänger fehlt die Rückstellkraft, die eine plausible Abweichung erkennt und zum gemeinsamen Muster des Repositorys zurückführt.

Genau darin liegt die Ingenieursaufgabe. Nicht die Stochastik muss verschwinden. Die Pipeline muss den Lösungsraum einengen, kontrollierte Redundanz bereitstellen und Abweichungen korrigieren, bevor sie Teil des Systems werden.

2.5 Shannons Modell, übersetzt auf Coding-Agenten

Überträgt man die funktionalen Rollen aus Shannons Modell auf agentisches Coding, ergibt sich folgende Zuordnung:

Shannon-Block	Beim Coding-Agenten
Informationsquelle	Absicht: ein Feature, ein Bug oder eine fachliche Entscheidung
Sender	Agent samt Harness, der Absicht in Spezifikation, Prompt und Kontext codiert
Kanal	Generierung durch das Sprachmodell
Rauschquelle	Sampling, Mehrdeutigkeit, unvollständiger Kontext, Modellgrenzen und technische Nichtdeterminismen
Empfänger	Parser, Compiler, Linter, Schema-Prüfungen, Tests und Reviews

Shannon-Block	Beim Coding-Agenten
Senke	das akzeptierte und laufende System

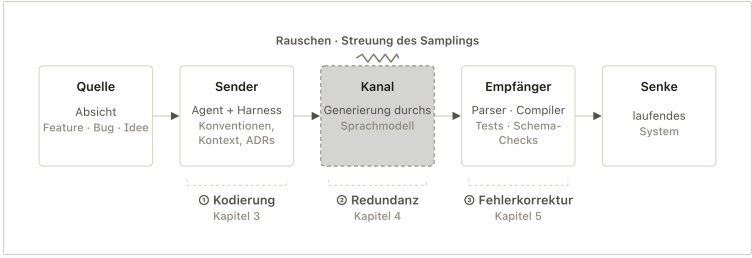


Abbildung 2.4: Dieselbe Kette, die Shannon 1948 für die Nachrichtenübertragung zeichnete, trägt das ganze Buch. Die Absicht ist die Quelle, der Agent samt Harness der Sender, die Generierung durchs Sprachmodell der verrauschte Kanal, die Prüfkaskade der Empfänger, das laufende System die Senke. Gegen das Rauschen des Kanals wirken drei Hebel, je einer am Sender, im Kanal und am Empfänger, die die nächsten drei Kapitel entfalten.

Der Harness ist das Gerüst aus Konventionen, Kontext und Werkzeugen rund um das Modell. Er hilft dem Sender, eine diffuse Absicht in ein verarbeitbares Signal zu übersetzen. Der Empfänger prüft nach der Generierung nicht nur Syntax und Typen, sondern rekonstruiert aus den vorhandenen Constraints, ob die ursprüngliche Absicht tatsächlich angekommen ist.

Auch diese Zuordnung ist ein Denkmodell, keine Identität. Ein Prompt ist kein Funksignal, ein Test kein Reed-Solomon-Decoder und ein Sprachmodell kein Kupferkabel. Ihr Nutzen liegt in der Trennung der Verantwortlichkeiten. Sie verhindert, dass jedes Problem vorschnell zum Promptproblem erklärt wird.

Daraus folgen drei Hebel. Kapitel 3 behandelt die Quellencodierung am Sender: Wie lässt sich Absicht so verdichten, dass wenig Raum für Fehldeutung bleibt? Kapitel 4 behandelt Rate und Redundanz im Kanal: Wie helfen ein passendes Modell, ein schlanker Kontext, kleine Aufgaben und getrennte Arbeitsphasen? Kapitel 5 behandelt den Empfänger: Wie erkennen Compiler, Tests, Schemata und Reviews das verbleibende Rauschen, und wie liefern sie ein brauchbares Korrektursignal?

Shannon hat diese konkreten Methoden für Software nicht bewiesen. Seine Theorie liefert jedoch die Rollen, die Grenzen und die entscheidende Frage nach zuverlässiger Übersetzung von Absicht zu funtionierender Software. Wer nur am Prompt feilt, optimiert den Sender und lässt den Empfänger blind. Wer nur Tests stapelt, kann am Ende sehr präzise das falsche Ziel prüfen. Verlässlichkeit entsteht erst aus dem Zusammenspiel von klarer Codierung, beherrschbarer Übertragung und wirksamer Fehlerkorrektur.

Quellen dieses Abschnitts

- Claude E. Shannon: A Mathematical Theory of Communication. Bell System Technical Journal, 1948. https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf
- DENSO WAVE: Error correction feature und QR-Code-Grundlagen. https://www.qrcode.com/en/about/error_correction.html
- Horace He und Thinking Machines Lab: Defeating Nondeterminism in LLM Inference. Batchinvariante Kernel können deterministische Decodierung reproduzierbar machen. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/
- LLM-42: Enabling Determinism in LLM Inference with Verified Speculation. arXiv:2601.17768. https://arxiv.org/abs/2601.17768

3 Quellencodierung am Sender

Shannons Quellencodierung komprimiert, ohne Information wegzuwerfen. Sie nutzt dazu die Wahrscheinlichkeiten der Quelle. Häufige und damit gut vorhersagbare Symbole erhalten kurze Codes. Seltene Symbole erhalten längere.

Ein Getränkestand macht das Prinzip greifbar. Die Hälfte aller Bestellungen lautet Kaffee, ein Viertel Tee und jeweils ein Achtel Kakao oder Wasser. Ein fester Code weist jeder der vier Möglichkeiten zwei Bits zu. Hundert typische Bestellungen benötigen dann 200 Bits. Ein Präfixcode kann Kaffee als `0`, Tee als `10`, Kakao als `110` und Wasser als `111` darstellen. Die Folge bleibt eindeutig lesbar, braucht im Mittel aber nur 175 Bits.

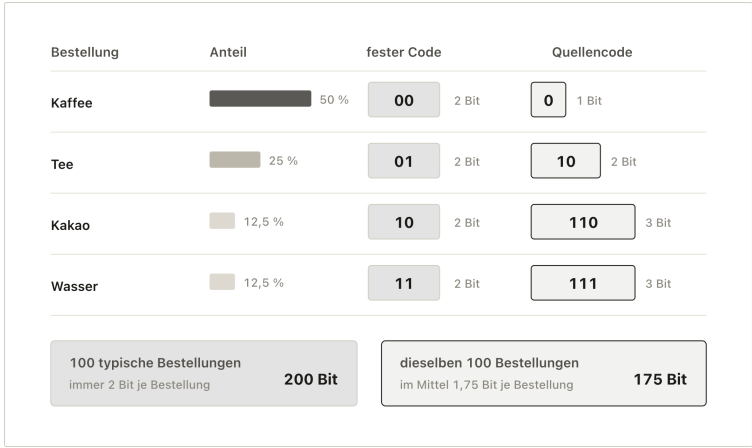


Abbildung 3.1: Bei einer festen Codierung benötigen alle vier Getränke jeweils zwei Bits. Die Quellencodierung gibt dem häufigen Kaffee den Ein-Bit-Code 0 und reserviert längere Codes für seltene Bestellungen. Im Mittel sinkt der Bedarf dadurch von 2 auf 1,75 Bit je Bestellung. Keine Information geht verloren: Keines der Codewörter ist der Anfang eines anderen, deshalb bleibt die Folge eindeutig decodierbar.

Keine Bestellung ist verschwunden. Der Code verwendet lediglich weniger Platz für das Erwartbare. Die Entropie aus Kapitel 2 markiert die theoretische Untergrenze für lange Folgen. Das führt zum Problem dieses Kapitels: Eine Spezifikation kann sehr lang sein und trotzdem kaum Entscheidungen übertragen.

Beim Coding-Agenten bilden Agent und Harness den Sender. Der Harness umfasst Regeln, Werkzeuge und die Auswahl des Kontexts. Gemeinsam übersetzen sie eine diffuse Absicht in eine Spezifikation, kurz Spec, einen Prompt und die relevanten Projektinformationen. Ein guter Sender überträgt dabei nicht möglichst viel Text. Er überträgt vor allem das, was das Zielmodell nicht zuverlässig erraten kann.

Ob eine Aussage neu ist, hängt vom Modell-Prior aus Kapitel 2 ab. Ein Satz trägt Signal, wenn er eine Entscheidung bindet oder plausible Lösungen ausschließt. Projektspezifische Konventionen, bewusste Abweichungen vom Üblichen und seltene Randfälle sind daher oft wertvoll. Ein ausdrücklich genannter Standard kann ebenfalls tragen, etwa wenn mehrere übliche Varianten konkurrieren. Entscheidend ist nicht, ob ein Satz bekannt klingt. Entscheidend ist, ob er für dieses Modell in dieser Aufgabe Unsicherheit verringert.

Prompt-Kompressoren wie LLMLingua zeigen, dass manche Eingaben stark gekürzt werden können, ohne die jeweilige Messgröße entsprechend zu verschlechtern. Für einzelne Aufgaben berichtet die Arbeit Kompressionsraten bis zum Zwanzigfachen. Daraus folgt jedoch keine allgemeine Löschregel. Statistisch überraschende Tokens sind nicht automatisch fachlich wichtig, und eine Kompressionsmetrik kann Zahlen, Grenzwerte oder Negationen falsch gewichten. Das Prinzip lautet deshalb nicht „kurz um jeden Preis“, sondern „jede folgenreiche Entscheidung sichtbar“.

Werkzeuge dafür sind typisierte Anforderungen, wenige konkrete Beispiele, Styleguides und Architecture Decision Records, kurz ADRs. Sie halten Entscheidungen fest, die der Modell-Prior nicht sicher liefert. Vibe-Coding wählt den Gegenweg: Die Spec bleibt dünn, das Modell füllt die Lücken. Bei einer gewöhnlichen Aufgabe kann das genügen. Bei projektspezifischen Regeln wird die Qualität der Quellencodierung entscheidend.

3.1 Das Spec-Driven-Paradox

Spec Driven Development, also Entwicklung entlang einer vorab formulierten Spezifikation, verspricht Klarheit. Trotzdem kann viel Text mit erstaunlich wenig Entscheidung entstehen. Besonders leicht geschieht das, wenn ein Sprachmodell aus einer vagen Aufgabe selbst eine lange Spec formuliert. Ohne neue Fakten, Rückfragen oder bewusste Festlegungen ordnet es vor allem seine eigenen Vermutungen.

Eine solche Spec kann dennoch nützen. Sie gliedert die Aufgabe, legt Widersprüche frei und bringt offene Fragen an die Oberfläche. Aktuelle Arbeiten zur Anforderungsverfeinerung zeigen, dass modellgestützte Überarbeitung die Lösung von Aufgaben verbessern kann. Der Gewinn entsteht aber durch Klärung und Präzisierung, nicht durch zusätzliche Seiten. Eine Spec kann keine fachliche Entscheidung erfinden, die in ihrer Quelle fehlt.

Verfasst dasselbe Modell anschließend aus derselben vagen Ausgangslage den Code, können Spec und Implementierung denselben Irrtum teilen. Die Spec sieht dann wie ein zweiter Beleg aus, stammt aber aus derselben falschen Vermutung. Darin liegt das Spec Driven Paradox: Viele Bytes müssen die Unsicherheit nicht verringern.

Sätze wie „Die Lösung soll sauber, effizient und wartbar sein“ klingen richtig, schließen aber kaum eine konkrete Lösung aus. Informationsdicht wird eine Spec dort, wo sie eine offene Dimension schließt, einen Randfall festlegt, eine Prior-Annahme überschreibt oder eine prüfbare Grenze setzt. Die entscheidende Frage lautet daher: Welche Entscheidung würde das Zielmodell ohne diesen Satz anders oder uneinheitlich treffen?

3.2 Klarheit braucht zwei Messachsen

Eine Spec ist nicht klar, weil sie klar klingt. Sie ist klar, wenn sie den Lösungsraum in die beabsichtigte Richtung verengt. Das lässt sich mit zwei getrennten Fragen prüfen. Erstens: Welche Aussage verändert die Ausgabe? Zweitens: Wie stark streuen wiederholte Ausgaben bei unveränderter Spec?

Für die erste Frage läuft dieselbe Aufgabe mehrfach mit der gleichen vollständigen Spec. Wählt das Modell nach der Entfernung andere Designs, hat die Aussage eine Entscheidung gebunden. Das ist die Shannon-Perspektive: Der Satz hat die bedingte Unsicherheit über den Output verändert.

Ein Satz wie „Speichere die Daten in PostgreSQL“ trägt beispielsweise dann, wenn das Modell ohne ihn zwischen PostgreSQL, SQLite und einer nur im Arbeitsspeicher gehaltenen Lösung schwankt. Ein Satz wie „Die Implementierung soll sauber und wartbar sein“ trägt wenig, wenn seine Entfernung nichts Messbares verändert. Bleiben die beobachteten Ausgaben in beiden Varianten gleich, war die entfernte Aussage für genau dieses Modell und diese Aufgabe wahrscheinlich redundant.

Um zwei unterschiedliche Fragen auseinanderzuhalten, werden zwei Messgrößen verwendet. Die erste ist die Ablation Sensitivity, also die Empfindlichkeit des Outputs gegenüber dem Entfernen einer Aussage. Dafür wird die Spec in Spans zerlegt. Ein Span ist eine einzelne Aussage, meist ein Satz oder Listenpunkt. Anschließend wird für jeden Span verglichen, wie sich die Ausgaben mit und ohne ihn verteilen. Verschiebt sich die Verteilung stark, steuert der Span eine Entscheidung. Bleibt sie gleich, war seine zusätzliche Wirkung in diesem Versuch gering.

Das Begleitexperiment macht beide Fragen an einem einzigen, absichtlich überschaubaren Problem messbar. `gpt-4o-mini` sollte eine REST-API für die Ressource `User` entwerfen. Über fünfzehn Begleittexte hinweg blieben Aufgabe, Systemanweisung, Ausgabeschema, Modell und Temperatur konstant. Nur die Spec wechselte. Jede vollständige Eingabe und jede abladierte Fassung lief zwanzigmal.

Der gemeinsame Aufgabenprompt lautete:

```
# Aufgabe
Entwirf eine REST-API fuer die Ressource 'User' mit
 ↪ Standard-Operationen (Liste, Anlegen, Lesen,
 ↪ Aktualisieren, Loeschen).
```

Eine Systemanweisung verlangte kein OpenAPI-Dokument und keinen Code, sondern ein JSON-Objekt mit genau fünfzehn Designentscheidungen. Dazu gehörten Basispfad, Aktualisierungsmethode, ID-Format, Schreibweise der Felder, Pagination, Fehlerformat, Authentifizierung und Versionierung. Jede Ausgabe wurde dadurch zu einem vergleichbaren

Tupel über dieselben fünfzehn Dimensionen. In der Baseline endete die Eingabe nach dem Aufgabenprompt. In den übrigen Varianten folgte eine Rubrik `# Spec` mit dem jeweiligen Begleittext. Der vollständige Prompt und die zentralen Specs stehen im Anhang.

Das Experiment ist explorativ. Es misst keine allgemeine Eigenschaft von Specs, sondern eine konkrete Kombination aus Aufgabe, Spec, Modell und Auswertung.

Zwei Achsen, nicht eine Zahl

Jede Spec erhält im Experiment zwei getrennte Koordinaten. Die Ablation Sensitivity zeigt, welche ihrer Aussagen den Output verändern. Die semantische Entropie zeigt, wie stark die Ausgaben bei unveränderter Spec streuen. Erst das Koordinatenpaar unterscheidet eine wirksame Festlegung von bloßer Stabilität oder reproduzierbarem Unsinn.

Achse 1: Anteil tragender Spans

Zuerst wird die Spec in Spans zerlegt, also in einzelne Aussageeinheiten auf der Ebene eines Satzes, Absatzes oder Listenpunkts. Für jeden Span werden Ausgaben mit der vollständigen Spec und mit einer um genau diesen Span gekürzten Fassung erzeugt. Die Ablation Sensitivity vergleicht anschließend beide empirischen Output-Verteilungen. Der Begriff Ablation stammt vom lateinischen *ablatio*, also Wegnahme:

$$\mathrm{ASS}(s) = D_{JS}\big(P(Y \mid \mathrm{Spec}) \,\|\, P(Y \mid \mathrm{Spec} \setminus s)\big)$$

D_{JS} bezeichnet die Jensen-Shannon-Divergenz, einen symmetrischen Abstand zwischen zwei Wahrscheinlichkeitsverteilungen. Bei Logarithmen zur Basis 2 liegt sie zwischen 0 und 1. Der Wert 0 bedeutet, dass in der Stichprobe keine Verschiebung sichtbar wurde. Der Wert 1 bedeutet, dass die beobachteten Verteilungen vollständig auseinanderliegen.

Für dieses Experiment gilt ein Span mit ASS $\geq 0{,}5$ als stark tragend. Ein Wert unter $0{,}05$ markiert einen Span in dieser Stichprobe als empirisch wirkungslos. Beide Schwellen sind Arbeitsdefinitionen für den Datensatz,

keine Naturkonstanten. Die erste Achse der Anteil stark tragender Spans, share(ASS $\geq 0{,}5$). So fällt eine Spec mit zwei tragenden Festlegungen und 38 Floskeln sofort auf.

Die Lehrbuch-CRUD-Spec liefert dafür das erste konkrete Bild. Ihr Titel `User REST API`, der Herkunftshinweis und die Überschrift `Endpoints` lagen zwischen 0 und 0,03. Auch die expliziten GET- und POST-Pfade veränderten die Entwürfe kaum, weil die Kombination aus REST-API und CRUD diese Entscheidungen bereits stark vorzeichnete. Anders die Vorgaben zu PUT und DELETE: Mit Werten von 0,62 und 0,50 banden sie offene Entscheidungen über Aktualisierungsmethode und Löschsemantik. Insgesamt überschritten neun der 26 Spans die Schwelle von 0,5.

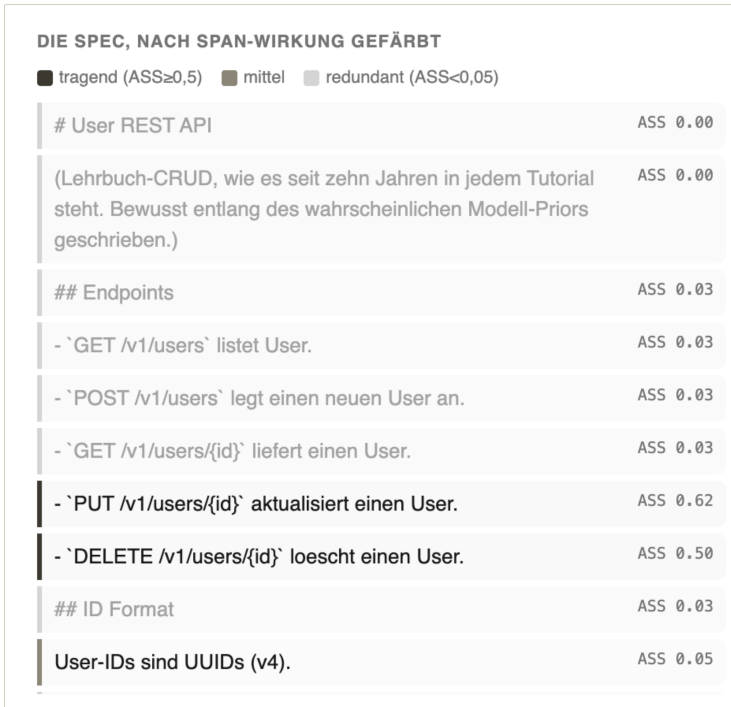


Abbildung 3.2: Die zeilenweise Ablation zeigt, welche Teile der Lehrbuch-CRUD-Spezifikation die Output-Verteilung verändern. Überschriften sowie die konventionellen GET- und POST-Endpunkte bleiben in diesem Versuch nahezu wirkungslos. Die Vorgaben zu PUT und DELETE tragen deutlich stärker, weil sie offene Designentscheidungen binden. Die vollständige Ergebnisansicht liegt unter https://ingo-eichhorst.de/arch-3/spec-quadranten.html .

Ein niedriger Wert macht einen Span nicht automatisch wertlos. Für Menschen kann er dokumentarische, rechtliche oder orientierende Bedeutung besitzen. Er hat in diesem Versuch lediglich keine zusätzliche Entscheidung des Modells gebunden. Abbildung 3.3 löst dieses Muster vom REST-Beispiel und zeigt die erste Achse als allgemeines Spektrum.

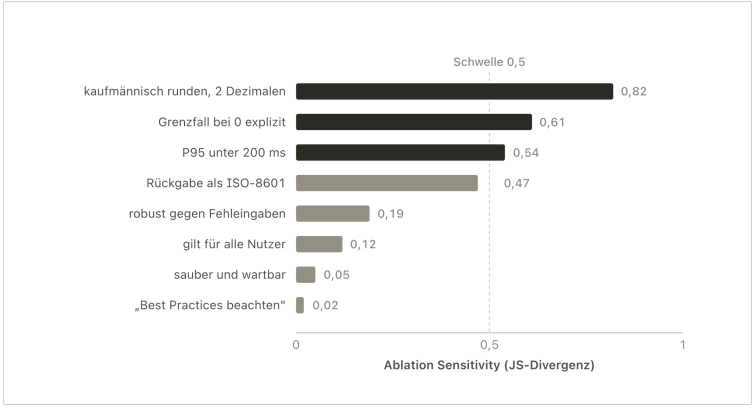


Abbildung 3.3: Jeder Span einer Spezifikation wird gestrichen und der Effekt auf die Output-Verteilung gemessen. Die Ablation Sensitivity ist null, wenn keine Verschiebung sichtbar wird, und eins, wenn die beobachteten Verteilungen vollständig auseinanderliegen. Nur wenige Spans überschreiten die hier gesetzte Schwelle für eine starke Wirkung (oben, dunkel). Niedrigere Werte bedeuten nicht automatisch, dass eine Aussage wertlos ist, aber ihre Entfernung verschiebt die beobachtete Output-Verteilung deutlich weniger. Länge ist eben nicht Klarheit. Die gezeigten Spans sind ein schematisches Beispiel; die Methode liegt als Experiment im Begleit-Repository, 03-quellenkodierung-transmitter/spec-signal offen.

Achse 2: Output-Streuung

Die zweite Achse verwendet die zwanzig Ausgaben mit vollständiger, unveränderter Spec. In Anlehnung an die semantische Entropie nach Farquhar et al. werden sie nicht nach Formulierungen, sondern nach ihren Designentscheidungen gruppiert. Zwei syntaktisch verschiedene Antworten landen im selben Cluster, wenn ihr Tupel über die fünfzehn Felder übereinstimmt. Die Entropie entsteht über die Verteilung dieser Cluster, nicht über einzelne Tokens. Das Clustering ist damit selbst Teil der Messung und muss für jede Aufgabendomäne nachvollziehbar definiert sein.

Formal bezeichnet $H_S = H_{sem}(Y \mid \text{Spec})$ die semantische Entropie mit vollständiger Spec. $H_0 = H_{sem}(Y \mid \emptyset)$ ist die modellspezifische Baseline ohne Spec. Für die zweite Koordinate wird H_S durch $\log_2 N$ normiert. $H_S = 0$ bedeutet, dass alle beobachteten Läufe im selben semantischen Cluster liegen. Ein hoher Wert bedeutet, dass sich die Läufe auf viele Designvarianten verteilen. Gilt $H_S > H_0$, streut der Output mit Spec sogar stärker als ohne sie. Das ist ein Warnsignal für Mehrdeutigkeit oder konkurrierende Vorgaben.

Eine niedrige Streuung beweist noch keine Klarheit. Ein Modell kann geschlossen auf dieselbe falsche Interpretation zulaufen. Umgekehrt kann eine Spec stark tragende Spans besitzen und trotzdem streuen. Wirkung, Reproduzierbarkeit und fachliche Richtigkeit bleiben drei verschiedene Fragen. Aus den ersten beiden entstehen die vier Quadranten des Experiments.

Vier Fälle, drei Kontraste

Die zwei Achsen spannen eine Matrix mit vier diagnostischen Feldern auf:

	geringe Output-Streuung	hohe Output-Streuung
viele tragende Spans	**Effektiv:** Die Spec trifft Entscheidungen und stabilisiert den Output.	**Widerspruch:** Aussagen wirken, ziehen den Output aber in konkurrierende Richtungen.
wenige tragende Spans	**Redundant:** Der Modell-Prior stabilisiert den Output bereits ohne die Spec.	**Theater:** Viel Text erzeugt kaum nachweisbare Wirkung.

Die Namen sind Diagnosen, keine Qualitätsurteile über das Dokument als Ganzes. Eine für Menschen wertvolle Architekturübersicht kann für eine

konkrete Generierungsaufgabe redundant sein. Gemessen wird nur ihre Wirkung auf das Zielmodell in dieser Aufgabe.

Aus der Baseline und den fünfzehn vermessenen Texten machen vier Fälle die Quadranten besonders anschaulich. Sie begleiten deshalb die weitere Argumentation.

Keine Spec. Diese Variante enthielt nur den Aufgabenprompt. Das bedeutet nicht, dass das Modell keine Anweisung erhielt. Es bekam lediglich keinen zusätzlichen Begleittext und musste alle fünfzehn Designentscheidungen aus seinem Prior ergänzen. Die Variante besitzt definitionsgemäß keine Spans und dient als Nullmessung.

Lehrbuch-CRUD. Dieser synthetische Text bündelte Konventionen aus typischen Tutorials und lag damit nahe am vermuteten Modell-Prior. Neben den bereits gezeigten Endpunkten legte er unter anderem `camelCase`, Offset-Pagination, ein JSON-Fehlerformat und Bearer-Authentifizierung fest. Neun seiner 26 Spans wirkten stark. Der Text bestätigte also nicht nur Bekanntes, sondern schloss mehrere Dimensionen, in denen die Baseline tatsächlich streute.

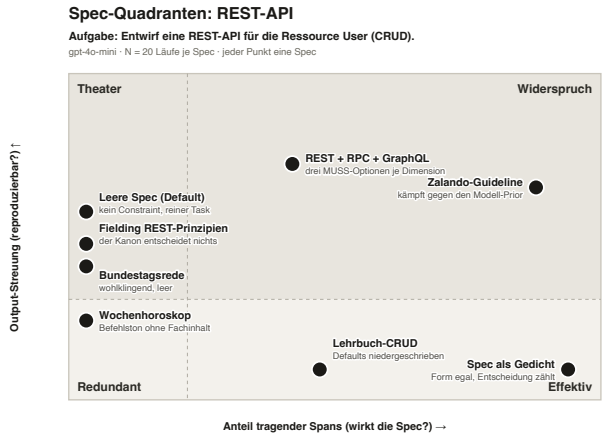
Zalando-Guidelines. Der übersetzte und gekürzte Auszug aus den öffentlichen Zalando RESTful API Guidelines setzte konkrete, teilweise priorferne Regeln. Dazu gehörten `snake_case`, der Verzicht auf URI-Versionierung, Cursor-Pagination und Problem JSON nach RFC 7807. Sechzehn von 24 Spans wirkten stark. Der Text war keineswegs vage, verlangte dem Modell aber Entscheidungen gegen bevorzugte Muster ab.

Wochenhoroskop. Die synthetische Negativkontrolle bestand aus fachfremdem Text im Ton einer Handlungsanweisung. Keiner ihrer 14 Spans enthielt eine API-Entscheidung, und keiner überschritt die Schwelle von 0,5. Diese Variante prüfte, ob bloßer Ton und zusätzlicher Kontext die Streuung verändern können, obwohl fachlich nichts übertragen wird.

Die vier Fälle ergeben folgende Koordinaten:

Eingabe	Anteil stark tragender Spans	semantische Entropie	Einordnung
keine Spec	keine Spans	H_0 zwischen 2,07 und 2,43	Referenz bei hoher Streuung
Lehrbuch-CRUD	0,35	$H_S = 0{,}29$	Effektiv
Zalando-Guidelines, Auszug	0,67	$H_S = 2{,}91$	Widerspruch
Wochenhoroskop	0,00	$H_S = 0{,}99$	Redundant

Die drei Spec-Vergleiche entstanden in getrennten Batches, deshalb schwankt ihre Baseline. Jede Spec wird mit dem H_0 ihres eigenen Batches verglichen. Für die gemeinsame Matrix werden der Anteil tragender Spans und die normierte Streuung $H_S / \log_2 N$ verwendet. Die vollständigen Daten liegen im Begleit-Repository (siehe Quellen dieses Abschnitts).



Fünfzehn Specs in zwei Achsen: Anteil tragender Spans gegen Output-Streuung

Kontrast 1: keine Spec und Lehrbuch-CRUD. Ohne Spec streuten die zwanzig Entwürfe über Basispfade, Aktualisierungsmethoden, Pagination und Fehlerformate. Die Baseline lag bei $H_0 = 2{,}38$. Die Lehrbuch-Spec

schrieb mehrere dieser Defaults fest. Rund 35 Prozent ihrer Spans veränderten den Output stark, während H_S auf 0,29 fiel. Sie landete im Quadranten Effektiv, weil sie Entscheidungen band und den Output stabilisierte.

Kontrast 2: Lehrbuch-CRUD und Zalando-Guidelines. Beide Texte enthielten konkrete Regeln. Der Zalando-Auszug besaß mit 0,67 sogar den höheren Anteil tragender Spans. Trotzdem stieg seine semantische Entropie auf 2,91 und damit über die zugehörige Baseline von 2,43. Regeln wie der Verzicht auf URI-Versionierung und verpflichtendes `snake_case` konkurrierten mit den bevorzugten Mustern `/v1/` und `camelCase`. Bei einer Temperatur von 1,0 setzte das Modell diese priorfernen Regeln nicht in jedem Lauf gleich um. Die Aussagen wirkten, zogen die Generierungen aber nicht zuverlässig in dieselbe Richtung. Das ist die Signatur des Quadranten Widerspruch.

Kontrast 3: keine Spec und Wochenhoroskop. Das Horoskop enthielt keine fachliche API-Entscheidung und erreichte bei keinem Span ASS $\geq 0{,}5$. Trotzdem fiel die semantische Entropie von einer Baseline von 2,07 auf 0,99. Möglicherweise drückte der kohärente Befehlston das Modell in einen stabileren Default. Bewiesen ist diese Erklärung nicht. Sicher ist nur, dass der Output gleichförmiger wurde, obwohl keine fachliche Information übertragen wurde. Genau deshalb landete das Horoskop im Quadranten Redundant. Die Entropie allein hätte diesen Unterschied verdeckt.

Die Variante ohne Spec markiert den Nullpunkt bei hoher Streuung und fehlenden Spans. Sie ist selbst kein Dokumentationstheater. Andere Texte, etwa Fieldings abstrakte REST-Prinzipien, landen jedoch aus demselben mathematischen Grund im Theater-Feld. Für Menschen können sie wertvoll sein, doch für diese konkrete Aufgabe entscheiden sie nicht über ID-Format, Pagination oder Fehlerformat.

> **Kurz:** Die Lehrbuch-Spec bindet Entscheidungen und stabilisiert. Die Zalando-Spec bindet Entscheidungen, streut aber gegen den Modell-Prior. Das Horoskop stabilisiert, ohne eine fachliche Entscheidung zu binden. Erst beide Achsen erklären den Unterschied.

Eine eigene Spec-Modell-Kombination vermessen

Nach der Diagnose folgt die Praxis. Nicht jede Spec muss durch Hunderte Generierungen laufen. Die vollständige Einzelablation ist der Messschieber im Labor, nicht der Zollstock für jeden Commit. Im Alltag genügt eine Eskalationsleiter. Erst wenn eine günstige Stufe keine klare Antwort liefert, folgt die nächste.

Das Messobjekt bleibt auf jeder Stufe dasselbe: eine konkrete Aufgabe, eine konkrete Spec und ein konkretes Modell unter festgelegten Bedingungen. Aufgabe, Systemanweisung und Ausgabeschema bleiben konstant. Jedes Modell erhält eine eigene Baseline ohne Spec. Nur so zeigt der Vergleich, welche Entscheidungen die Spec tatsächlich überträgt und wie reproduzierbar die Ausgaben werden.

Vier Stufen statt eines Messmonsters

Stufe	Leitfrage	Typischer Aufwand	Geeignet für
1. Lint	Trifft der Text konkrete Entscheidungen, und welche wichtigen Fragen bleiben offen?	statisch oder ein Modelllauf	jede größere Spec-Revision
2. Output-Probe	Streut das Zielmodell trotz vollständiger Spec?	ungefähr zwölf vollständige Läufe	wichtige Änderungen und Modellwechsel
3. Teilmengen-Ablation	Welche Bereiche der Spec tragen wahrscheinlich?	ungefähr 32 zufällige Teilmengen	gezielte Diagnose bei langen Specs

Stufe	Leitfrage	Typischer Aufwand	Geeignet für
4. Einzelablation	Was verändert jeder einzelne Span?	$N \cdot (m + 2)$ Generierungen	Referenzmessung und strittige Grenzfälle

Die erste Stufe ist ein Filter. `Spec-Signal` im Begleit-Repository markiert konkrete Festlegungen, Floskeln, offene Entscheidungen und mögliche Widersprüche. Im Benchmark über 28 Specs und 488 Spans sortierte sein statischer Signal-Score tragende Aussagen mit einer ROC-AUC von 0,74 besser als Zufall. Den tatsächlichen tragenden Anteil schätzte er jedoch nur mäßig gut, mit Spearman $\rho = 0{,}41$. Der Linter findet also verdächtige Stellen. Er vermisst ihre Wirkung nicht.

Die zweite Stufe beobachtet das Zielmodell direkt. Ungefähr zwölf Läufe mit der vollständigen Spec reichen im vorhandenen Datensatz aus, um die Output-Streuung brauchbar anzunähern. Diese Kurzprobe korrelierte mit der Referenz bei zwanzig Läufen mit Spearman $\rho = 0{,}94$. Sie sagt noch nicht, welcher Span wirkt. Sie beantwortet aber die dringendere Frage, ob die Spec das Modell überhaupt auf einen reproduzierbaren Pfad bringt.

Die dritte Stufe sucht diesen Pfad abschnittsweise ab. Das Verfahren `spec-ablate` im Begleit-Repository erzeugt zufällige Teilmengen der Spec und schätzt die Wirkung der Spans anschließend über eine Lasso-Regression. Das spart viele Aufrufe, lieferte über lange Specs hinweg aber nur eine ROC-AUC von 0,59 für einzelne Spans. Im Benchmark traf es den richtigen Quadranten in der Hälfte der Fälle und benötigte knapp 1.800 statt fast 10.000 Generierungen. Das ist ein nützlicher Mittelweg, aber noch keine verlässliche Wirkungskarte.

Erst die vierte Stufe entfernt jeden Span einzeln und wiederholt die Generierung. Sie bleibt die Referenz, wenn ein überraschender Quadrant erklärt, ein Modellwechsel geprüft oder eine wichtige Spec geeicht werden soll. Das vollständige How-to mit Kostenformel, Domänenkonfiguration und Auswertung steht im README von `sit` im Begleit-Repository.

Für den Prozess ergibt sich daraus eine einfache Regel: Specs werden zunächst gelintet. Bei wichtigen oder ungewöhnlichen Aufgaben folgt eine kurze Output-Probe mit dem Zielmodell. Nur wenn diese Probe stark

streut oder eine Änderung unerwartet wirkt, lohnt die teure Ablation. Über mehrere Revisionen ist der Trend oft wertvoller als ein einzelner Wert. Wächst die Spec, während ihr tragender Anteil fällt und ihre Streuung steigt, wächst vermutlich vor allem Watte.

Ein Modellwechsel verändert den Kanal

Warum muss schon die kurze Output-Probe mit dem späteren Zielmodell laufen? Weil eine Spec nicht für sich allein wirkt. Sie trifft auf das Vorwissen und die bevorzugten Lösungen eines bestimmten Modells. Was für `gpt-4o-mini` erklärt werden muss, kann für ein neueres Modell selbstverständlich sein. Umgekehrt kann eine Regel bei einem Modell stabil greifen und bei einem anderen gegen einen starken Prior kämpfen. Im Shannon-Bild wurde damit nicht der Empfänger, sondern der Kanal ausgetauscht.

MCP macht diesen Unterschied greifbar. Der dokumentierte Wissensstichtag von `gpt-4o-mini` ist der 1. Oktober 2023. Anthropic veröffentlichte das Model Context Protocol erst am 25. November 2024. Ein MCP-Auftrag muss diesem Modell daher erst erklären, dass die Kommunikation auf JSON-RPC 2.0 beruht, Fähigkeiten ausgehandelt werden und Server Resources, Prompts und Tools anbieten.

`gpt-5.2` besitzt dagegen einen dokumentierten Wissensstichtag vom 31. August 2025. MCP gehört dadurch bereits zum Modell-Prior. Eine lange Protokollerklärung trägt dann möglicherweise wenig, während eine projektspezifische Abweichung weiterhin entscheidend bleibt. Der spätere Stichtag beweist nicht, dass ein bestimmtes Dokument im Trainingsdatensatz lag. Er zeigt aber, warum Ergebnisse immer an eine Modellversion gebunden sind.

Wirkung ist noch keine Richtigkeit

Bis hierhin wurde nur der Übertragungskanal geprüft. Ein stabil reproduzierbarer Output kann die falsche Architektur ebenso zuverlässig erzeugen wie die richtige. Die beiden Achsen sagen, ob eine Spec Entscheidungen an ein Modell überträgt und dessen Ausgaben stabilisiert. Sie sagen nicht, ob diese Entscheidungen fachlich sinnvoll sind.

Damit verschiebt sich die nächste Frage. Bevor eine Absicht dicht codiert wird, muss feststehen, wohin der Output überhaupt gelenkt werden soll. Genau dafür braucht es Triangulation und ein Klärungs-Gate.

3.3 Triangulation: das fachliche Ziel festlegen

Eine wirksame Spec ist noch keine richtige Spec. Deshalb wird zuerst geklärt, welche fachliche Entscheidung gelten soll. Erst danach wird diese Entscheidung als Regel, Beispiel und Test festgehalten. Die Regel nennt die Vorgabe, das Beispiel zeigt ihre Anwendung und der Test prüft ihre Grenze. Entstehen alle drei aus einer ungeklärten Beschreibung, wiederholen sie nur dieselbe Vermutung des Modells. Die Reihenfolge ist deshalb einfach: erst entscheiden, dann mehrfach codieren.

Interaktive Klärung kann viel bewirken. Im Ambig-SWE-Benchmark stieg die Leistung je nach Aufbau um bis zu 74 Prozent. Das ist ein Benchmarkbefund, keine Garantie für jedes Repository. Das Prinzip bleibt belastbar: Fakten liest der Agent aus Repository, ADRs, APIs und Tests. Folgenreiche Entscheidungen beantwortet die verantwortliche Person oder delegiert sie bewusst. Erst wenn keine blockierende Frage offen ist, beginnt die Implementierung. So ersetzt der Modell-Prior keine Entscheidung, die das Projekt treffen muss.

Das Klärungs-Gate

Das konkrete Werkzeug kann morgen anders heißen. Sein Vertrag bleibt: Fakten suchen, Entscheidungen sichtbar machen, Beschlüsse speichern und erst dann implementieren. Ein Klärungs-Gate folgt dieser Reihenfolge:

1. Fakten im vorhandenen Projektkontext suchen.
2. Nur folgenreiche Entscheidungen als Fragen übrig lassen.
3. Die Empfehlung des Modells mit der Zielvorstellung vergleichen.
4. Bestätigte Antworten dauerhaft in Spec, ADR oder Akzeptanzkriterien schreiben.
5. Erst danach einen kurzen Design-Probelauf erzeugen.

Aktuelle Werkzeuge liefern Beispiele für diesen Vertrag. Matt Pococks `grill-me` stellt jeweils eine Frage und interviewed den Entwickler solange, bis keine Unklarheiten mehr übrig sind. GitHub Spec Kits `/speckit.clarify` sucht systematisch nach Lücken, priorisiert höchstens fünf Fragen und schreibt Antworten in die Spec zurück. `Spec-Signal` erkennt zuvor offene Designentscheidungen und legt sie zur Abarbeitung vor. Das hier vorgeschlagene Gate ergänzt zwei Zustände: `BLOCKIERT` für Entscheidungen, die vor der Implementierung fallen müssen, und `DELEGIERBAR` für bewusst freigegebene Spielräume.

Der daraus abgeleitete Prompt lautet:

```
Prüfe Aufgabe, Spec und vorhandenen Projektkontext
vor der Implementierung. Implementiere noch nichts.

Beantworte Fakten zuerst selbst aus Repository, ADRs,
APIs und Tests. Erstelle intern eine Warteschlange mit
Entscheidungen, für die mehrere plausible Antworten zu
unterschiedlichen Lösungen führen.

Stelle genau eine Frage auf einmal. Biete genau vier
Antwortmöglichkeiten an:
A bis C sind realistische Optionen.
D ist immer "Andere Antwort".

Markiere eine Option als EMPFOHLEN und begründe knapp,
warum sie zu Projektkontext und bisherigen Antworten passt.
Warte danach auf die Antwort.

Protokolliere nach jeder Antwort:
1. die getroffene Entscheidung,
2. ihre Auswirkung,
3. den Status BLOCKIERT oder DELEGIERBAR,
4. ob die empfohlene Option gewählt wurde.

Passe spätere Empfehlungen an die bisherigen Antworten an.
Zähle nur unabhängige, fachlich relevante Entscheidungen.
Beginne die Serie bei einer abweichenden Antwort oder
Korrektur erneut. Nach fünf gewählten Empfehlungen in Folge
```

```
fasse die daraus abgeleiteten Präferenzen zusammen und
lasse sie bestätigen.

Frage nicht nach lokalen und leicht reversiblen Details.
Beginne nicht mit der Implementierung, bevor das
Klärungs-Gate ausdrücklich geöffnet wurde.
```

Jede Antwort wird als Entscheidung festgehalten, nicht nur als vergängliche Chatnachricht:

Status	Frage und Auswirkung	Zuständigkeit und Beschluss
BLOCKIERT	Löscht `DELETE /users/{id}` physisch oder setzt es einen Löschmarker? Soft Delete unterstützt Audit und Wiederherstellung, verändert aber Abfragen und Datenschutzprozesse.	Product und Datenschutz, noch offen
DELEGIERBAR	Liegt die Validierung im Controller oder im Use Case? Beide Varianten sind akzeptabel, solange die Domänenregel zentral getestet wird.	implementierender Agent, freie Wahl innerhalb dieser Grenze

`BLOCKIERT` betrifft Entscheidungen mit externer oder schwer reversibler Wirkung, etwa öffentliche Schnittstellen, Datenverlust, Geldflüsse, Berechtigungen oder Compliance. `DELEGIERBAR` bedeutet nicht ungeklärt. Es bedeutet, dass die Spec einen Spielraum bewusst freigibt und seine Grenzen benennt.

Warum der Dialog den Generierungspfad verändert

Bei jeder Frage zeigt die Empfehlung, welchen Weg das Modell ohne weitere Vorgabe bevorzugen würde. Das ist keine perfekte Messung seines Priors, aber ein brauchbarer Blick auf die nächste Abzweigung. Danach gibt es drei Fälle:

1. Die Empfehlung passt. Dann ist für diese Entscheidung keine zusätzliche Korrektur nötig.
2. Eine andere Option passt besser. Die bestätigte Antwort wird als neuer Span in die Spec geschrieben und verschiebt den Pfad.
3. Keine Option passt. `Andere Antwort` zeigt, dass die Zielvorstellung außerhalb des vom Modell vorgeschlagenen Lösungsraums liegt. Gerade diese Antwort trägt oft besonders viel Information.

Die folgende Frage berücksichtigt die bisherigen Antworten. So entsteht nicht bloß eine Liste isolierter Festlegungen. Über mehrere Entscheidungen wird auch eine Richtung sichtbar, etwa „Auditierbarkeit vor Implementierungskomfort“ oder „reversible Änderungen vor minimalem Datenmodell“. Bestätigt die verantwortliche Person eine solche Zusammenfassung, wird aus einer impliziten Präferenz eine explizite Regel.

Wie viele zusätzliche Spans braucht es dafür? Genau so viele, bis ein kurzer Probelauf mit dem Zielmodell zuverlässig in der akzeptablen Zielregion landet. Diese Zielregion ist kein einzelner perfekter Entwurf. Sie umfasst alle Lösungen, die die bestätigten Entscheidungen einhalten und nur noch bewusst delegierte Freiheiten nutzen. Bei einer Standardaufgabe kann der Modell-Prior bereits passen, dann sind null korrigierende Spans nötig. Bei einer ungewöhnlichen Domäne oder bewussten Abweichungen vom Mainstream können es viele sein.

Damit wird auch der Unterschied zwischen Vibe-Coding und professioneller Entwicklung präzise. Vibe-Coding übernimmt den ersten plausiblen Pfad des Modells. Professionelle Entwicklung vergleicht diesen Pfad mit der Zielvorstellung und greift nur dort ein, wo beide auseinanderlaufen. Auch null Eingriffe können professionell sein, wenn die Übereinstimmung geprüft und die verbleibenden Entscheidungen bewusst delegiert wurden. Die Kontrolle liegt nicht in der Textmenge, sondern in der Entscheidungshoheit.

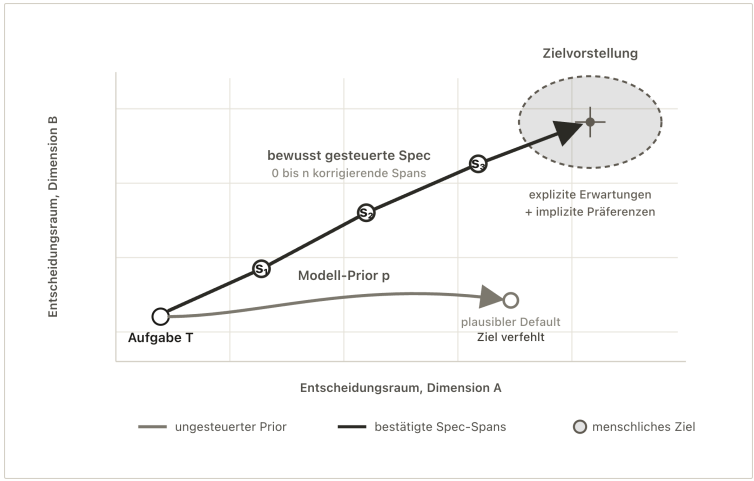


Abbildung 3.4: Ohne zusätzliche Festlegung läuft die Generierung entlang des Modell-Priors zu einem plausiblen Default. Bestätigte Entscheidungen verschieben diesen Pfad Span für Span. Die Zahl der nötigen Korrekturen ist nicht fest. Sie kann null sein, wenn Prior und Ziel bereits übereinstimmen, oder wachsen, wenn projektspezifische Erwartungen vom Mainstream abweichen. Professionelle Entwicklung erkennt man nicht an viel Text, sondern daran, dass die Zielregion bewusst festgelegt und der Pfad bis dorthin kontrolliert wird. Die Darstellung ist schematisch und zeigt keine gemessenen Koordinaten.

Warum fünf passende Empfehlungen nur ein Thermometer sind

In der Praxis kann die Faustregel von fünf akzeptierten Empfehlungen in Folge genommen werden. Sie besagt, dass die Befragung nach fünf akzeptierten Empfehlungen gestoppt werden kann. Der hintergrund ist dabei, dass die Modell-Prior scheinbar so gut getroffen wurde, dass das Modell ein ausreichendes Verständnis der Aufgabe hat. Sie sind ein billiges Signal dafür, dass Modell-Prior und Zielvorstellung in diesem Teil des Entscheidungsraums ähnlich verlaufen. Bei einer abweichenden Antwort beginnt die Serie erneut. Nach fünf Treffern darf das Modell die vermuteten Präferenzen zusammen als Spec zusammenfassen.

Das Verfahren ist bewusst nur ein Thermometer. Fünf ähnliche oder belanglose Fragen können falsche Sicherheit erzeugen. Gezählt werden deshalb nur unabhängige Entscheidungen mit erkennbarer Auswirkung. Ob das Modell den abgeleiteten Regeln später wirklich folgt, zeigt erst der Design-Probelauf.

Das Gate öffnet, wenn vier Bedingungen erfüllt sind:

1. Es gibt keine Entscheidung mehr mit dem Status `BLOCKIERT`.
2. Jeder delegierte Spielraum besitzt klare, prüfbare Grenzen.
3. Alle Beschlüsse stehen in Spec, ADR oder Akzeptanzkriterien.
4. Ein Design-Probelauf mit dem Zielmodell reproduziert die bestätigten Entscheidungen. Kritische oder priorferne Regeln werden mit mehreren Läufen geprüft.

Damit schließt sich der rote Faden dieses Kapitels. Zuerst wird sichtbar gemacht, welche Aussagen überhaupt wirken. Dann wird geklärt, welche Richtung fachlich gewollt ist. Schließlich codiert die Spec nur die Korrekturen, die das Zielmodell nicht zuverlässig aus seinem Prior ableitet. Erst danach beginnt die Implementierung.

Quellen dieses Abschnitts

- Detecting hallucinations using semantic entropy, Farquhar et al., Nature 2024. https://www.nature.com/articles/s41586-024-07421-0
- Ambig-SWE, misst, wie gut Agenten Unterspezifikation erkennen und per Rückfrage auflösen (bis zu 74 Prozent Verbesserung). arXiv:2502.13069. https://arxiv.org/abs/2502.13069
- What Prompts Don't Say: Underspecification in LLM Prompts. LLMs erschließen ungenannte Anforderungen nur teilweise zuverlässig. arXiv:2505.13360. https://arxiv.org/abs/2505.13360
- Underspecification Does Not Imply Incoherence: The Risks of Semantic Collapse in Coding Models. Geringe Streuung kann eine stabile Fehlinterpretation verdecken. arXiv:2607.01953. https://arxiv.org/abs/2607.01953
- REAgent: Requirement-Driven LLM Agents for Software Issue Resolution. Strukturierte und iterativ verfeinerte Anforderungen erhöhten die Zahl gelöster Issues gegenüber den untersuchten Baselines im Mittel um 17,40 Prozent. arXiv:2604.06861. https://arxiv.org/abs/2604.06861
- Semantic Entropy Probes. arXiv:2406.15927. https://arxiv.org/abs/2406.15927
- JoPA: Joint Prompt Attribution. arXiv:2405.20404. https://arxiv.org/abs/2405.20404

- ContextCite: Attributing Model Generation to Context, Cohen-Wang et al. arXiv:2409.00729. https://arxiv.org/abs/2409.00729
- AttriBoT: A Bag of Tricks for Efficiently Approximating Leave-One-Out Context Attribution, Liu et al., ICLR 2025. arXiv:2411.15102. https://arxiv.org/abs/2411.15102
- LLMLingua: Prompt Compression for Accelerated Inference. arXiv:2310.05736. https://arxiv.org/abs/2310.05736
- LLMLingua-2: Task-Agnostic Prompt Compression. arXiv:2403.12968. https://arxiv.org/abs/2403.12968
- FrugalPrompt: Token Attribution for Contextual Overhead. arXiv:2510.16439. https://arxiv.org/abs/2510.16439
- Demystifying Prompts via Perplexity Estimation, Gonen et al. arXiv:2212.04037. https://arxiv.org/abs/2212.04037
- Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design (FormatSpread), Sclar et al., ICLR 2024. arXiv:2310.11324. https://arxiv.org/abs/2310.11324
- Can Large Language Models Explain Themselves?, Huang et al. arXiv:2310.11207. https://arxiv.org/abs/2310.11207
- Are self-explanations from Large Language Models faithful?, Madsen et al., Findings of ACL 2024. arXiv:2401.07927. https://arxiv.org/abs/2401.07927
- Conditional Entropy Reduction for Reasoning. arXiv:2508.20395. https://arxiv.org/html/2508.20395v1
- Information Emergence (zurückgezogene ICLR-2025-Einreichung), "On the Entropy of Language Models in Getting Semantic from Tokens", Chen et al. https://openreview.net/forum?id=z3DMFpaP6m
- Small Language Model Helps Resolve Semantic Ambiguity of LLM Prompt. arXiv:2604.23263. https://arxiv.org/abs/2604.23263
- GitHub Spec Kit, `/speckit.analyze` als Spec-Lint und `/speckit.clarify` als sequenzielle Klärung mit höchstens fünf Fragen. https://github.com/github/spec-kit
- Matt Pocock, `grilling`: jeweils eine Frage mit empfohlener Antwort, bis ein gemeinsames Verständnis bestätigt ist. https://github.com/mattpocock/skills/tree/main/skills/productivity/grilling
- Matt Pocock, `grill-me`: der zustandslose, manuell gestartete Einstieg in das Grilling. https://github.com/mattpocock/skills/tree/main/skills/productivity/grill-me

- Matt Pocock, `grill-with-docs`: derselbe Klärungsdialog mit dauerhaften ADRs und Glossar. https://github.com/mattpocock/skills/tree/main/skills/engineering/grill-with-docs
- Matt Pocock, `to-spec`: überführt das bestätigte Gespräch in eine dauerhafte Spec. https://github.com/mattpocock/skills/tree/main/skills/engineering/to-spec
- Kiro, Deep Spec Analysis: Mehrdeutigkeit als semantische Entropie über Modell-Läufe. https://kiro.dev/blog/deep-spec-analysis/
- OpenAI, `gpt-4o-mini`, dokumentierter Wissensstichtag 1. Oktober 2023. https://developers.openai.com/api/docs/models/gpt-4o-mini
- OpenAI, `gpt-5.2`, dokumentierter Wissensstichtag 31. August 2025. https://developers.openai.com/api/docs/models/gpt-5.2
- Anthropic, Vorstellung des Model Context Protocol am 25. November 2024. https://www.anthropic.com/news/model-context-protocol
- Anthropic, Claude Code Best Practices, die tragend-Handregel für CLAUDE.md. https://code.claude.com/docs/en/best-practices
- Caveman, regelbasierte Token-Kompression für Coding-Agenten (Skill). https://github.com/JuliusBrussee/caveman
- Spec Information Transfer (SIT), eigene teure Referenzmessung hinter den Tabellen dieses Kapitels: semantische Entropie über N Läufe plus per-Span-Ablation, gegen die alle billigeren Schätzer geeicht sind. https://github.com/ingo-eichhorst/der-agent-tickt-anders/tree/main/03-quellenkodierung-transmitter/sit
- Spec-Signal, eigenes Skill und Benchmark zur pragmatischen Zwei-Achsen-Schätzung, gegen die teure Ablation geeicht. https://github.com/ingo-eichhorst/der-agent-tickt-anders/tree/main/03-quellenkodierung-transmitter/spec-signal
- Spec-Ablate, eigenes Skill zur Teilmengen-Ablation nach dem ContextCite-Rezept, misst beide Achsen realisiert (Streuung H_S und per-Span-Wirkung) für rund ein Sechstel der vollen Ablation, über drei Domänen und achtundzwanzig Specs gegen sie geeicht. https://github.com/ingo-eichhorst/der-agent-tickt-anders/tree/main/03-quellenkodierung-transmitter/spec-ablate

Glossar

Kurze Definitionen der wiederkehrenden Fachbegriffe, geordnet in der Reihenfolge, in der Shannons Modell sie einführt. Wer einen Begriff mitten im Buch trifft, findet ihn hier in einem Satz.

Shannon-Grundlagen

Bit. Die Maßeinheit für Information. Ein Bit entspricht der Information aus einer Entscheidung zwischen zwei gleich wahrscheinlichen Möglichkeiten.

Entropie (H). Die durchschnittliche Zahl der Bits, die eine optimale verlustfreie Codierung für die Ausgaben einer Quelle benötigt. Entropie misst Unsicherheit, nicht Umfang. Eine schwer vorhersagbare Quelle hat eine hohe Entropie.

Strukturelle Entropie. Arbeitsmodell für die Zahl untereinander widersprüchlicher, aber jeweils plausibler Implementierungswege, die ein Repository für dieselbe Aufgabe zulässt. Anders als die Semantic Entropy misst sie nicht die Streuung mehrerer Generierungen, sondern die Offenheit des Lösungsraums selbst, die klare Modulgrenzen und Architekturregeln einschränken.

Information (nach Shannon). Statistische Überraschung, nicht Bedeutung. Ein Ereignis mit Wahrscheinlichkeit eins trägt keine neue Information, wie wichtig es semantisch auch sein mag.

Kanal. Das Medium, das ein Signal überträgt und dabei durch Rauschen stört. Beim Coding-Agenten ist der Kanal die Generierung durch das Sprachmodell.

Rauschen. Eine Störung, durch die empfangenes und gesendetes Signal voneinander abweichen. Im Arbeitsmodell dieses Buches umfasst es Sampling, Mehrdeutigkeit, unvollständigen Kontext, Modellgrenzen und technische Nichtdeterminismen.

Kanalkapazität (C). Die obere Schranke der zuverlässigen Informationsrate eines mathematisch definierten Kanals. Für Coding-Agenten ist keine solche Größe gemessen. Das Buch überträgt den Begriff als Bild auf den zuverlässigen Arbeitsbereich einer Kombination aus Modell, Harness und Aufgabenklasse.

Noisy Channel Coding Theorem. Shannons Kanalkodierungssatz: Unterhalb der Kapazität lässt sich die Fehlerwahrscheinlichkeit mit geeigneten Codes und hinreichend langen Blöcken beliebig klein machen. Oberhalb der Kapazität gelingt das nicht.

Quellencodierung. Eine Quelle möglichst kompakt darstellen, indem statistische Redundanz entfernt wird. Bei verlustfreier Codierung bleibt die Nachricht vollständig rekonstruierbar.

Kanalcodierung. Kontrollierte Redundanz hinzufügen, damit der Empfänger Übertragungsfehler erkennen oder korrigieren kann. Im Arbeitsmodell dienen etwa getrennte Pläne, Beispiele und Akzeptanzkriterien als zusätzliche Spuren derselben Absicht. Zerlegung regelt dagegen vor allem die Belastung des Kanals.

Reed-Solomon-Code. Eine Familie fehlerkorrigierender Codes, die auch QR-Codes verwenden. Je nach QR-Fehlerkorrekturstufe können nominell bis zu etwa dreißig Prozent beschädigter Codewörter rekonstruiert werden. Das ist keine Garantie für denselben Anteil einer verdeckten Fläche.

Das Modell auf Coding-Agenten

Halluzinations-Lasagne. Bild für sauber formatierten, überzeugend wirkenden Agentencode, der verdeckt eine erfundene oder falsche Stelle enthält. Anders als Spaghetti-Code zeigt sie ihr Problem nicht offen, sondern tarnt es als Ordnung.

Sender (Transmitter). Der Agent samt Harness, also alles, was die Absicht in Prompt und Kontext übersetzt.

Harness. Das Gerüst aus Konventionen, Kontext und Werkzeugen rund um das Modell. “Harness Engineering” ist der Bau dieses Gerüsts.

Empfänger (Receiver). Die Prüfstraße zwischen roher Generierung und akzeptiertem Ergebnis: Parser, Compiler, Linter, Tests, Fitness Functions, Prüfagenten und Menschen.

Agent-Channel-Modell. Der Name für die im Buch verwendete Übertragung von Shannons Kommunikationsmodell auf Coding-Agenten, mit Sender, Kanal, Rauschen und Empfänger wie in diesem Glossar definiert.

Vibe-Coding. Das Gegenmodell zur Quellencodierung: sich fast vollständig auf die statistische Mitte des Modells verlassen, ohne die Absicht zu verdichten.

ADR (Architecture Decision Record). Ein kurzer, dauerhafter Vermerk über eine Architekturentscheidung samt Begründung, in der Praxis oft ergänzt um erwogene und verworfene Alternativen.

Sender-Seite (Kapitel 3)

Modell-Prior. Die aus den Trainingsdaten stammende Ausgangserwartung eines Sprachmodells, welche Muster und Lösungen es ohne zusätzlichen projektspezifischen Kontext für am wahrscheinlichsten hält. Eine Spec wirkt nur dort, wo sie diese Ausgangserwartung überschreibt.

Span. Eine einzelne Aussage-Einheit einer Spec, etwa ein Bullet-Point oder ein Satz.

Tragender oder empirisch toter Span. Ein Span trägt in einem Versuch, wenn seine Entfernung die beobachtete Outputverteilung verändert. Bleibt sie gleich, heißt er für diese Kombination aus Aufgabe, Modell und Stichprobe empirisch tot. Daraus folgt kein allgemeines Urteil über seinen Wert für Menschen.

Ablation Sensitivity (ASS). Das Maß, wie stark sich die Output-Verteilung verschiebt, wenn man einen Span aus der Spec entfernt.

Jensen-Shannon-Divergenz. Ein symmetrischer, auf null bis eins beschränkter Abstand zwischen zwei Wahrscheinlichkeitsverteilungen.

Semantic Entropy / Output-Streuung (H_S). Wie weit der Output über mehrere Läufe semantisch streut, gemessen über Bedeutungscluster statt über einzelne Token.

Spec Driven Paradox. Eine lange, vom Modell erzeugte Spec kann wenig neue Information tragen, wenn sie nur die vage Ausgangslage in geordneter Form wiederholt. Klärt sie offene Entscheidungen oder deckt Widersprüche auf, kann sie dennoch nützen.

Triangulation. Dieselbe Absicht bewusst über mehrere unabhängige Pfade senden, etwa Regel, Beispiel und Gegenbeispiel, damit sich die Restunsicherheit einengt.

Klärungs-Gate. Ein vorgelagerter Schritt vor der Implementierung, der Fakten aus dem Projektkontext sammelt, offene und folgenreiche Entscheidungen als Fragen an eine verantwortliche Person stellt und bestätigte Antworten in Spec, ADR oder Akzeptanzkriterien festhält.

Kanal-Seite (Kapitel 4)

Zuverlässiger Arbeitsbereich. Das Feld aus Aufgabenkomplexität und der Fähigkeit von Modell und Harness, innerhalb dessen ein Coding-Agent eine Aufgabe mit hinreichend geringer Fehlerwahrscheinlichkeit löst. Die praktische, empirisch beobachtete Entsprechung zur Kanalkapazität.

Taskkomplexität. Wie schwer eine Aufgabe unter einer bestimmten Modell-Harness-Kombination ist. Umfang, Änderungsstreuung, Abhängigkeiten, offene Entscheidungen, Domänenwissen und Prüfbarkeit dienen als Näherungen.

Hunk. Im Diff-Jargon ein zusammenhängender Änderungsblock. Er markiert eine einzelne Stelle, die ein Patch anfasst.

METR-Zeithorizont. Die menschliche Bearbeitungsdauer, bei der die angepasste Erfolgskurve eines Agenten eine feste Schwelle erreicht, etwa 50 oder 80 Prozent. Der Wert gilt für eine bestimmte Aufgabenverteilung und einen bestimmten Harness.

Kontextfenster. Die maximale Menge an Tokens, die ein Modell in einem Aufruf berücksichtigen kann. Die nominelle Länge garantiert nicht, dass

relevante Informationen an jeder Position und zwischen allen Ablenkern gleich zuverlässig genutzt werden.

Prompt Chaining, Routing, Evaluator-Optimizer. Drei Workflow-Muster aus Anthropics *Building Effective Agents*. Prompt Chaining verkettet feste Verarbeitungsphasen, Routing wählt je nach Fall zwischen mehreren Pfaden, und Evaluator-Optimizer lässt eine Ausgabe von einer zweiten Instanz bewerten und verbessern.

Empfänger-Seite (Kapitel 5)

Constraint-Dichte. Die Zahl prüfbarer Aussagen pro hundert Token. Zunächst eine Eigenschaft der Spec: Sie beschreibt, wie viel eindeutige Prüfinformation die Spec anbietet.

Hartes Gate, weiches Gate. Ein hartes Gate liefert bei gleichem Code deterministisch dasselbe Ergebnis, etwa Compiler, Linter oder ausführbare Tests. Ein weiches Gate beurteilt, etwa ein LLM-Reviewer oder ein Mensch, und kann sich dabei irren.

Recovery Rate (RR). Der Anteil der an einem Gate gescheiterten Läufe, die denselben Fehler nach dem erhaltenen Feedback nachweislich beheben. $RR^{(1)}$ misst den nächsten Versuch, $RR^{(\leq k)}$ die Recovery innerhalb von höchstens k Retries. Ein unabhängiges Orakel muss die Reparatur bestätigen.

False Acceptance Rate (FAR). Der Anteil grüner Gate-Urteile an einer einzelnen Prüfstation, die ein unabhängiges Orakel später verwirft.

Escape Rate. Die False Acceptance Rate am äußersten automatischen Gate der gesamten Pipeline: der Anteil automatisch akzeptierter Änderungen, die im Nachhinein doch falsch waren.

Hidden Oracle. Ein vom Reparatur-Loop getrenntes Endorakel, dessen Tests oder Regeln der Agent während der Optimierung nicht sieht. Es zeigt, ob ein sichtbarer Fortschritt auch außerhalb des optimierten Gates trägt.

Reward-Hacking-Gap. Die Differenz zwischen dem Ergebnis am sichtbaren Gate und am Hidden Oracle. Ein wachsender Gap bedeutet, dass der Loop den Proxy schneller verbessert als die tatsächliche Zielerfüllung.

Netto-Fortschritt. Die Zunahme unabhängig bestätigter Erfolge abzüglich der Regressionen einer Runde. Für eine feste Kohorte entspricht er der Änderung der Hidden-Oracle-Erfolgsquote.

Task Success Rate (TSR). Der Anteil einer festen Task-Kohorte, der nach einer definierten Pipeline-Stufe ein unabhängiges Endorakel besteht.

GateLift. Die relative Reduktion der verbleibenden Fehlerquote, die eine zusätzliche Pipeline-Stufe aus Gate und erlaubtem Retry in derselben Task-Kohorte erreicht.

Orakel. Der Prüfmaßstab, an dem sich zeigt, ob Code die Absicht trifft. Eine getrennte Herleitung von Code und Orakel reduziert geteilte Missverständnisse, garantiert aber keine Unabhängigkeit.

Intent-Drift. Eine plausible Implementierung, die intern tut, was ihre Autorin beabsichtigt hat, aber an der Spec vorbeigeht. Aus derselben Lesart erzeugte Tests können diesen Fehler übernehmen.

Toxisches Paar. Code und Test, die aus derselben fehlerhaften Annahme entstanden sind und sich deshalb gegenseitig fälschlich bestätigen, obwohl beide von der eigentlichen Absicht abweichen.

CIFE (Code Instruction-Following Evaluation). Ein Benchmark für funktionale Korrektheit und Constraint-Einhaltung. Selbst starke Modelle erreichen laut einem Preprint von Ende 2025 nur 39 bis 66 Prozent strikte Adherence.

Fitness Function. Eine ausführbare Kombination aus Metrik, architektonischem Zielbereich und einer meist automatisierten Reaktion auf Abweichungen.

Blast Radius. Die Reichweite möglicher Folgen einer Änderung. Afferente Kopplung kann ihn annähern, bildet betriebliche und fachliche Abhängigkeiten aber nicht vollständig ab.

LLM-as-Judge. Ein Sprachmodell als weicher, probabilistischer Prüfer, etwa um Schwierigkeit einzuschätzen oder Code zu reviewen.

Loop und Autonomie (Kapitel 6)

Feedback Loop. Die Schleife zwischen Sender und Empfänger, in der ein Korrektursignal zu einem neuen Versuch führt.

Single Loop Learning, Double Loop Learning. Nach Argyris und Schön wird beim Single Loop Learning ein Fehler innerhalb unveränderter Regeln korrigiert, etwa wenn ein fehlschlagender Test zu einem neuen Patch führt. Beim Double Loop Learning ändert sich stattdessen die Regel selbst, etwa wenn wiederkehrende Fehler den Harness anpassen.

PAC Learning (Probably Approximately Correct Learning). Ein lerntheoretisches Konzept, das angibt, wann aus endlich vielen Beispielen eine Regel gelernt werden kann, die mit hoher Wahrscheinlichkeit auch auf neue Fälle zutrifft. Im Buch begründet es, warum der Hypothesenraum sich selbst verbessernder Harness-Varianten begrenzt bleiben muss.

Ralph. Ein von Geoffrey Huntley benanntes, absichtlich einfaches Ausführungsmuster: eine Endlosschleife, die denselben Prompt wiederholt an einen Coding-Agenten übergibt, wobei das Repository selbst als Gedächtnis zwischen den Läufen dient. In seiner Grundform ein Produktionsmechanismus, kein lernendes System.

Convergence Slope (CS). Die Steigung eines unabhängigen Fehlerscores über die Iterationen. Ein negativer Wert zeigt Annäherung, ein positiver Verschlechterung.

Kaskadenregelung. Das regelungstechnische Muster geschachtelter Loops: Der äußere Regler setzt den Sollwert des inneren, die innere Schleife schwingt schneller ein.

Autonomie-Slider. Karpathys Bild für die Stellgröße von “nur vorschlagen” bis “vollautonom”, pro Pfad gesetzt, nicht global.

Dark Factory. Aus der Fertigung entlehnt (lights-out manufacturing): die vollautonome, sich selbst verbessernde Pipeline ohne Menschen im inneren Loop.

Cost to Resolve. Der Aufwand an Zeit, Tokens oder Kosten, den ein Agent benötigt, um von einem fehlgeschlagenen Gate bis zu einer verifiziert korrekten Lösung zu gelangen. Ein langsamer äußerer Feedback-Loop treibt diesen Wert nach oben.

agent-readyness. Das Open-Source-Werkzeug des Autors, das eine Codebasis daraufhin bewertet, wie gut sie für die Arbeit mit KI-Agenten vorbereitet ist, unter anderem nach Code-Qualität, Dokumentation, Architektur und Tests.

Ausblick (Kapitel 8)

Viabilität. Im konstruktivistischen Sinn die Brauchbarkeit einer Konstruktion für ein gesetztes Ziel unter den bisher erfahrenen Constraints. Viabel bedeutet passend, nicht endgültig wahr oder optimal.

Stressor. In der Residuality Theory eine technische, fachliche, wirtschaftliche, gesellschaftliche oder andere Einwirkung, unter der ein System betrachtet wird. Seine Wahrscheinlichkeit muss dafür nicht bekannt sein.

Residue. Das, was von einem System unter dem Einfluss eines Stressors funktionsfähig oder handlungsfähig bleibt. Mehrere Residues werden zu einer Architektur integriert, die Stress begrenzt und Optionen erhält.

Clarity Trader, Sampling-Designer, Orakel-Bauer. Drei vom Buch vorgeschlagene Rollenbilder für Softwarearchitektur mit Coding-Agenten, angelehnt an Sender, Kanal und Empfänger. Der Clarity Trader verdichtet Absicht zu einer präzisen Spezifikation, der Sampling-Designer plant unabhängige Wiederholungen und Prüfagenten ein, und der Orakel-Bauer verankert harte Prüfungen im inneren Loop.

Wie es weitergeht

Hier endet die Leseprobe. Das Denkmodell steht: Sender, Kanal, Empfänger. Angewendet ist es noch nicht.

Im vollständigen Buch folgen fünf Kapitel, die aus dem Modell ein Verfahren machen.

Kapitel 4, Kanalcodierung im verrauschten Kanal. Jeder Kanal hat eine Kapazitätsgrenze. Oberhalb davon kippt das Signal ins Rauschen, und keine noch so gute Anweisung holt es zurück. Das Kapitel zeigt, woran sich diese Grenze im eigenen Repository erkennen lässt und wie sich Aufgaben so zuschneiden, dass sie darunter bleiben.

Kapitel 5, Fehlerkorrektur am Empfänger. Der teuerste Teil des Systems ist der, der prüft. Das Kapitel baut den Empfänger aus gestapelten Gates auf, misst, welche davon tatsächlich Fehler abfangen, und zeigt an einem eigenen Experiment, wie stark die Reihenfolge der Prüfungen das Ergebnis verändert.

Kapitel 6, Feedback Loops: vom einzelnen Diff zur Dark Factory. Aus einzelnen Prüfungen werden Regelkreise. Das Kapitel ordnet die Autonomiestufen vom mitlesenden Entwickler bis zur Fabrik aus Agenten und benennt, welche Rückkopplung jede Stufe voraussetzt.

Kapitel 7, Klassische Praktiken auf Agenten übertragen. Code Review, Testpyramide, Continuous Integration, Trunk-based Development. Das Kapitel prüft, welche vertraute Praktik unter agentischer Erzeugung noch trägt, welche sich verschiebt und welche ihren Zweck verliert.

Kapitel 8, Vom Piloten zum Fluglotsen. Was die Verschiebung für die eigene Rolle bedeutet, welche drei Kompetenzen daraus folgen und welche Baupläne als Nächstes anstehen.

Dazu kommen die Anhänge mit den Messdaten und Aufgabenkohorten der Experimente sowie das vollständige Referenzverzeichnis.

Das vollständige Buch

https://leanpub.com/der-agent-tickt-anders

Der Code zu den Experimenten und die Skills aus dem Buch liegen offen im Repository:

https://github.com/ingo-eichhorst/der-agent-tickt-anders