CORPORATE LLM

HARNESS ENGINEERING

Vom seat-basierten KI-Einsatz
zu skalierbaren LLM-Architekturen

MARK ZIMMERMANN | STEPHAN KEMPF

Corporate LLM

Corporate LLM

Architektur, Governance und Harness Engineering
für die neue betriebliche Basisschicht

Mark Zimmermann & Stephan Kempf

2. Auflage · Juni 2026

Bibliografische Information der Deutschen Nationalbibliothek: Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Herstellung und Verlag:
BoD · Books on Demand GmbH, Überseering 33, 22297 Hamburg

ISBN: 978-3-6963-1170-4
2. Auflage

Inhaltsverzeichnis

Vorwort

Dieses Buch ist kein neutraler Marktüberblick und keine Schritt-für-Schritt-Anleitung. Es versteht sich als Reifegradmodell. Nicht jedes Unternehmen muss morgen den vollen 6-Ebenen-Stack implementieren, den wir hier beschreiben. Jedes Unternehmen sollte jedoch wissen, an welchem Punkt der Skalierung die heutige Seat-Logik an ihre wirtschaftlichen Grenzen stößt. Es sollte ebenso wissen, welche architektonischen Weichen dann neu gestellt werden müssen.

Wir schreiben dieses Buch, weil wir seit über zwei Jahren beobachten, wie große Organisationen in Deutschland an der gleichen Stelle stehen bleiben. Der Halt liegt nicht am Modell, nicht an der Technologie und nicht am Budget. Ihre Herausforderung liegt im Operating Model. Sie kaufen Lizenzen, richten Chatbots ein, schulen Mitarbeiter:innen im Prompt und fragen sich dann, warum die Transformation ausbleibt. Der Grund ist unbequem, doch erklärbar. Sie stellen die falsche Frage.

Die richtige Frage lautet nicht „Welches Modell sollen wir einsetzen?" oder „Wie viele Copilot-Lizenzen benötigen wir?" Sie zielt auf etwas anderes ab. Gefragt ist, wie sich die Zusammenarbeit verändern muss, wenn eine kognitive Maschine rund um die Uhr mitarbeitet.

Dieses Buch gibt dir keine Schritt-für-Schritt-Anleitung für die Einrichtung eines Chatbots. Stattdessen liefert es dir Architekturverständnis. Du lernst den 6-Ebenen-Stack kennen, der einem Corporate LLM zugrunde liegt. Du erfährst, warum die Unterscheidung zwischen Seat-basierter und Skill-basierter KI die zentrale Weichenstellung der nächsten Jahre ist. Wir führen dich durch die technischen Schichten, von Token-Budgets über Multi-Agenten-Pipelines bis zur Knowledge Distillation. Gleichzeitig arbeiten wir die organisatorischen, regulatorischen und menschlichen Dimensionen heraus.

Zwei Begriffe ziehen sich durch das ganze Buch. Ein Harness ist die Laufzeit um EINEN Agenten, also ein Modell mit Schleife, Tool-Routing,

Permissions und Recovery. Ein Agentic OS ist das Substrat für VIELE koordinierte Agenten, also die betriebliche Basisschicht des Unternehmens. Beide sind keine Alternativen, sondern verschiedene Schichten. Spätere Kapitel verweisen knapp auf diese Definition. Die empirische Schlagkraft des Harness ist 2026 belegt. Lin, Liu, Pan et al. heben einen Agenten auf Terminal-Bench 2 von 69,7 auf 77,0 Prozent. Ihr Ansatz heißt Agentic Harness Engineering.[103] Davon klar zu trennen ist die Arbeit von Lee et al. aus Stanford und MIT. Ihr Meta-Harness erreicht auf Terminal-Bench 2 einen Wert von 76,4 Prozent. Beide Zahlen meinen verschiedene Systeme und dürfen nicht vermischt werden.

Du wirst merken, dass wir durchgehend „du" sagen. Das ist Absicht. Dieses Buch spricht Menschen an, die Entscheidungen treffen, also CIOs, CDOs, Enterprise-Architekt:innen, KI-Verantwortliche und technische Führungskräfte. Die Distanz des akademischen „Sie" passt nicht zu einem Text, der dich auffordert, am Montag etwas anders zu machen als am Freitag.

Was dieses Buch nicht ist

Dieses Buch stellt ausdrücklich keine Rechtsberatung dar. Die Kapitel zum EU-AI-Act, zur DSGVO, zu NIS2, DORA und zum Betriebsverfassungsgesetz helfen dir, die architektonischen Konsequenzen regulatorischer Anforderungen zu verstehen. So sprichst du mit deiner Rechtsabteilung auf Augenhöhe. Für rechtlich verbindliche Bewertungen benötigst du qualifizierte Rechtsberatung.

Ebenso ist dieses Buch keine Produktempfehlung. Wir nennen Anbieter:innen, Modelle und Plattformen beim Namen, also Microsoft, Anthropic, OpenAI und Google. Das tun wir, weil Abstraktion hier niemandem hilft. Die Architekturprinzipien, die wir beschreiben, gelten jedoch unabhängig von der konkreten Anbieter:in. Modelle kommen und gehen. Die Frage, wie du Wissen organisierst, Agenten kontrollierst und Governance skalierst, bleibt bestehen.

Zur Einordnung

Die Meinungen, Einschätzungen und Empfehlungen in diesem Buch basieren ausschließlich auf unserer persönlichen Expertise. Sie spiegeln nicht die Position unserer Arbeitgeber:innen oder anderer Organisationen wider. Wir haben die Inhalte sorgfältig erstellt, erheben jedoch keinen Anspruch auf Vollständigkeit oder Aktualität in einem Feld, das sich schneller bewegt als jeder Druckzyklus. Kritische Entscheidungen solltest du stets mit aktuellen Quellen und Fachleuten abgleichen.

Der Stand dieser zweiten Auflage ist Juni 2026. Manche der hier beschriebenen Technologien, Lizenzmodelle und regulatorischen Rahmenbedingungen werden sich verändert haben, wenn du diese Zeilen liest. Die Architekturprinzipien werden es nicht. Token-Rechnungen verwenden eine Blended Rate von 5 €/M Token (Stand Juni 2026), wenn nicht anders angegeben.

Mark Zimmermann und Stephan Kempf (Juni 2026)

Die neue betriebliche Basisschicht

Wenn du heute als CIO, CDO oder Enterprise-Architekt:in über den Einsatz von KI in deinem Unternehmen nachdenkst, stößt du auf eine widersprüchliche Situation. Einerseits stehen Modelle, die in wenigen Sekunden Texte generieren, Code schreiben, Verträge analysieren und verschachtelte Sachverhalte zusammenfassen. Auf der anderen Seite steht eine Organisation, die diesen Fähigkeiten weitgehend ratlos gegenübersteht.

Der Grund liegt nicht in der Technologie, sondern in den internen Strukturen, die nicht darauf ausgelegt sind, KI produktiv einzusetzen. Die Statistiken bestätigen dieses Phänomen. Rund 78 Prozent der Unternehmen weltweit setzen inzwischen generative KI ein. Nur ein kleiner Teil hat jedoch eine nachhaltige betriebliche Integration jenseits von Einzellösungen aufgebaut.[11]

Dieses Kapitel stellt eine zentrale These auf. Corporate LLMs sind kein weiteres IT-Werkzeug, sondern eine neue betriebliche Basisschicht. Der Engpass für ihren produktiven Einsatz liegt nicht im Modell, sondern im Operating Model.

Was ein Corporate LLM nicht ist

Den Anfang macht eine Abgrenzung. Ein Corporate LLM ist nicht ChatGPT im Browser, kein API-Key, kein Prompt-Playground, kein Slack-Bot und kein einzelner Copilot. All das sind Oberflächenphänomene, die Zugangspunkte zu einer Fähigkeit darstellen, die erst durch eine durchdachte Architektur betrieblich nutzbar wird.

Abbildung 1.1: Was Corporate LLM nicht ist

Die meisten Unternehmen behandeln LLMs heute noch wie eine Zusatzfunktion, als Werkzeug neben dem Arbeitsplatz, um E-Mails schneller zu formulieren oder Meetings zusammenzufassen. Das ist zu kurz gegriffen. Corporate LLMs berühren gleichzeitig Wissensarbeit, Softwareentwicklung, Dokumentation, Recherche, Analyse, Kommunikation, Planung und Automatisierung. Sie berühren nicht eine Funktion, sondern alle.

Damit sind sie nicht bloß ein Werkzeug, sondern eine neue Schicht betrieblicher Arbeit. Genau so musst du sie verstehen.

Diese betriebliche Schichtung ist kein theoretisches Konstrukt. Sie ist die direkte Antwort auf eine Reifung, die sich in der KI-Technologie selbst über die letzten vier Jahre vollzogen hat. Was als reines Foundation Model begann, ist heute ein gestaffelter Stack aus Gewichten, Kontextmanagement und Wirksamkeit. Jede dieser Schichten löst ein Problem, das die darunter liegende offen gelassen hat. Modelle allein generalisieren, aber sie wissen nichts über dein Unternehmen. Kontext bringt situatives Wissen, aber er wirkt nicht von selbst. Erst Werkzeuge, Protokolle und Orchestrierung machen aus Generierung Wirksamkeit. Wer

heute über Corporate LLMs nachdenkt, denkt zwangsläufig über diese Schichten nach und über die Frage, an welcher Stelle der eigene Engpass liegt.[73][45]

Der 6-Ebenen-Stack

Das robuste Corporate-LLM-Betriebsmodell besteht aus mehreren Schichten, die zusammen den Corporate LLM Stack bilden und von der technischen Basis bis zur regulatorischen Klammer reichen. Konzeptionell ist dieser Stack ein Agentic OS, das viele Agenten persistent und koordiniert betreibt, und nicht bloß ein einzelner Harness um ein einzelnes Modell. Eine Quellcode-Analyse von Claude Code stützt diese Schichtung. Nur rund 1,6 Prozent des Codes sind Entscheidungslogik.[104] Die übrigen 98,4 Prozent bilden die operative Harness. Die Autor:innen beschreiben die Architektur als geschichteten Stack.[104] Sie sehen den Kernel-Loop samt umgebendem Agentic-OS als die eigentlich tragende Schicht. Wie weit dieser OS-Charakter reicht, zeigt Agent libOS. Es modelliert einen Agenten als schedulbaren AgentProcess.[105] Dazu gehören Identität, Capabilities, Object Memory und ein Audit. Tools werden dort zu libc-ähnlichen Wrappern. Die eigentliche Autorität liegt in den Runtime-Primitiven. Belegt wird das durch einen Python-Prototyp mit 123 Regressionstests.[105]

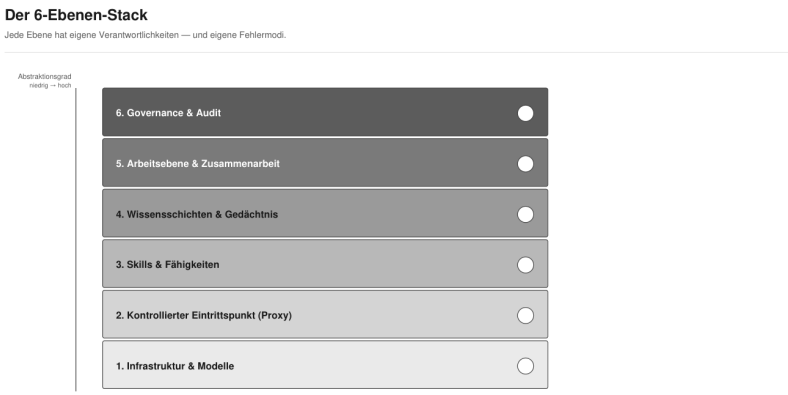

Abbildung 1.2: Der 6-Ebenen-Stack

Ganz unten liegen die Modelle als Basis. Foundation Models wie GPT-5, Claude, Gemini, Llama und europäische Open-Source-Derivate liefern die kognitive Engine mit stochastischem Sprachverständnis und Textgenerierung. Für sich genommen sind sie beeindruckend, betrieblich aber wertlos. Ein Modell ohne Kontext, ohne Regeln und ohne Ziel produziert bestenfalls plausiblen Text oder im schlimmsten Fall gefährlichen Unsinn.

Kontrolle und Betrieb machen aus einem Modellzugang einen betrieblich steuerbaren Zugang. Hier sitzen Proxy-Layer und Gateways, die den Datenfluss regulieren und Prompt-Caching durchführen. Sie sind auditierbar und bilden den Eintrittspunkt für alle generativen Anfragen. Policy Enforcement, Zugriffskontrolle, Logging, Token-Steuerung und das Routing zwischen verschiedenen Modellen geschehen auf dieser Ebene. Ohne sie ist jeder Modellzugang unkontrolliert und nicht auditierbar. Es entsteht ein Wildwuchs.

Eine eigene Ebene organisiert die Fähigkeiten und Skills, die das System inhaltlich können soll. Hier leben die Skills. Das sind modulare, versionierte Prompt-Bausteine, die spezifische Aufgaben lösen. Hier findet

auch die Definition statt, welche Werkzeuge (Tools) ein Agent aufrufen darf. Welche APIs stehen ihm zur Verfügung? Welche Protokolle, etwa das Model Context Protocol (MCP), stellen die Verbindung zwischen Modell und Außenwelt her? Skills sind keine Prompts im umgangssprachlichen Sinn. Sie sind produktive Software-Artefakte, die gebaut, getestet, überprüft und freigegeben werden müssen.

Die Wissensebene entscheidet, welches Unternehmenswissen in welcher Form für Menschen und Maschinen verfügbar ist. Rohe SharePoint-Dokumente und verstreute Confluence-Seiten reichen hier nicht. Es braucht kuratierte Wissensbibliotheken, distillierte Wissensartefakte, maschinenlesbare Arbeitsanweisungen und persistente Projektkonfigurationen. Die Frage ist nicht, ob du ausreichend Wissen hast. Die Frage ist, ob dein Wissen in einer Form vorliegt, die ein Agent verarbeiten kann.

Die Arbeitsebene ist der Ort der eigentlichen Wertschöpfung und beschreibt, wie Menschen und Systeme tatsächlich zusammenarbeiten. Dazu zählen Teamzuschnitte, Delivery-Rituale, Mensch-Agent-Kollaboration, Planungs- und Review-Workflows sowie Delegationslogik. Ein Agent, der Code schreiben kann, bleibt wertlos, wenn niemand definiert hat, wer den Review macht. Genauso wertlos bleibt er, wenn unklar ist, wie die Ergebnisse in den Entwicklungsprozess einfließen. Auch wenn offen bleibt, wer am Ende die Verantwortung trägt, taugt er nichts.

Governance und europäische Regulierung bilden die übergeordnete Klammer, die alles zusammenhält. Dazu gehören Datenschutz, EU-AI-Act, Informationssicherheit, Betriebsratsmitbestimmung, Auditierbarkeit, Nachvollziehbarkeit und Dokumentation. In Europa sind diese Anforderungen nicht optional. Sie sind die Betriebserlaubnis. Ein Corporate LLM, das diese Schicht ignoriert, ist in Europa schlicht nicht betreibbar.[1][7]

Warum das Modell nicht der Engpass ist

Die Modelle sind da, die APIs verfügbar, Hosting in Europa möglich, Sicherheit adressierbar, Kosten steuerbar. Trotzdem scheitern die meisten Organisationen an der produktiven Integration. Die Gründe liegen nicht in der Technik, sondern im Arbeitsalltag. Teams arbeiten noch in alten Mustern, Rollenbilder sind nicht aktualisiert. Führung ist oft nicht vorbereitet, Governance kommt zu grob oder zu spät. Die Wissensbasis ist nicht maschinenfähig, die Dokumentation nicht anschlussfähig. Die Delivery-Rituale stammen aus einer Vor-KI-Welt.

Die Herausforderung verschiebt sich von „Können wir das technisch?" zu „Können wir das als Organisation?". Diese Verschiebung ist der rote Faden dieses Buches.

Wir stehen mit dieser Beobachtung nicht für uns. Der Begriff Harness Engineering erfuhr 2026 mehrere parallele Aufnahmen. OpenAI machte ihn als Praxisdisziplin prominent [91], im Coding-Agent-Kontext geht er auf Böckeler und Fowler zurück. Die Disziplin baut das Gerüst um ein Modell, statt das Modell selbst zu verbessern. An dieser Stelle lohnt eine klare Begriffsdefinition, die das ganze Buch trägt. Ein Harness ist die Laufzeit um EINEN Agenten, also die Schicht, die ein Modell umschließt und zum Agenten macht. Ein Agentic OS ist demgegenüber das Substrat für VIELE koordinierte Agenten, also die betriebliche Basisschicht selbst. Der Harness löst das Problem eines einzelnen Agenten, das Agentic OS koordiniert viele davon. Spätere Kapitel verweisen nur knapp auf diese Unterscheidung. Die quantitative Bestätigung für die Harness-Ebene kommt 2026 aus der akademischen Literatur. Lin, Liu, Pan et al. arbeiten zum Agentic Harness Engineering. Sie zeigen, dass automatisierte Harness-Optimierung bei eingefrorenem Modell die Erfolgsrate auf Terminal-Bench 2 um 7,3 Prozentpunkte hebt. Sie steigt von 69,7 auf 77,0 Prozent, und zwar in zehn Iterationen ohne menschlichen Eingriff. Noch mehr sagt der Transfer-Befund. Derselbe Harness, ohne weitere Anpassung auf vier andere Basismodelle gelegt, bringt zwischen 5,1 und 10,1 Prozentpunkte Gewinn, am stärksten auf den schwächeren Modellen.

Harness-Engineering ist also produktiver und zugleich portabel. Was an einem Modell gelernt wurde, hilft auch dem nächsten.[103]

Auch Anthropic liefert eine empirische Bestätigung. Im Engineering-Beitrag „Quantifying infrastructure noise in agentic coding evals" (05.02.2026) dokumentiert das Unternehmen, dass allein die Konfiguration der Infrastruktur die Benchmark-Ergebnisse verschiebt. Gemeint ist, welche Tools verfügbar sind, wie der Kontext geschnitten wird und welche Sub-Agenten existieren. Auf Terminal-Bench 2.0 trennen 6 Prozentpunkte die beste von der schwächsten Konfiguration ($p<0{,}01$).[92] Diese Spanne ist größer als der Sprung mancher Modellgeneration zwischen ihren Versionen. Wer in einem Unternehmen über den Einsatz von KI entscheidet, optimiert deshalb das, was um das Modell herum steht. Das Modell selbst kommt später.

Die Hierarchie innerhalb des Harness ist dabei selbst messbar. Eine Komponentenablation derselben Studie zeigt, wo der Wert tatsächlich liegt. Werkzeuge tragen 3,3 Prozentpunkte, Middleware 2,2, das Langzeitgedächtnis 5,6 und der System-Prompt allein minus 2,3 Prozentpunkte. Wer Harness-Engineering auf Prompt-Tuning reduziert, trifft also genau die Komponente, die isoliert als einzige negativ wirkt. Was zählt, sind die ausführbaren Schichten, also Werkzeuge, Middleware und Memory. Diese Erkenntnis trägt durchs ganze Buch.

Die Leitunterscheidung

An dieser Stelle führen wir eine Unterscheidung ein, die das gesamte Buch durchzieht, nämlich die Leitunterscheidung zwischen Seat-basierter und Skill-basierter KI. Sie erklärt, warum manche Organisationen trotz massiver Investitionen in KI-Lizenzen kaum Wirkung erzielen. Andere verändern mit einem Bruchteil des Budgets ihre Wertschöpfung grundlegend.

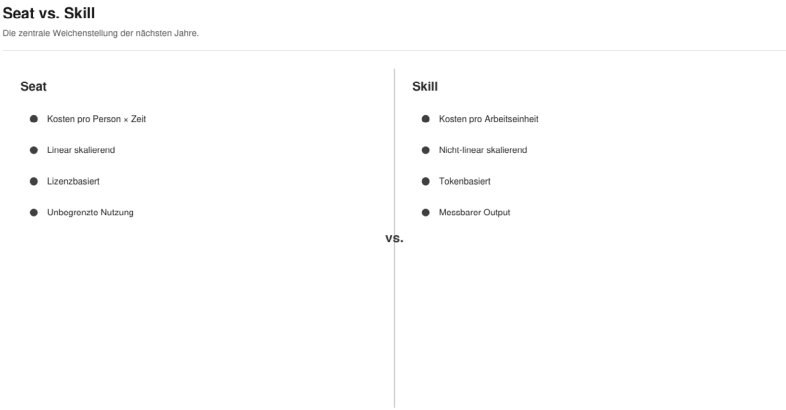

Abbildung 1.4: Seat vs. Skill

Seat-basierte KI folgt dem klassischen SaaS-Modell. Pro Mitarbeiter:in wird eine Lizenz erworben. Der Copilot sitzt neben dem Menschen, hilft bei E-Mails, fasst Meetings zusammen und schlägt Code-Ergänzungen vor. Die Produktivität des Einzelnen steigt inkrementell. Das Betriebsmodell bleibt unverändert. Es wird lediglich ein effizienterer Assistent neben den bestehenden Arbeitsplatz gestellt.

Skill-basierte KI denkt anders. Nicht der:die einzelne Mitarbeiter:in steht im Zentrum, sondern die Aufgabe. Ein Skill ist ein modulares, getestetes, versioniertes Fähigkeitsbündel, das eine bestimmte Arbeit erledigt. Das kann sein, einen Vertrag zu prüfen, eine Datenanalyse durchzuführen oder einen Sicherheitsreport zu generieren. Diesen Skill führen Agenten aus, die sich in Pipelines orchestrieren lassen, ohne dass ein Mensch jeden einzelnen Schritt auslöst. Die Kosten werden nicht pro Sitz abgerechnet, sondern pro verarbeiteten Token, pro Arbeitseinheit. Eine Feldstudie von Harvard Business School und Boston Consulting Group [31] liefert dazu konkrete Zahlen. Wissensarbeiter:innen erledigten mit GPT-4 Aufgaben rund 25 Prozent schneller und schafften etwa 12 Prozent mehr erledigte Aufgaben. Die Ergebnisqualität lag rund 40 Prozent höher. Diese Gewinne

galten jeweils innerhalb der Fähigkeitsgrenze, der sogenannten jagged frontier. Die Studie untersuchte individuelle Modellnutzung, keine skillbasierten Agenten-Architekturen. Den Sprung von dieser Einzelnutzung zur skillbasierten Architektur ziehen wir als eigene Interpretation, nicht als Befund der Studie. Auf Organisationsebene fallen die Gewinne erfahrungsgemäß niedriger aus, weil sich Skalierungseffekte und Koordinationsoverhead unterschiedlich auswirken.

Der Unterschied klingt technisch, hat aber fundamentale wirtschaftliche und organisatorische Konsequenzen. Seat-basierte KI skaliert linear mit der Mitarbeiterzahl, skill-basierte KI mit der Rechenleistung. Seat-basierte KI lässt das Organisationsdesign unangetastet, skill-basierte KI stellt es infrage. Seat-basierte KI tangiert die Modellebene und die Arbeitsebene nur oberflächlich. Skill-basierte KI erfordert dagegen die tiefe, synchrone Orchestrierung aller sechs Schichten.

Kapitel 11 wird diese Unterscheidung wirtschaftlich durchdeklinieren. Die architektonische Weichenstellung fällt jedoch bereits hier, in Kapitel 1. Sie fällt nicht auf die Ebene des Modells. Sie fällt auf die Ebene des Operating Models. Wie Brynjolfsson und McAfee in ihrer Analyse der digitalen Transformation argumentieren, markiert der Übergang zu automatisierter Wissensarbeit einen fundamentalen Umbruch. Er verändert, wie Organisationen sich strukturieren und Wert schaffen.[44]

Die Basisschicht

Bevor es tiefer in die technische Realisierung geht, musst du verstehen, woraus eine produktionsreife Agenten-Architektur besteht. Ein Agent ist kein monolithisches System. Er besteht aus sechs Abstraktionen, die eine saubere, rekursive Architektur bilden. Jede dieser Abstraktionen hat sich in tausenden realen Sessions als notwendig erwiesen, nicht weil sie theoretisch elegant ist, sondern weil sie praktisch funktioniert. Auf dieser Ebene betrachten wir einen einzelnen Agenten, also einen Harness um ein Modell. Erst die Koordination vieler solcher Agenten ergibt das Agentic OS der oberen Stack-Ebenen.

Die Abfrageschleife (Query Loop) ist das Herzstück. Sie ist ein Generator, der einen einfachen Kreis ausführt. Die Anfrage geht an das Modell. Der Generator streamt die Token ein, sammelt auftauchende Werkzeugaufrufe und führt sie aus. Die Ergebnisse gehen zur Nachrichtenhistorie, woraufhin die Schleife iteriert. Dieser Kreislauf läuft weiter, bis das Modell keine neuen Werkzeugaufrufe mehr generiert. Externe Bedingungen wie Token-Budget-Erschöpfung, maximale Iterationen, Benutzerabbruch oder Hook-Intervention können die Schleife stoppen.

Ein praktisches Beispiel macht das anschaulich. Ein:e Entwickler:in sagt dem Agenten „Refaktoriere diese Datei und schreib Tests dafür". Der Agent startet die Schleife. Er ruft „file_read" auf. Das Modell liest die Datei, erkennt das Pattern, schreibt einen Plan und ruft „file_write" für die refaktorierte Datei auf. Die Schleife iteriert. Das Modell sieht die Bestätigung, dass die Datei geschrieben wurde. Es ruft jetzt „run_tests" auf. Die Tests schlagen fehl, das Modell sieht den Fehler, refaktoriert weiter und ruft „file_write" erneut auf. Die Schleife iteriert abermals. Das Modell sieht, dass die Tests jetzt passen, und gibt keine neuen Tool-Aufrufe mehr aus. Daraufhin stoppt die Schleife und ein Report wird generiert, ohne dass ein Mensch zwischendrin eingegriffen hat. Es gibt keine separaten Werkzeug-Handling-Phasen, keine Event-Emitter und keine Callbacks. Die Schleife ist das System. Jede Interaktion fließt durch diese eine Funktion, ob interaktive REPL, API-Anfrage, Sub-Agent oder Batch-Mode.

Das Werkzeugsystem (Tool System) ist das Nervensystem des Agenten. Ein Werkzeug ist jede Aktion, die der Agent durchführen kann, etwa Datei lesen, Befehl ausführen, Code editieren oder Web durchsuchen. Ein Werkzeug ist mehr als eine Funktion. Es trägt Metadaten wie Name, Beschreibung, Eingabeschema, Ausführungslogik, Nebenläufigkeitssicherheit, Berechtigungsanforderungen und Rendering-Hinweise.

Dieser Unterschied zählt im Betrieb, weil er bestimmt, welche Aufrufe gefahrlos parallel laufen dürfen. Zwei Werkzeuge können äußerlich identisch aussehen, weil beide Input und Output haben. Eines lässt sich

jedoch ohne Warnung parallel ausführen, das andere nur sequenziell. Eines darf einen Datenbankzustand verändern, das andere nicht. Das System muss wissen, welche Werkzeuge sicher parallelisierbar sind und welche sequenziell laufen müssen. Es muss verstehen, welche Berechtigungsprüfungen notwendig sind, bevor ein Werkzeug die Welt verändert. Diese Informationen sind nicht zentral registriert, sondern im Werkzeug selbst definiert. So kann das System skalieren, ohne dass jeder neue Werkzeugtyp eine zentrale Orchestrierungsschicht verändern muss.

Ein konkretes Beispiel sind ein Datei-Lesen-Werkzeug und ein Datei-Schreiben-Werkzeug. Das Lesen kann hundertfach parallel erfolgen, und zehn Agenten können dieselbe Datei gleichzeitig lesen. Das Schreiben kann das nicht, denn nur ein Agent darf eine Datei zur gleichen Zeit verändern. Das System erkennt das an den Metadaten des Werkzeugs und verwaltet die Parallelisierung entsprechend. Wenn ein Agent in eine Datei schreiben will, die gerade von einem anderen Agenten geschrieben wird, kommt der zweite Agent in eine Warteschlange. Das ist keine globale Richtlinie, sondern im Werkzeug „file_write" definiert. Das reicht aus.

Die Aufgabenverwaltung mit Tasks und Rekursion schafft die Grundlage, damit Agenten delegieren können. Ein Task ist eine Hintergrundarbeit, meist ein Sub-Agent, die einem einfachen Zustandsautomaten mit pending, running, completed, failed oder killed folgt. Ein Sub-Agent ist keine Variante des Hauptagenten. Er ist ein neuer Durchlauf derselben Abfrageschleife mit eigener Nachrichtenhistorie und eigenem Werkzeugsatz. Rekursion ist hier kein Fehler, sondern das zentrale Architekturprinzip.

Ein Beispiel zeigt das praktisch. Ein PR-Review-Agent sieht einen Pull Request mit zehn Dateien. Er könnte alle zehn selbst prüfen, aber das wäre ineffizient, weil es zu viel Kontext bündelt. Stattdessen delegiert er. Er spawnt zehn Sub-Agenten, je einen für eine Datei. Jeder Sub-Agent bekommt exakt eine Datei, seinen eigenen System-Prompt, optimiert für den Filetype dieser Datei, und seinen eigenen Werkzeugsatz mit nur Review-relevanten Tools. Die Sub-Agenten laufen parallel. Der

Hauptagent wartet auf alle zehn, sammelt die Reviews und fügt sie zu einem Gesamt-Review zusammen. So können Agenten Aufgaben zerlegen, sie an Sub-Agenten vergeben und deren Ergebnisse integrieren, ohne dass die Kernschleife etwas davon weiß.

Die Zustandshaltung mit der State Layer besteht aus zwei getrennten Schichten. Eine Singleton-Schicht hält die Session-Infrastruktur wie Working Directory, Modellkonfiguration, Kostentracking und Telemetrie. Sie wird beim Start gesetzt und direkt mutiert. Reaktivität ist hier nicht nötig. Es braucht keine Event-Notifikationen. Eine zweite, reaktive Schicht treibt die UI an, also Nachrichtenhistorie, Berechtigungsanfragen und Fortschrittsanzeigen. Sie rendert bei jeder Änderung neu. Die Trennung ist Absicht, denn Infrastruktur-State ändert sich selten, UI-State ständig. Mit einer einzigen reaktiven Schicht zahlst du den Preis für Reaktivität auf Daten, die sich gar nicht ändern.

Das Gedächtnis (Memory) eines Agenten geht über einen statischen Prompt-Prefix hinaus. Es ist ein mehrschichtiges Dateisystem mit Projekt-Level für lokale Konfigurationen, Benutzer-Level für Präferenzen und Team-Level für Konventionen. Beim Session-Start liest der Agent diese Dateien und wählt über einen LLM-Recall-Prozess die passenden Erinnerungen. Das ist einfacher als eine Vektor-Datenbank und vertrauenswürdiger, weil Benutzer:innen genau sehen, was der Agent über sie weiß.

Ein konkretes Beispiel zeigt den Nutzen, wenn ein Code-Assistent-Agent mit einem Entwickler arbeitet. Der Agent muss vier Dinge wissen. Welche Sprachen benutzt das Projekt, welche Linting-Rules gibt es, welche Abhängigkeiten sind installiert und welche bekannten Pain-Points hat dieses Projekt? Ein einfacher Agent würde diese Informationen jedes Mal neu erfragen oder halluzinieren. Ein Agent mit Memory hat dagegen eine Datei namens project_context.md. Darin stehen beispielsweise die Sprache Python und das Framework FastAPI. Hinzu kommen die Linting-Regeln Black und Flake8 ohne Type-Checks sowie der Hinweis, dass die Dependency-Injection schwer durchschaubar ist und viele

Entwickler:innen sie nicht verstehen. Diese Datei wird gelesen. Beim Recall-Prozess prüft der Agent dann, welche Punkte für die aktuelle Aufgabe einschlägig sind. Fragen die Nutzer:innen „Schreib ein neues Modul", dann gehört der Kontext dazu. Fragen sie „Schreib Unit-Tests", ebenso. Fragen sie „Welches Datum ist heute", dann nicht. Der Agent startet mit Kontext, nicht mit leerer Tafel.

Memory-Indizes haben in produktiven Systemen feste Obergrenzen, typischerweise um zweihundert Zeilen oder fünfundzwanzig Kilobyte. Diese Grenze ist nicht weich, sondern wird beim Lesen erzwungen. Was darüber hinausgeht, fällt heraus, ohne dass das System darauf hinweist. Lange Memory-Einträge, die innerhalb der Zeilenzahl noch passen würden, geraten so trotzdem über die Byte-Grenze und nehmen kürzere, jüngere Einträge mit sich. Das Phänomen ist schwer zu bemerken, weil ein gekürzter Index immer noch wie ein vollständiger Index aussieht.

Die einzige robuste Antwort ist Disziplin auf Eintragsebene. Indexeinträge sind einzeilig, knapp, datiert und verweisen auf eine Detaildatei mit dem ausführlichen Inhalt. Mehrsatzige Zusammenfassungen, ausgeschriebene Argumentationen oder eingebettete Codebeispiele gehören nicht in den Index, sondern in die zugehörige Detaildatei. Wer das ignoriert, baut sich ein scheinbar reiches Memory-System, dessen jüngste Einträge stillschweigend verloren gehen.

Die Hooks (Erweiterungspunkte) sind benutzerdefinierte Interceptoren an 27 definierten Stellen im System. Ein Hook kann Werkzeugaufrufe blockieren, Eingaben ändern, Kontext injizieren oder die Schleife unterbrechen. Hooks sind keine Plugins, sondern externe Prozesse, die JSON über stdin und stdout austauschen. Prozess-Isolation ist hier ein Feature. Ein fehlerhafter Hook crasht seinen eigenen Prozess, nicht den Host. Ein Unternehmen kann Policy-Enforcement deployen, ohne Entwickler-Sessions zu gefährden.

Diese sechs Abstraktionen bilden zusammen die Basisschicht eines produktionsreifen Agentensystems, also die Schleife, die Werkzeuge, die Tasks, der State, das Gedächtnis und die Hooks. Sie sind unverzichtbar.

Ohne sie hast du ein sprachliches Modell. Mit ihnen hast du einen Agenten, der als betriebliche Infrastruktur funktioniert.

Bei der Architektur unserer ersten Skill-Pipeline zeigte sich, dass diese sechs Ebenen nicht optional sind. Als wir ohne Memory-Layer starten wollten, verloren wir den Kontext zwischen den Agent-Schritten. Nach der Implementierung einer dateibasierten Memory sank die Fehlerrate spürbar.

Warum ist dieser Unterschied so groß? Ohne diese Abstraktionen baust du einen Chatbot, bei dem die Nutzer:innen Input geben und der Agent Output generiert. Mit diesen Abstraktionen baust du ein Arbeitssystem. Der Agent handelt in der Welt über Query Loop und Tools. Er beauftragt andere Agenten über Tasks, hält Zustand in der State Layer, erinnert sich über Memory und wird über Hooks überwacht. Das ist keine Produktivitäts-Verbesserung, sondern eine fundamentale Architektur-Änderung.

Fünf Architektur-Wetten

Theoretisch gibt es viele Wege, diese sechs Abstraktionen zu implementieren. In der Praxis haben sich fünf Designentscheidungen als robust erwiesen, nicht weil sie theoretisch optimal sind, sondern weil sie sich unter Last bewährt haben.

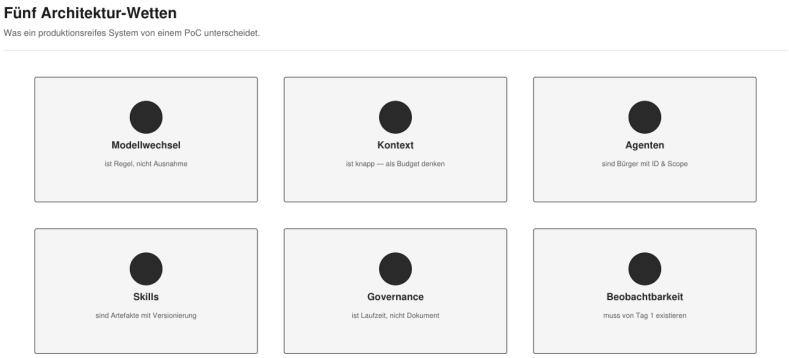

Abbildung 1.6: Fünf Architekturwetten

Eine bewährte Architekturentscheidung setzt auf eine Generator-Schleife statt Callbacks. Viele Agent-Frameworks geben dir eine Pipeline. Du definierst Werkzeuge, registrierst Händler und lässt das Framework orchestrieren. Die Entwickler:innen schreiben Callbacks, das Framework entscheidet, wann es sie aufruft. Eine produktionsreife Architektur kehrt das um, denn die Abfrageschleife gehört dem System, nicht dem Framework. Wenn ein Framework eine Callback-Architektur hat, passiert etwas Subtiles. Die Entwickler:innen schreiben „Wenn Tool X aufgerufen wird, rufe Handler Y auf". Das Framework registriert das, danach iteriert das Modell. Es ruft Tool X auf, das Framework sieht das, lädt Handler Y und führt ihn aus. Was passiert, wenn Handler Y fehlschlägt? Gibt es einen Retry-Mechanismus? Wer entscheidet, dass der Agent nun abbrechen soll? Das Framework hat Defaults, die zentral sind. Sie stecken nicht in deinem Code. Wenn die Defaults nicht auf deine Situation passen, musst du das ganze Framework neu konfigurieren. Eine Generator-Schleife löst das anders. Das Modell streamt eine Antwort, die Schleife erkennt Werkzeugaufrufe, führt sie aus, hängt Ergebnisse an die Historie an und iteriert. Ein einziger Datenfluss, ein einziger Ort, an dem jede Interaktion passiert. Du siehst genau, was passiert. Das Rückgabeobjekt kodiert dabei

jeden möglichen Endzustand, also normale Fertigstellung, Benutzerabbruch, Token-Budget-Erschöpfung, Hook-Intervention, maximale Durchläufe und nicht behebbarer Fehler. Diese Struktur ist strikter, typsicherer und leichter zu debuggen als verteilte Callback-Graphen.

Ebenso grundlegend ist dateibasiertes Gedächtnis statt Datenbanken. Datenbanken liefern reicher abfragbare Inhalte, schnellere Lookups und transaktionale Sicherheit. Dateien verzichten darauf. Sie gewinnen dafür Transparenz und Kontrolle. Benutzer:innen, die die Gedächtnisdatei ihres Agenten in einem Texteditor öffnen, sehen genau, was der Agent über sie weiß. Sie haben damit ein fundamental anderes Vertrauensverhältnis als jene, die nur fragen können „was weißt du über mich?" und auf Vollständigkeit hoffen müssen. Dateien sind maschinenlesbar, versionierbar via Git und auditierbar. Der LLM-gesteuerte Recall-Prozess ist eine kleine Anfrage an das Modell, das fünf passende Erinnerungen aus einem Manifest auswählt. Er kompensiert die Speicherungs-Einfachheit mit Abruf-Intelligenz.

Auf der Werkzeug-Ebene hat sich ein anderer Ansatz durchgesetzt, nämlich selbstbeschreibende Werkzeuge statt zentraler Orchestrierung. Agent-Frameworks definieren typischerweise ein Werkzeug-Register. Du beschreibst deine Werkzeuge zentral, das Framework präsentiert sie dem Modell. Eine produktionsreife Architektur macht das Gegenteil. Jedes Werkzeug trägt seine Metadaten selbst. Ein Werkzeug-Objekt enthält seinen Namen, seine Beschreibung, sein Input-Schema, seine Ausführungslogik, seine Nebenläufigkeits-Flags und seine Berechtigungsanforderungen. Das Werkzeugsystem braucht die Werkzeuge nicht dem Modell zu erklären. Die Werkzeuge erklären sich selbst. Das skaliert besser. Ein neues Werkzeug hinzuzufügen erfordert null Änderungen am Orchestrierer. Ein Standard-Werkzeugprotokoll wie das Model Context Protocol wird so zum gleichberechtigten Bürger. Ein MCP-Werkzeug und ein eingebautes Werkzeug sind nach dem Wrapping ununterscheidbar.

Wirtschaftlich besonders wertvoll ist Prefix-Sharing zwischen Sub-Agenten, weil es die Kosten pro gespawntem Agenten drastisch senkt. Wenn du einen Sub-Agenten spawnst, könntest du ihm die gesamte bisherige Konversation neu prozessieren lassen. Das ist kostspielig. Alternativ kannst du ihm nur eine Zusammenfassung geben. Das ist schnell, verliert jedoch Kontext. Die dritte Option setzt anders an. Der Sub-Agent startet mit der vollständigen Konversation des Eltern-Agenten, aber er teilt den Prompt-Cache der Eltern. Die ersten Tokens, der Prefix, sind bereits im Cache und kostenlos. Der Sub-Agent bekommt damit neunzig Prozent Rabatt auf die Input-Tokens, was das Spawnen von Agenten für kleine Aufgaben wirtschaftlich macht. Das ist kein Performance-Hack. Es ist eine architektonische Wette, dass diese Cache-Sharing-Komplexität sich lohnt.

Für Erweiterbarkeit haben sich Hooks statt Plugin-Systeme bewährt, insbesondere mit Prozess-Isolation. Extensibility über Plugins bedeutet, dass Code in den Host-Prozess lädt, sich dessen Speicher teilt und ihn crashen kann. Extensibility über Hooks bedeutet, dass externe Prozesse an Lifecycle-Punkten laufen, über stdin, stdout und Exit-Codes kommunizieren und isoliert crashen. Der Overhead ist real, also einige Millisekunden für Prozess-Spawning, die ein In-Process-Callback nicht bräuchte. Der Gewinn ist Sicherheit. Ein fehlerhafter Hook-Prozess bricht nicht die Host-Session. Ein Unternehmen kann Policy-Enforcement-Hooks deployen, ohne das Risiko, dass ein falsch geschriebener Hook die Entwickler-Sessions zum Absturz bringt. Für externe und unvertraute Extensions ist Prozess-Isolation nicht optional.

Diese fünf Wetten gelten für produktive Systeme. Sie zeigen, wo Komplexität sich auszahlt.

Das Designprinzip

Eine tiefere Struktur verbindet alles. Verschiebe Komplexität an die Ränder, halte den Kern sauber.

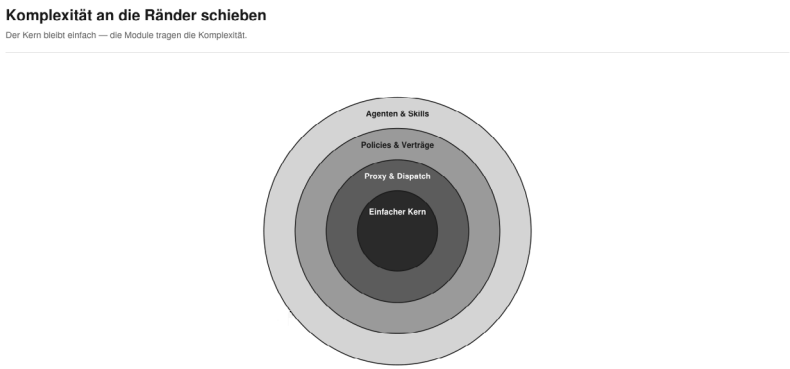

Abbildung 1.7: Komplexität an die Ränder

Die Ränder eines Agentensystems sind unordentlich. Terminal-Protokolle unterscheiden sich je nach System, OAuth-Server sind fragil, Werkzeuge vielfältig und MCP-Transporte variantenreich. Ein produktionsreifes System absorbiert diese Unordnung an den Rändern und exportiert Sauberkeit nach innen. Terminal-Bytes werden zu typisierten Events, OAuth-Token zu Benutzer-Objekten und MCP-JSON zu Werkzeug-Objekten. An den Rändern herrscht Chaos. Im Kern herrschen Typsicherheit, Klarheit und vollständige Kontrolle.

Die Abfrageschleife braucht nicht zu unterscheiden, ob Input von SSH oder Web-IDE kam. Sie verarbeitet typisierte Events. Das Werkzeugsystem unterscheidet nicht zwischen eingebauten und MCP-Werkzeugen. Beides ist ein Tool-Objekt. Das Gedächtnis-System unterscheidet nicht zwischen Git und Datenbank. Es arbeitet mit Dateien und LLM-Recall.

Die Lektion ist klar. Definiere deine Ränder, absorbiere dort die Komplexität und halte den Kern sauber. An den Rändern läuft das System-Engineering, im Kern läuft das Engineering angenehm. Konzentriere die Komplexität auf die Grenzen.

Cloud-agnostische Prinzipien

Ein Hinweis zur Einordnung gehört hierher. Dieses Buch verwendet an vielen Stellen Microsoft-Produkte als Referenz (E3/E5/E7, Agent 365, Entra Agent ID), weil Microsoft 2026 die sichtbare Enterprise-Infrastruktur für agentische KI liefert. Die Architekturprinzipien, die dieses Buch beschreibt, sind jedoch nicht an eine:n Anbieter:in gebunden. Der 6-Ebenen-Stack funktioniert mit Azure, AWS Bedrock, Google Cloud oder einem selbst betriebenen Open-Source-Stack auf Basis von Llama oder Mistral. Ein Proxy-Layer lässt sich vor jede API stellen. Skill Governance benötigt Git und ein Review-Verfahren, keine:n bestimmten Cloud-Anbieter:in. Audit-Trails sind ein kryptografisches Prinzip, keine Microsoft-Funktion. Wo die Beispiele anbieterspezifisch sind, gilt das Prinzip dahinter universell.

Werkzeug vs. Agent

Bevor du organisatorische Entscheidungen triffst, musst du eine fundamentale Differenzierung verstehen. Sie bestimmt gänzlich unterschiedliche Betriebsmodelle.

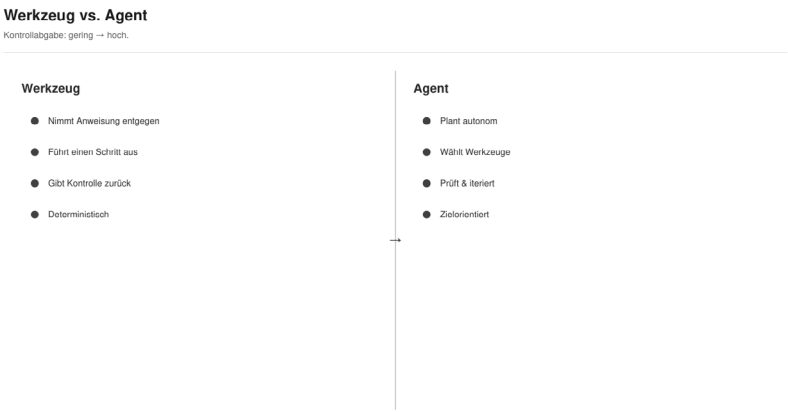

Abbildung 1.9: Werkzeug vs. Agent