

Martin Richardt

Agentenbasierte KI-unterstützte Softwareentwicklung

Impressum

Agentenbasierte KI-unterstützte Softwareentwicklung

Von Large Language Models zu autonomen Entwicklungsagenten – inklusive Model Context Protocol (MCP) und moderner Kontext-Engineering

© 2026 Martin Richardt
Alle Rechte vorbehalten.

1. Auflage 2026
Autor: Martin Richardt
Selbstverlag

Dieses Werk ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

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

Lizenzen

Manuskript, Anhang, meta/, bilder/: CC BY-NC-SA 4.0
Code unter code/, Build-Tooling: MIT

Herstellung

Umschlaggestaltung: Martin Richardt (mit KI-Unterstützung)
Satz und Layout: Martin Richardt
Erstellt mit Unterstützung künstlicher Intelligenz

Das Manuskript entstand mit KI-Unterstützung (Entwurf, stilistische Ausarbeitung). Die fachliche und redaktionelle Verantwortung für den Text liegt beim Autor.

Haftungsausschluss

Trotz sorgfältiger inhaltlicher Kontrolle übernimmt der Autor keine Haftung für die Richtigkeit, Vollständigkeit und Aktualität der Inhalte. Das Buch ersetzt keine fachliche Beratung (u. a. Medizin, Recht, Finanzen).

Inhaltsverzeichnis

Vorwort 1

Die neue Architektur der Softwareentwicklung: Warum MCP und agentenbasiertes Kontext-Management alles verändert 1

Vom Prompt zum Protokoll 1

Was dieses Buch anders macht 2

Für wen dieses Buch geschrieben ist – und für wen nicht 2

Wie du dieses Buch liest 2

Ein Wort zur Geschwindigkeit der Specs – und was stabil bleibt 3

Lesevereinbarung 3

Einleitung 5

Von Chatbots zu autonomen Software-Agenten – Die Rolle von MCP als universelle Kontext- und Tool-Schicht 5

1. Drei Evolutionsstufen – und warum die dritte alles ändert 5

2. Der eigentliche Engpass: Kontext, nicht nur Parameterzahl 6

3. Das N×M-Problem klassischer Integrationen 7

4. MCP in einem Absatz – und warum wir es vertiefen 7

5. Context Engineering: die Disziplin hinter dem Prompt 8

6. Was du in den vier Teilen des Buches erwartest 9

7. Voraussetzungen und Arbeitsweise 9

8. Leitmotive, die sich durch das Buch ziehen 9

9. Wie dieses Buch mit Code und Tools umgeht 10

Key Takeaways der Einleitung 10

Was als Nächstes kommt 10

Kapitel 1: Die Evolution der Sprachmodelle 11

Lernziele 11

1.1 Von n-Grammen zu Transformern 11

1.2 Die Skalierungsära: Pretraining, Emergenz, Alignment 12

1.3 Von Completion zu Chat zu Tools 13

1.4 Open-Source vs. Closed Models – und der Aufstieg standardisierter Kontext-Protokolle 15

Praxisbeispiel: Derselbe Use Case, drei Generationen 16

Best Practices 17

Pitfalls 17

Key Takeaways 18

Was als Nächstes kommt 18

Vorwort

Die neue Architektur der Softwareentwicklung: Warum MCP und agentenbasiertes Kontext-Management alles verändert

Wenn du in den letzten Jahren Software gebaut hast, hast du die KI-Welle wahrscheinlich in mehreren Schüben erlebt. Zuerst war da die verblüffende Autovervollständigung im Editor – nützlich, aber noch kein Systemwechsel. Dann kamen Chat-Oberflächen, in die man Stacktraces und Codefragmente kopierte. Danach Function Calling und „Tools“: Das Modell durfte plötzlich APIs anstoßen, Dateien lesen, Tests starten. Und irgendwann stand die Frage im Raum, die dieses Buch beantwortet:

Was unterscheidet einen cleveren Chatbot von einem Software-Agenten, dem du echte Entwicklungsarbeit anvertraust – und welche Infrastruktur braucht dieser Agent wirklich?

Die kurze Antwort: Es ist weniger das „Intelligenz-Level“ des Foundation Models allein. Es ist die **Architektur um das Modell herum** – vor allem, **wie Kontext bereitgestellt, ausgewählt, aktualisiert und mit Werkzeugen verbunden wird**.

Vom Prompt zum Protokoll

Lange Zeit haben wir Kontext behandelt wie eine Einmalzahlung an das Modell: alles Wichtige in den Prompt, hoffen auf das Beste, Kontextfenster maximieren. Das funktioniert für Demos. In der Softwareentwicklung scheitert es systematisch:

- Codebasen sind größer als jedes Context Window.
- Wahrheit ändert sich sekündlich: Branch, CI-Status, Issue-Board, laufende Tests.
- Aktionen sind riskant: falsches `git push`, falsches Schema-Migration, falsches Produktions-Secret.
- Integrationen explodieren kombinatorisch: jedes Agenten-Framework × jedes Ticket-System × jede IDE.

Genau hier setzt der Bruch der Jahre 2024 bis 2026 an. Mit dem **Model Context Protocol (MCP)** hat sich – ausgehend von einer offenen Spezifikation und einem rasch wachsenden Ökosystem – etwas etabliert, das Praktiker längst gebraucht haben: eine **standardisierte Schicht zwischen AI-Anwendungen (Hosts) und externen Fähigkeiten (Servern)**.

Die oft zitierte Analogie ist treffend: **MCP ist USB-C für AI**. Nicht weil es „magisch intelligent“ wäre, sondern weil es **Entdecken, Verbinden und Austauschen von Kontext und Tools** vereinheitlicht. Statt N Hosts mal M Eigenintegrationen bekommst du eine gemeinsame Sprache für:

- **Tools** – Aktionen, die das Modell ausführen darf,
- **Resources** – Daten und Kontext, die gelesen werden können,
- **Prompts** – wiederverwendbare Vorlagen und Workflows.

Gleichzeitig reift eine zweite Disziplin, die viele Teams noch unterschätzen: **Context Engineering**. Sie fragt nicht nur „Welches Modell?“, sondern:

> Welcher Kontext wird *wann*, in *welcher Form*, mit *welcher Autorisierung* und *welcher Qualitätsgarantie* in den Agenten geladen – und was bleibt bewusst draußen?

Dieses Buch behandelt MCP und Context Engineering nicht als Bonuskapitel am Ende, sondern als **zentrale Architektur der agentenbasierten Softwareentwicklung**.

Was dieses Buch anders macht

Es gibt exzellente Einführungen in Prompting, RAG und Agent-Frameworks. Was oft fehlt, ist die **systemische Sicht für Software-Teams**:

1. **Vom LLM zum Agenten mit Produktionsanspruch** – nicht nur Toy-Demos, sondern SDLC, Reviews, CI, Governance.
2. **MCP als First-Class-Thema** – Architektur (Host–Client–Server), Primitives, lokale vs. remote Server, Security, Hands-on-Bau.
3. **Context Engineering als Handwerk** – on-demand Loading, Selective Exposure, Memory-Server, weniger Context-Bloat.
4. **Deutsche Fachsprache für eine internationale Tech-Realität** – klare Begriffe, ohne die etablierten englischen Termini zu verbiegen.

Du wirst Theorie bekommen, wo sie Entscheidungen verbessert – und Praxis, wo sie dich handlungsfähig macht. Besonders **Kapitel 10** ist als Werkstatt gedacht: eigener Coding Agent, eigene MCP-Server, Integration von Git, Issues, Tests und Memory.

Für wen dieses Buch geschrieben ist – und für wen nicht

Es ist für dich, wenn du:

- Agenten in echte Entwicklungsworkflows bringen willst,
- verstehst, dass „einfach GPT anbinden“ keine Architektur ist,
- MCP-Server bauen, bewerten oder absichern musst,
- als Tech-Lead Verantwortlichkeiten, Risiken und Skills im Team klären willst.

Es ist weniger für dich, wenn du:

- eine reine ML-Forschungsmonografie zu Transformer-Varianten suchst,
- No-Code-Marketing-Automatisierung ohne Engineering-Tiefe willst,
- oder erwartest, dass ein einzelnes Modell „die Entwickler ersetzt“, ohne Prozesse, Tests und Verantwortung.

Agenten **verstärken** gute Engineering-Kultur. Schlechte Prozesse und fehlende Kontextqualität multiplizieren sie nur.

Wie du dieses Buch liest

Vier pragmatische Pfade:

Pfad	Fokus	Empfohlene Reihenfolge
Fundament	Verstehen, wie LLMs und Kontext wirklich funktionieren	Vorwort → Einleitung → Kap. 1–3
Architekt	Agenten, MCP, Multi-Agent, Frameworks	Einleitung → Kap. 2.6 → Kap. 4–6
Builder	Schnell in die Praxis und zum eigenen Server	Einleitung → Kap. 2.6 → Kap. 4 → Kap. 9–10
Lead	Risiken, Rollen, Branchenwirkung	Einleitung → Kap. 7–8 → Kap. 11–13

Du kannst springen. Wo Vorwissen nötig ist, verweisen die Kapitel zurück – vor allem auf die MCP-Grundlagen in Kapitel 2 und 4.

Ein Wort zur Geschwindigkeit der Specs – und was stabil bleibt

MCP, Host-Implementierungen (Cursor, Claude Code, Windsurf, Continue und andere), Frameworks und Sicherheitsempfehlungen ändern sich weiter. Dieses Buch ist bewusst als **Architektur- und Musterbuch** geschrieben: Es priorisiert Konzepte, die über einzelne SDK- und Produktversionen hinaus tragen.

Drei Schichten der Haltbarkeit:

Schicht	Was du mitnimmst
Stabil	Host / Client / Server; Tools, Resources, Prompts; Capability Discovery und Lifecycle; Trennung Data Layer (JSON-RPC) und Transport (STDIO, Streamable HTTP); Context-Engineering- und Harness-Prinzipien
Mittelstabil	Entscheidungsleitfäden, Server-Schnitt, Verify-Loops, Hybrid-Knowledge-Base, Role Policies
Volatil	Konkrete Dekoratoren, Host-JSON, Framework-Node-APIs – im Text als *Beispielimplementierung (Stand Juli 2026, …)* gekennzeichnet

Wo APIs flüchtig sind, sind Beispiele **Illustrationen**, keine Langzeit-Referenz. Normative Spezifikationsquelle: **MCP Specification 2025-11-25** unter modelcontextprotocol.io/specification/2025-11-25 (Lifecycle, Primitives, Transports).

> Die im Buch gezeigten Codebeispiele spiegeln den Stand **Juli 2026** wider. Das **Companion** ist das Verzeichnis `code/` in diesem Repository (Version-Tags `book-2026-07` und `latest`; siehe `code/COMPATIBILITY.md`, `code/README.md`).

Lesevereinbarung

Ich setze voraus, dass du **mitdenkst und mitbaust**: Experimente im Terminal, Skepsis gegenüber Demo-Prompts, Respekt vor Produktionsdaten. Im Gegenzug bekommst du kein Hype-Marketing, sondern eine ehrliche Landkarte – inklusive Grenzen, Failure Modes und Security-Fallen.

Das Buch ist als Ganzes lesbar – von den Grundlagen über Harness und Projekt-Knowledge-Base bis zum Hands-on in Kapitel 10. Springe gezielt (Lesepfade im README), und baue mit: Terminal, Skepsis, Respekt vor Produktionsdaten.

Viel Erfolg – und vor allem: **bau Systeme, in denen Kontext und Werkzeuge so sauber spezifiziert sind wie deine APIs.**

Der Autor wünscht dir produktive Agenten, schlanke Context Windows und MCP-Server, denen du – mit den richtigen Guardrails – vertrauen kannst.

Einleitung

Von Chatbots zu autonomen Software-Agenten – Die Rolle von MCP als universelle Kontext- und Tool-Schicht

Lernziele der Einleitung

Nach dieser Einleitung kannst du:

- die Entwicklung von Chatbots über Tool-Assistenten zu Software-Agenten einordnen,
- erklären, warum **Kontext** der zentrale Engpass moderner KI-Systeme in der Softwareentwicklung ist,
- das **N×M-Integrationsproblem** und die Rolle von MCP skizzieren,
- **Context Engineering** als Disziplin von reinem Prompting und reinem RAG abgrenzen,
- die Struktur des Buches und deinen Lesepfad wählen.

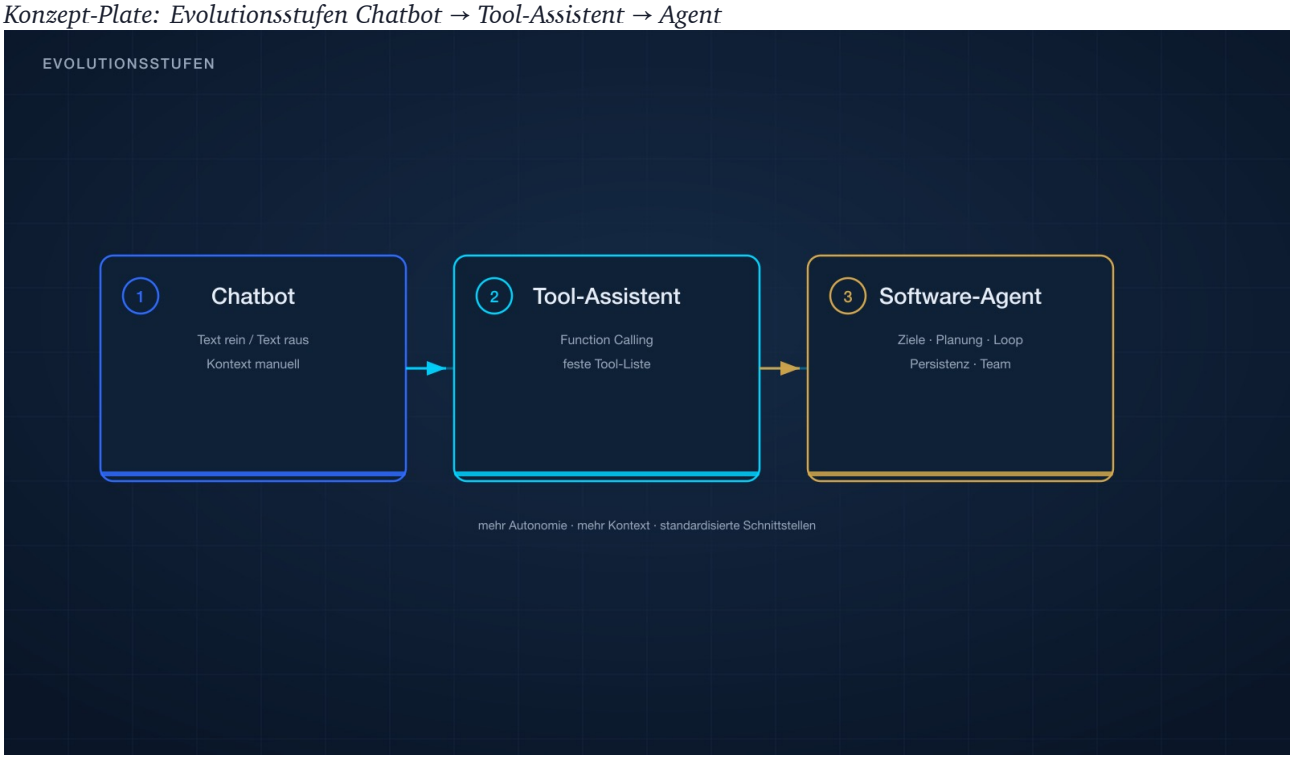
---

1. Drei Evolutionsstufen – und warum die dritte alles ändert

Stelle dir vor, du stellst einem Junior-Entwickler am ersten Tag eine Aufgabe: „Fix den Flaky-Test in `payments`." Ein Mensch fragt nach Logs, liest den letzten PR, checkt CI, reproduziert lokal. Ein naiver Chatbot bekommt denselben Satz – und rät.

Zwischen beiden liegt nicht nur „mehr Intelligenz", sondern **Zugang zu Weltzustand und Handlungsfähigkeit**. Genau diese Dimension trennt die Evolutionsstufen der KI in der Softwareentwicklung:

Konzept-Plate: Evolutionsstufen Chatbot → Tool-Assistent → Agent



Stufe 1: Der Chatbot

Du kopierst Fehlertexte, Dateiausschnitte, manchmal ganze Module. Das Modell antwortet mit Vorschlägen. Wertvoll für Brainstorming und Erklärungen – aber **du** bist das Integrationslayer: Clipboard, Kontextpflege, Ausführung, Verifikation.

Stufe 2: Der Tool-Assistent

Function Calling und IDE-Copilots erlauben dem Modell, *deklarierte* Funktionen aufzurufen: Datei lesen, Suche, Lint, manchmal Terminal. Das ist ein Quantensprung – und gleichzeitig eine Sackgasse, wenn jede Anwendung ihre eigenen Tool-Schemas, Auth-Flows und Discovery-Mechanismen bastelt.

Stufe 3: Der Software-Agent

Ein Agent verfolgt ein **Ziel** über mehrere Schritte: planen, handeln, beobachten, korrigieren. Er arbeitet mit **wechselndem Kontext** (Repo-Stand, Issues, CI), koordiniert ggf. mit anderen Agenten und braucht eine **stabile Schnittstelle zur Außenwelt**. Genau hier werden Protokolle und Context Engineering unverzichtbar – nicht optional.

> **Analogie:** Stufe 1 ist ein Berater ohne Internetzugang. Stufe 2 ist ein Berater mit drei genehmigten Telefonnummern auf dem Zettel. Stufe 3 ist ein Mitarbeiter mit Badge, Ticket-System, CI-Zugang und der Pflicht, Änderungen nachvollziehbar zu machen – idealerweise über standardisierte Schnittstellen statt Privatabsprachen.

---

2. Der eigentliche Engpass: Kontext, nicht nur Parameterzahl

Marketing-Narrative drehen sich oft um Benchmarks und Modellgenerationen. In der Praxis der Softwareentwicklung scheitern Agenten seltener an „zu wenig IQ“ als an **falschem, veraltetem, fehlendem oder überladenem Kontext**.

Typische Failure Modes:

Symptom	Eigentliche Ursache
Halluzinierte API	Modell kennt eure interne Library nicht; Kontext fehlt
Falscher Fix	Agent sieht nicht den aktuellen Branch / die letzten CI-Logs
Endlosschleife	Tool-Ergebnisse werden nicht sauber in den Kontext zurückgeführt
Teure Latenz	Halbe Codebase + Docs + Chatverlauf im Window, 95 % irrelevant
Sicherheitsvorfall	Zu mächtige Tools, kein Sandboxing, Prompt-Injection über Tool-Output

Das **Context Window** ist ein knappes, teures Medium – vergleichbar mit dem Arbeitsspeicher eines Prozesses, nicht mit einer unendlichen Festplatte. Alles, was du „einfach mitgibst“, verdrängt etwas anderes und kostet Tokens, Zeit und Aufmerksamkeit des Modells.

Daraus folgt eine Leitthese dieses Buches:

> **Agentenqualität = Modellqualität × Kontextqualität × Toolqualität × Prozessqualität.**

Ein schwaches Glied dominiert das Ergebnis. MCP adressiert vor allem Kontext- und Toolqualität auf **Infrastrukturebene**; Context Engineering adressiert die **Betriebslogik**.

---

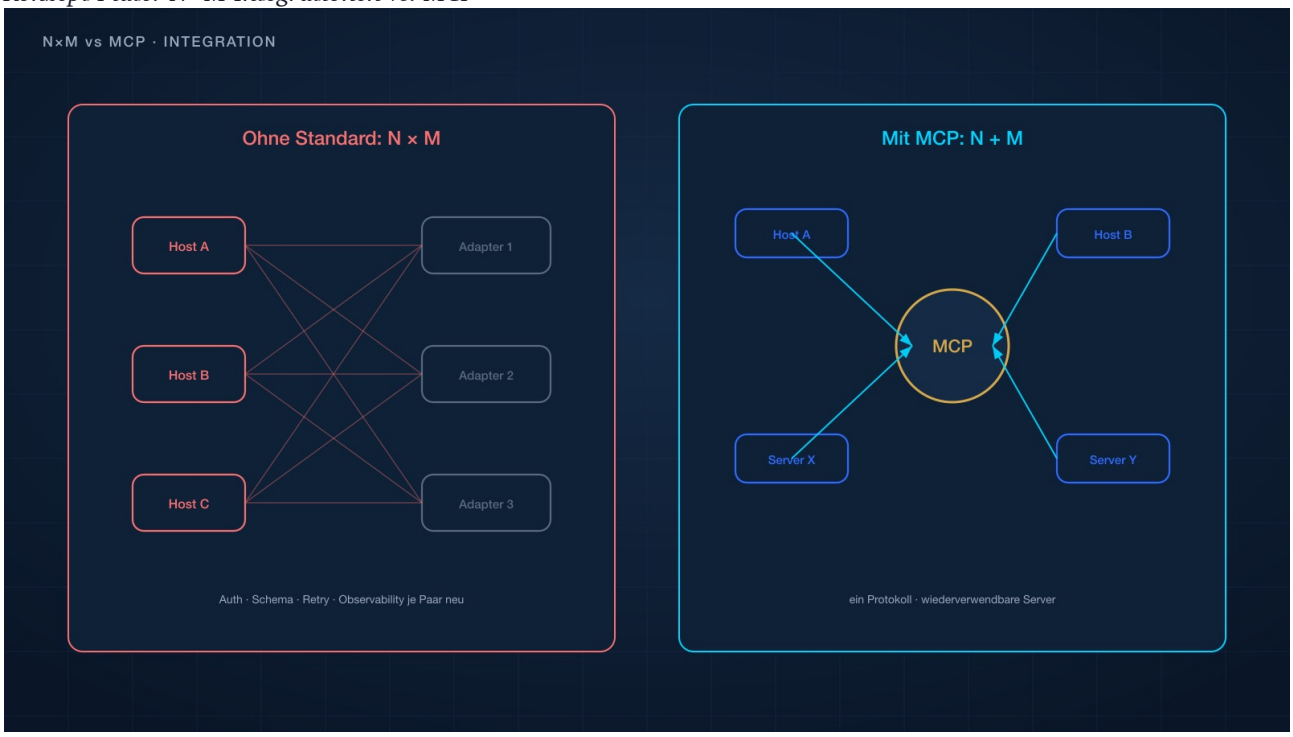
3. Das N×M-Problem klassischer Integrationen

Angenommen, du hast:

- *N* AI-Hosts (Claude Code, Cursor, eine interne Agent-Plattform, ein CI-Bot, …),
- *M* Systeme (GitHub, Jira, Postgres, Kubernetes, interner Feature-Store, …).

Ohne Standard baust du bis zu **N×M** Adapter: Auth, Schemas, Retries, Observability, Versionierung – jeweils neu. Teams lösen das mit internen SDKs, bis das SDK selbst zum Legacy wird.

Konzept-Plate: N×M-Integrationen vs. MCP



MCP bricht das Muster: Server exponieren Fähigkeiten **einmal**; Hosts sprechen **dieselbe Protokollsprache**. Neue IDEs und Agenten docken an bestehende Server an. Neue Server (z. B. euer interner Architecture-Decision-Server) werden für alle kompatiblen Hosts nutzbar.

Das eliminiert nicht Produktpolitik oder Auth-Komplexität – aber es verschiebt den Aufwand von „jedes Mal neu erfinden“ zu „Server bauen und betreiben“.

---

4. MCP in einem Absatz – und warum wir es vertiefen

Das **Model Context Protocol** ist ein offenes Protokoll (JSON-RPC-basiert), mit dem eine AI-Anwendung (**Host**) über eingebettete **Clients** Verbindungen zu einem oder mehreren **MCP-Servern** hält. Server stellen standardisiert bereit:

- **Tools** – ausführbare Funktionen,
- **Resources** – adressierbare Kontextdaten,
- **Prompts** – Vorlagen und geführte Workflows.

Transport typischerweise **STDIO** für lokale Server und **Streamable HTTP** für remote Dienste; beim Verbindungsaufbau verhandeln Client und Server Capabilities.

Die Kurzformel „USB-C für AI“ ist hilfreich, solange du sie nicht überinterpretierst: MCP macht Agenten nicht von selbst sicher oder klug. Es macht **Anschluss und Austausch standardisiert** – die Qualität der Server und des Context Engineerings bleibt eure Aufgabe.

In **Kapitel 2.6** führen wir MCP technisch ein; in **Kapitel 4** binden wir es an Agenten-Architekturen; in **Kapitel 10** baust du Server selbst.

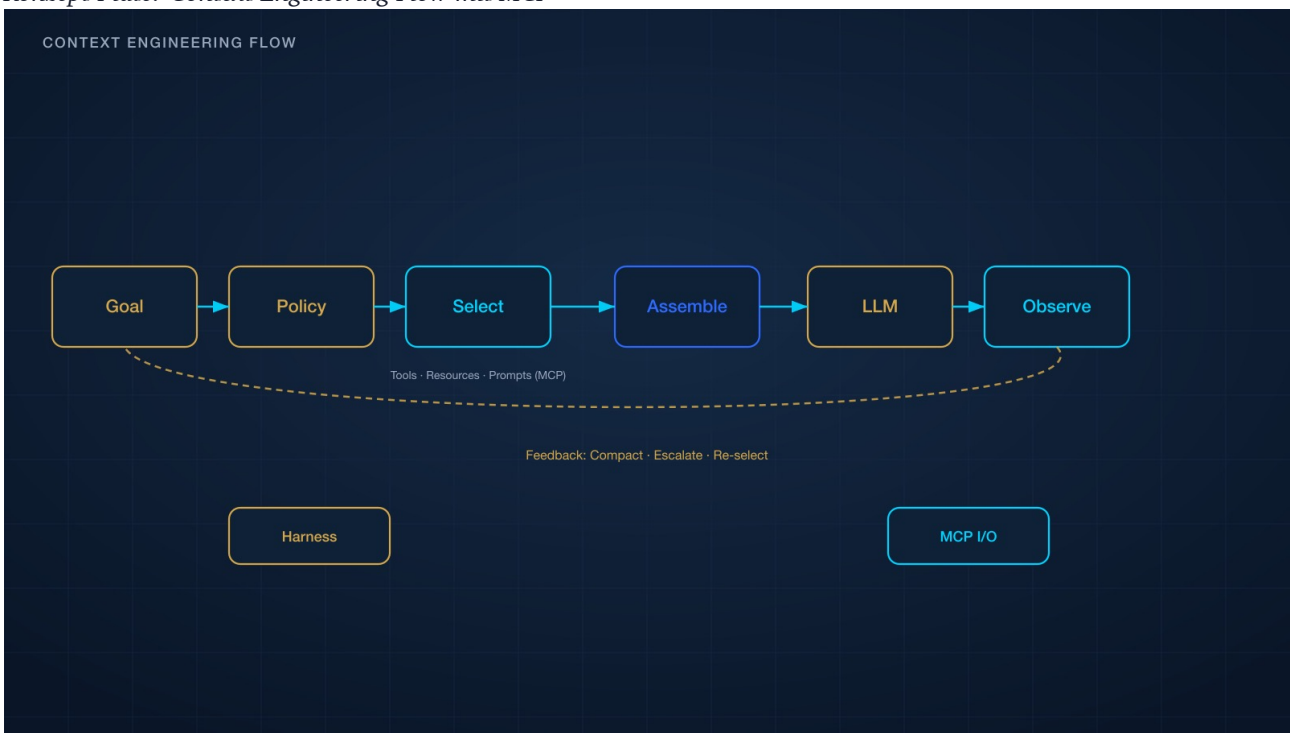
---

5. Context Engineering: die Disziplin hinter dem Prompt

Prompt Engineering war der Einstieg: Formuliere klar, gib Beispiele, setze Rollen. Für Agenten reicht das nicht. **Context Engineering** umfasst u. a.:

1. **Selektion** – Was ist für *diesen* Schritt relevant?
2. **Beschaffung** – Woher kommt der Kontext (Repo, Graph, Issue, MCP-Resource)?
3. **Repräsentation** – Rohdatei, Diff, Zusammenfassung, strukturiertes JSON?
4. **Aktualität** – Ist der Stand verifiziert (Commit-Hash, Timestamp)?
5. **Budget** – Wie viele Tokens bleiben für Reasoning und Tool-Results?
6. **Schutz** – Welche Daten dürfen das Modell / den Remote-Host nie erreichen?
7. **Feedback** – Werden Tool-Ergebnisse und User-Korrekturen persistiert (Memory)?

Konzept-Plate: Context Engineering Flow mit MCP



MCP ist dabei oft die **Transport- und Discovery-Schicht**; Context Engineering ist die **Policy**, die entscheidet, was wann fließt. Beides zusammen ist die „neue Architektur“, von der das Vorwort spricht.

---

6. Was du in den vier Teilen des Buches erwartest

Teil I – Grundlagen

Du verstehst LLMs so weit, dass du **realistische Erwartungen** an Agenten stellst: Evolution (Kap. 1), Funktionsweise und Context Window (Kap. 2), Fähigkeiten, Grenzen und Context Engineering (Kap. 3).

Teil II – Agenten und MCP als Schicht

Definition und Muster von Agenten, Deep Dive MCP, fortgeschrittene Speicher- und Multi-Agent-Architekturen, Framework-Ökosystem 2026 inklusive Evaluierung.

Teil III – Praxis im SDLC

Coding Agents im Entwicklungsprozess, Multi-Agent-Teams, IDE-Integration, **Hands-on MCP-Server-Bau**, Risiken und Governance.

Teil IV – Zukunft und Branche

Roadmaps, neue Rollen (Context Engineer, MCP Server Engineer), Auswirkungen auf Teams und Karrieren.

---

7. Voraussetzungen und Arbeitsweise

Mitbringen solltest du:

- Erfahrung mit mindestens einer modernen Sprache (Beispiele nutzen Python und TypeScript),
- Git-Workflows und grobes CI-Verständnis,
- Bereitschaft, mit CLI, JSON und APIs zu arbeiten,
- ethischen Anspruch: keine blinden Agenten auf Produktionsgeheimnissen.

Arbeitsempfehlung:

- Lies aktiv: skizziere deine eigene Host–Server-Landschaft.
- Probiere Mini-Beispiele aus Kapitel 2; vertiefe in Kapitel 10.
- Führe ein „Context Journal“: Welche Informationen braucht euer Agent wirklich für die Top-5-Aufgaben?

---

8. Leitmotive, die sich durch das Buch ziehen

1. **Kontext ist Infrastruktur** – behandle ihn wie Datenpipelines, nicht wie Prompt-Folklore.
2. **Standards schlagen Eigenbau**, sobald mehr als ein Host oder Team andockt.
3. **Weniger Kontext kann mehr sein** – Selektion und On-Demand schlagen Dump.
4. **Tools sind Privilegien** – jedes Tool ist eine Angriffs- und Fehlerfläche.
5. **Menschen bleiben verantwortlich** – für Ziele, Reviews, Produktion und Ethik.

---

9. Wie dieses Buch mit Code und Tools umgeht

Dieses Buch ist **kein SDK-Handbuch** und keine IDE-Versionsdokumentation. Es ist ein **Architektur- und Musterbuch** für agentenbasierte Softwareentwicklung:

1. Zuerst das **stabile Muster** (Protokoll, Context Engineering, Harness, Policies).
2. Dann – wo hilfreich – eine **knappe Illustration** mit einem gängigen SDK oder Host (*Stand Juli 2026*), klar als zeitgebunden markiert.
3. Ausführliche, lauffähige Versionen und CI-getestete Demos liegen im **Companion** `code/` (Tags `book-2026-07` / `latest`, siehe `code/COMPATIBILITY.md`).

Wenn sich FastMCP, LangGraph oder die Config-Syntax eines Hosts ändern, bleibt die Lehrabsicht erhalten: Capabilities portabel machen, Kontext diszipliniert laden, Tools als Privilegien behandeln.

Zur Orientierung: **stabile** Schicht = Protokoll & Architektur; **mittelstabile** Schicht = methodische Muster; **volatile** Schicht = konkrete SDK-/Host-Beispiele (siehe auch Anhang D).

Key Takeaways der Einleitung

- Software-KI reift von Chat über Tool-Assistenten zu **zielgetriebenen Agenten**.
- Der kritische Engpass ist oft **Kontextqualität und -architektur**, nicht nur Modellwahl.
- Klassische Integrationen skalieren mit **N×M**; MCP zielt auf **N+M** über eine gemeinsame Protokollschicht.
- **Tools, Resources und Prompts** sind die Kern-Primitives von MCP-Servern.
- **Context Engineering** steuert, was wann ins Context Window gelangt.
- Dieses Buch verbindet LLM-Grundlagen, Agenten-Muster und **MCP-Praxis** bis zum eigenen Server-Bau.
- Konkrete SDK-/Host-Beispiele sind **zeitgebunden**; lernrelevant sind Architektur und Muster.

Was als Nächstes kommt

Kapitel 1 erzählt die Evolution der Sprachmodelle bis zum Punkt, an dem standardisierte Kontext-Protokolle unvermeidlich wurden. **Kapitel 2** erklärt die Technik unter der Haube – und führt MCP als Antwort auf die Grenzen reiner Context Windows und klassischen RAGs ein.

Wenn Ton und Tiefe für dich passen: Feedback geben, dann gehen wir in Kapitel 3 und den vollständigen Agenten-/MCP-Block.

Kapitel 1: Die Evolution der Sprachmodelle

Teil I · Grundlagen – Large Language Models und Kontext verstehen

---

Dieses Kapitel legt die **historische und konzeptionelle Basis** für alles Folgende: wie aus n-Grammen und rekurrenten Netzen Transformer mit Self-Attention wurden – und warum genau diese Architektur große Kontexte und In-Context Learning überhaupt möglich macht. Es steht am Anfang von Teil I, bevor Kapitel 2 unter die Haube schaut und Kapitel 3 Fähigkeiten, Grenzen und Context Engineering systematisiert.

Der Leitgedanke: Heutige Coding Agents und Tool-Assistenten sind nicht „plötzlich intelligent geworden", sondern das Ergebnis einer **API- und Systemevolution** – von Completion über Chat und Function Calling bis zum Agent-Loop. Parallel wächst die Vielfalt von Modellen (open vs. closed) und Tools; das macht **standardisierte Kontext- und Tool-Schichten** (später MCP) architektonisch nötig, nicht nur wünschenswert.

Besonders relevant, wenn du Marketing-Narrative („Emergenz", „das Modell kann alles") von **Engineering-Realität** trennen willst. Achte darauf, die Geschichte nicht als Selbstzweck zu lesen: Jede Station erklärt, warum Agenten später Kontext, Tools und Verifikation brauchen – und warum Modellwahl und Capability-Schicht entkoppelt werden sollten.

Lernziele

Nach diesem Kapitel kannst du:

- die Entwicklung von klassischen Sprachmodellen zu modernen Transformern grob nachzeichnen und erklären, warum **Attention** den Weg für lange Kontexte geebnet hat,
- Pretraining, Instruction Tuning und Alignment (RLHF/RLAIF) als Bausteine heutiger Assistenten einordnen,
- den Übergang von reiner Text-Completion über Chat-APIs zu **Tool Use** und Agenten beschreiben,
- Vor- und Nachteile von **Open-Source-** und **Closed Models** für Softwareteams bewerten,
- begründen, warum mit wachsender Tool- und Kontextvielfalt **standardisierte Protokolle** (allen voran MCP) architektonisch nötig wurden.

1.1 Von n-Grammen zu Transformern

Die alte Welt: Zählen, Markov, dann Sequenzen

Bevor „Large Language Model" zum Alltagsbegriff in Engineering-Teams wurde, war Sprachmodellierung ein eher spezielles NLP-Thema. Frühe Ansätze schätzten die Wahrscheinlichkeit des nächsten Wortes aus **n-Gramm-Statistiken**: Wie oft folgt auf „continuous integration" das Wort „pipeline"? Das skaliert schlecht, leidet unter Daten-Sparse und kennt keine lange Abhängigkeit über Satzgrenzen hinweg.

Neuronale Netze brachten **Wort-Embeddings** und später **Rekurrente Netze (RNNs/LSTMs)**: Sequenzen wurden Schritt für Schritt verarbeitet, ein versteckter Zustand „erinnerte" sich an Vergangenes. Für viele Aufgaben reichte

das – für sehr lange Kontexte und massiv paralleles Training stieß die Architektur an Grenzen. Information musste sich durch die Zeitachse „durchquetschen"; Gradienten und Parallelisierung wurden zum Flaschenhals.

Der Wendepunkt: Attention is (almost) all you need

Der Transformer (Vaswani et al., 2017) ersetzte die starre Rekurrenz weitgehend durch **Self-Attention**: Jedes Token kann – vereinfacht gesagt – gewichtet auf andere Token der Sequenz blicken. Plötzlich ließen sich lange Abhängigkeiten direkter modellieren, und das Training ließ sich auf GPUs/TPUs massiv parallelisieren.

> **Analogie:** Ein RNN liest einen Roman Wort für Wort und darf nur einen Notizzettel (Hidden State) mitführen. Self-Attention erlaubt dem Leser, bei Bedarf an jede frühere Seite zurückzublättern – mit einem Score, wie relevant diese Seite *jetzt* ist. Das ist teurer (Rechenaufwand wächst mit Sequenzlänge), aber expressiver.

Für die Softwareentwicklung ist diese Geschichte relevant, weil sie die **technische Voraussetzung für große Context Windows und In-Context Learning** schafft: Das Modell kann Muster aus Beispielen im Prompt „mitlesen", ohne für jede Aufgabe neu trainiert zu werden. Agenten, die Diffs, Logs und Tool-Ergebnisse im selben Fenster sehen, stehen auf genau diesem Fundament.

```
timeline
    title Stationen der Sprachmodell-Evolution (vereinfacht)
    1990er–2000er : n-Gramme und klassisches Statistical NLP
    2013–2017     : Word2Vec, RNNs/LSTMs, Seq2Seq
    2017          : Transformer / Self-Attention
    2018–2020     : BERT, GPT-2, Transfer Learning
    2020–2022     : GPT-3, In-Context Learning, Skalierung
    2022–2023     : Chat-Assistenten, RLHF, Massennutzung
    2023–2024     : Tool Use, Function Calling, frühe Agenten
    2024–2026     : MCP, Context Engineering, Coding Agents in IDEs
```

Was „groß" an Large Language Models ist

„Large" bezieht sich historisch vor allem auf **Parameterzahl**, **Trainingsdaten** und **Compute**. Für dich als Praktiker zählt weniger die exakte Milliarden-Zahl als die Konsequenz:

- Modelle lernen allgemeine Sprach- und Weltregularitäten im **Pretraining**,
- sie können im Prompt **Aufgabenformate** adaptieren,
- und sie bleiben **probabilistische Generatoren** – keine garantierten Wissensdatenbanken und keine sicheren Executor.

Diese Unterscheidung wird uns durch das ganze Buch begleiten.

1.2 Die Skalierungsära: Pretraining, Emergenz, Alignment

Pretraining: das Fundament

Im Pretraining maximiert das Modell im Kern eine simple Zielfunktion: **Vorhersage des nächsten Tokens** (oder verwandte Maskierungsziele). Auf riesigen Textkorpora entsteht so ein reichhaltiges internes Modell von Syntax, Idiomen, Code-Konventionen und statistischen Fakten – inklusive Biases und Fehlern der Trainingsdaten.

Für Coding-Modelle wurde Quellcode (öffentliche Repos, Dokumentation, Q&A) ein zentraler Datenbestandteil. Deshalb „können" moderne Modelle oft Python und TypeScript erstaunlich gut – und halluzinieren trotzdem interne APIs eures Monolithen, die nie im Training waren.

Instruction Tuning: vom Rohmodell zum Assistenten

Rohe Base Models vervollständigen Text. Produktive Assistenten brauchen zusätzlich **Instruction Tuning**: Feintuning auf Paaren aus Anweisung und erwünschter Antwort. Erst dadurch wird aus „setze den Absatz fort“ ein System, das auf „Schreibe einen Unit-Test für diese Funktion“ sinnvoll reagiert.

RLHF, RLAIF und die Kunst der Nutzbarkeit

Reinforcement Learning from Human Feedback (RLHF) und Varianten mit AI-Feedback (**RLAIF**) richten das Modell an Präferenzen aus: hilfreich, harmlos, ehrlich – zumindest in der Theorie. Für Entwickler heißt das praktisch:

- Antworten werden dialogischer und tool-affiner,
- Safety-Filter und Verweigerungspolitiken kommen dazu,
- das Modell optimiert auf *bewertete* Qualität, nicht auf absolute Wahrheit.

> **Pitfall:** Alignment macht Modelle angenehmer, aber nicht infallibel. Ein höflich halluzinierter Import ist im Code-Review gefährlicher als ein grober Syntaxfehler.

Emergenz – nützlich als Beobachtung, gefährlich als Mythos

Mit der Skala tauchten Fähigkeiten auf, die man bei kleinen Modellen nicht sah: besseres Few-Shot-Lernen, grobes Reasoning, Code-Synthese. „Emergenz“ ist ein Forschungs- und Debattenbegriff. Als Praktiker nutzt du ihn am besten so:

- **Ja:** Größere/bessere Modelle können qualitative Sprünge machen.
- **Nein:** Das rechtfertigt nicht, Tests, Typen und Specs wegzulassen.
- **Vor allem:** Viele „Emergenzen“ im Produkt sind in Wahrheit **besseres Prompting, bessere Tools und besserer Kontext**.

Skalierungsgesetze – und ihre Ergänzung 2025/26

Skalierungsgesetze beschreiben grob, wie Loss mit Compute, Daten und Parametern zusammenhängt. Parallel hat sich die Praxis verschoben:

- **Inference-Time Compute:** mehr Nachdenken (Chain-of-Thought, Search, Agent-Loops) statt nur größeres Modell,
- **Spezialisierung:** Coding-, Reasoning- und kleine On-Device-Modelle,
- **System-Skalierung:** RAG, Memory, Tools, MCP – also Intelligenz *um* das Modell herum.

Genau diese System-Skalierung ist der rote Faden ab Kapitel 2.6.

1.3 Von Completion zu Chat zu Tools

Completion: die Ur-API

Frühe APIs waren **Completion-APIs**: Du schickst einen Prompt-String, bekommst Fortsetzungstext. Für Code-Vervollständigung im Editor war das ideal. Für Multi-Turn-Dialoge und strukturierte Aktionen wurde es unbeholfen – Rollen, Systemprompts und Tool-Ergebnisse mussten in einen String gehackt werden.

Chat: Messages als Schnittstelle

Chat-APIs strukturieren den Kontext als **Nachrichtensequenz** mit Rollen (`system`, `user`, `assistant`). Das ist nicht nur UX: Es ist ein **Kontextprotokoll im Kleinen**. Der Host (die Anwendung) entscheidet, welche Messages bleiben, welche zusammengefasst werden, was der Systemprompt darf.

```
# Illustratives Chat-Message-Format (vereinfacht, OpenAI-ähnlich)
messages = [
    {
        "role": "system",
        "content": "Du bist ein Senior-Python-Entwickler. Antworte präzise.",
    },
    {
        "role": "user",
        "content": "Erkläre, warum dieser Test flaky sein könnte:\n\n" + test_code,
    },
]
# response = client.chat.completions.create(model="…", messages=messages)
```

Function Calling / Tool Use: das Modell darf handeln (indirekt)

Der nächste Schritt: Das Modell erzeugt nicht nur Prosa, sondern **strukturierte Aufrufwünsche** – z. B. JSON mit Funktionsname und Argumenten. Der Host führt die Funktion aus und schreibt das Ergebnis als weitere Message zurück.

```
# Illustratives Tool-Schema (Ausschnitt)
tools = [
    {
        "type": "function",
        "function": {
            "name": "run_tests",
            "description": "Führt den Test-Suite-Befehl im Projekt aus.",
            "parameters": {
                "type": "object",
                "properties": {
                    "path": {
                        "type": "string",
                        "description": "Optionaler Testdateipfad",
                    }
                },
            },
        },
    }
]
```

```
// TypeScript: Tool-Ergebnis fließt zurück in den Dialog (Skizze)
type ToolCall = { name: string; arguments: Record<string, unknown> };

async function handleToolCall(call: ToolCall): Promise<string> {
  if (call.name === "run_tests") {
    // Host führt aus – nicht das LLM selbst
    return await runTests(String(call.arguments.path ?? ""));
  }
  throw new Error(`Unbekanntes Tool: ${call.name}`);
}
```

Wichtig für später:

- **Das Modell führt nichts aus.** Der Host ist Executor und Policy-Enforcer.
- Tool-Schemas fressen **Context-Budget**.
- Jede App definierte lange Zeit **eigene** Schemas, Transports und Discovery – Vendor- und Framework-Lock-in inklusive.

Von Tool Use zum Agenten

Ein einzelner Tool-Call ist noch kein Agent. Ein **Agent** nutzt Tools in einer Schleife aus Ziel, Planung, Aktion und Beobachtung – oft über Minuten und viele Schritte. Sobald mehrere Datenquellen und wechselnde Toolsets dazukommen, rächt sich fehlende Standardisierung.

```
flowchart TB
    subgraph evolution["Schnittstellen-Evolution"]
        C[Completion API<br/>ein String] --> CH[Chat Messages<br/>Rollen + Historie]
        CH --> FC[Function Calling<br/>feste Tool-Schemas]
        FC --> AG[Agent Loop<br/>Plan → Act → Observe]
        AG --> MCP[Standardisierte<br/>Kontext- & Tool-Schicht<br/>MCP]
    end
```

1.4 Open-Source vs. Closed Models – und der Aufstieg standardisierter Kontext-Protokolle

Die Landschaft 2026 (pragmatisch)

Ohne Vendor-Ranking im Marketing-Ton – als Entscheidungsraum:

Dimension	Closed / Managed APIs	Open Weights / Self-Host
Beispiele (illustrativ)	GPT-, Claude-, Gemini-Klassen	Llama-, Qwen-, Mistral-, DeepSeek-Linien u. a.
Einstieg	schnell, wenig Ops	mehr Ops, mehr Kontrolle
Datenpfad	Daten verlassen oft die eigene Infra (vertraglich regeln!)	lokal/VPC möglich
Spitzenqualität / Features	häufig Early Access zu Reasoning, langen Fenstern, nativen Tools	stark abhängig von Fine-Tunes und Serving-Stack
Kostenmodell	pro Token / pro Request	Hardware + Energy + People
Anpassbarkeit	begrenzt (Prompts, LoRA-Angebote, System Layer)	hoch (Fine-Tunes, Routing, Distillation)
Compliance	AVV, Regionen, Zertifikate prüfen	Verantwortung liegt bei euch

In echten Systemen siehst du 2026 oft **Hybride**: Frontier-Modelle für schwierige Planung, kleinere/cheaper Modelle für Klassifikation, Routing und Routine-Edits – orchestriert von einer Agent-Runtime.

Der unterschätzte Lock-in: nicht nur Gewichte, sondern Tools

Viele Teams diskutieren „Open vs. Closed“ nur auf Modellebene. Der schmerzhaftere Lock-in sitzt häufig **darüber**:

- proprietäre Tool-Formate der jeweiligen Plattform,
- IDE-spezifische Indexer und Memory-Formate,
- Agent-Frameworks mit eigenen Plugin-APIs,
- Prompt-Templates, die nur in einem Host funktionieren.

Wenn dein „Agent“ nur in *einer* Oberfläche lebt und euer Jira-Connector nur dort, hast du ein Produkt – keine Plattform.

MCP: Interoperabilität als Architekturentscheidung

Das **Model Context Protocol** (ab Ende 2024 öffentlich vorangetrieben, 2025/26 mit breitem Ökosystem in IDEs, Desktop-Hosts und Server-Libraries) adressiert genau diese Schicht:

- Ein **MCP-Server** kapselt eine Fähigkeit (z. B. GitHub, interner ADR-Store, Test-Runner).

- Ein **MCP-Host** (Claude Code, Cursor, eigene Runtime …) startet pro Server einen **MCP-Client**.
- Discovery, Aufruf und Kontextzugriff folgen einem gemeinsamen Protokoll statt ad-hoc JSON.

```
flowchart LR
    subgraph hosts["Hosts N"]
        H1[IDE Host]
        H2[CLI Agent]
        H3[CI Bot]
    end

    subgraph protocol["MCP"]
        P[Standardisierte<br/>Tools / Resources / Prompts]
    end

    subgraph servers["Server M"]
        S1[Repo & Git]
        S2[Issues]
        S3[CI / Tests]
        S4[Project Memory]
    end

    H1 --> P
    H2 --> P
    H3 --> P
    P --> S1
    P --> S2
    P --> S3
    P --> S4
```

Historische Einordnung für Praktiker:

1. **Vor MCP:** Jeder Host eigene Plugins; Frameworks erfinden eigene Tool-Registries.
2. **Function Calling-Welle:** Standard *innerhalb* einer Model-API, nicht *zwischen* Apps und Unternehmenssystemen.
3. **MCP-Welle:** Standard *zwischen* AI-Anwendung und externer Welt – komplementär zu Model-APIs, nicht als Ersatz fürs LLM.

Damit ändert sich die Teamfrage von „Welches Plugin schreiben wir für IDE X?“ zu „Welchen **MCP-Server** betreiben wir für Capability Y – und welche Hosts dürfen andocken?“

Open Models + MCP: eine besonders starke Kombination

Self-Hosted Models ohne Tool-Ökosystem bleiben Insellösungen. Closed Models ohne portable Kontextschicht erzeugen strategische Abhängigkeit. **Open (oder portable) Agent-Infrastruktur + MCP-Server** gibt dir:

- Wahlfreiheit beim Modell (Routing, Failover, Kosten),
- Wiederverwendung von Server-Logik über Hosts hinweg,
- klarere Security-Grenzen (Server als Trust Boundary).

> **Best Practice:** Entkopple **Modellwahl** (häufig wechselnd) von **Kontext- und Tool-Schicht** (sollte stabil und versioniert sein). MCP ist ein Mittel zu dieser Entkopplung.

Praxisbeispiel: Derselbe Use Case, drei Generationen

Aufgabe: „Fasse die letzten Fehler der CI-Pipeline zum Service `billing` zusammen und schlage einen Fix vor.“

Generation	Vorgehen	Limit
Chatbot 2022	Mensch kopiert Log-Auszug in den Chat	Auswahlbias, keine Live-Daten

Generation	Vorgehen	Limit
Tool-Assistent 2024	Feste Funktion `get_ci_logs(service)` im Host verdrahtet	Nur dieser Host; Schema-Duplikation
MCP-basiert 2026	Host entdeckt CI-Server-Tools/Resources dynamisch; Resource `ci://billing/latest` + Tool `rerun_job`	Server-Qualität, Auth und Context Policy bleiben kritisch

Der Sprung ist weniger „das Modell wurde magisch besser“ als „das System darf **standardisiert** die richtige Welt sehen und anfassen“.

Best Practices

1. **Wähle Modelle nach Workload**, nicht nach Hype: Planung/Architektur vs. Bulk-Edit vs. Klassifikation trennen.
2. **Investiere in portable Kontext-Schnittstellen** (MCP-Server, klare Resources), bevor du den dritten proprietären Plugin-Stack baust.
3. **Behandle Alignment und Safety als Produktfeature**, nicht als lästige Bremse – besonders bei schreibenden Tools.
4. **Versioniere Systemprompts und Tool-Policies** wie Code (Review, Changelog).
5. **Miss Systemqualität** (Task Success, Human Edit Rate, Token Cost), nicht nur Modell-Benchmarks.

Pitfalls

Pitfall	Folge	Gegenmaßnahme	Kapitel-Querverweis
„Wir brauchen nur das größte Modell“	Kosten ↑, Kontextmüll bleibt	Routing + Context Engineering	Kap. 1
Tool-Logik tief in einer IDE vergraben	Lock-in, Doppelarbeit	MCP-Server / shared capability layer	Kap. 1
Open-Source-Modell ohne Serving- und Eval-Disziplin	Latenz, Drift, stille Qualitätsfälle	Gleiche SRE-Standards wie für APIs	Kap. 1
Completion-Denken in Agenten-Systemen	Keine klare Rollen-/Tool-Grenze	Message- und Tool-Protokolle ernst nehmen	Kap. 1
Trainingdaten mit „unserem Wissen“ verwechseln	Halluzinierte interne APIs	Runtime-Kontext (Repo, Docs, MCP)	Kap. 1

Key Takeaways

- Transformer und Self-Attention haben die technische Basis für große Kontexte und In-Context Learning gelegt.
- Heutige Assistenten sind Stacks aus **Pretraining + Instruction Tuning + Alignment**, nicht „nur“ Next-Token-Prediction in Reinform.
- Die API-Evolution **Completion → Chat → Tool Use → Agent Loop** verschiebt Arbeit vom Menschen im Clipboard zum Host als Executor.
- Open vs. Closed ist eine **Trade-off-Matrix** (Ops, Compliance, Qualität, Kosten) – Hybride sind normal.
- Der strategische Hebel 2026 ist oft die **Kontext- und Tool-Schicht**; MCP etabliert sich als Standard gegen das N×M-Problem.
- Entkopple Modellwahl von stabilen Capabilities: **Server und Policies überdauern Modellgenerationen**.

Was als Nächstes kommt

Kapitel 2 taucht unter die Haube: Tokenisierung, Attention, Context Window, Sampling – und erklärt, warum reines „mehr Tokens“ und klassisches RAG an Grenzen stoßen. Dort lernst du MCP bereits als moderne Form der **Kontext-Provisionierung** kennen: Host, Client, Server, Tools, Resources, Prompts und Context Flow.

Übungsanregungen

1. Liste fünf Aufgaben in deinem Team, die heute Clipboard-Kontext brauchen. Welche wären **Resources**, welche **Tools**?
2. Skizziere eure aktuellen AI-Hosts und Integrationen als N×M-Matrix. Wo wäre ein einzelner MCP-Server sofort multi-host-fähig?
3. Nimm einen internen API-Namen, der *nicht* öffentlich dokumentiert ist, und teste (sicher, ohne Secrets), ob euer Modell ihn „kennt“. Was bedeutet das Ergebnis für eure Kontextstrategie?