PROMPT ENGINEERING MEISTERN

BAND 7

Prompting-fuer-Entwickler

Code, APIs und Automatisierung

Belkis Aslani

2026

Prompt Engineering Meistern
Band 7: Prompting-fuer-Entwickler – Code, APIs und Automatisierung

1. Auflage, März 2026

Satz und Layout: Eigensatz des Autors
Umschlaggestaltung: Belkis Aslani

Inhaltsverzeichnis

Vorwort

Sechs Bände lang warst du Endnutzer. Du hast in Chat-Fenster getippt, Ergebnisse gelesen, iteriert. Gut so – das ist die Grundlage.

Aber jetzt bist du bereit für den nächsten Schritt: KI nicht nur nutzen, sondern einbauen. In deine Software. In deine Workflows. In deine Produkte.

Was diesen Band anders macht

In den Bänden 1 bis 6 ging es darum, wie du mit KI *kommunizierst*. In Band 7 geht es darum, wie du KI *programmierst*.

Das ist ein fundamentaler Unterschied:

- Du schreibst nicht mehr einzelne Prompts – du baust Systeme, die tausende Prompts automatisch generieren und verarbeiten.
- Du copy-pastest keine Antworten mehr – du verarbeitest strukturierte API-Responses in deinem Code.
- Du wartest nicht mehr auf Ergebnisse – du baust Pipelines, die asynchron, parallel und skalierbar arbeiten.
- Du promptest nicht mehr allein – du baust Agenten, die Tools nutzen, Entscheidungen treffen und selbstständig handeln.

Willkommen in der Welt des programmatischen Promptings.

Was du brauchst

Technische Voraussetzungen für diesen Band:

- **Grundlegende Programmierkenntnisse** – Python oder JavaScript. Du musst kein Senior Developer sein, aber du solltest wissen, was eine Funktion, eine Variable und ein API-Call ist.
- **Ein Terminal/Kommandozeile** – Du wirst Befehle ausführen.
- **API-Zugang** – Mindestens einen API-Key von Anthropic (Claude), OpenAI oder Google (Gemini). Kostenlose Tiers reichen für die Übungen.
- **Eine Entwicklungsumgebung** – VS Code, Cursor, oder ein beliebiger Editor.
- **Neugier** – Weil die Landschaft sich schnell bewegt und du experimentieren musst.

Wenn du nicht programmierst: Lies trotzdem die Kapitel über Code-Generierung (Kapitel 1-2) und das Konzept-Kapitel über Context Engineering (Kapitel 9). Der Rest ist für Entwickler.

Was dich erwartet

Code-Generierung – Nicht “Schreib mir eine Funktion”, sondern: Wie du KI für Architekturentscheidungen, Code-Reviews, Refactoring und Test-Generierung nutzt. Von der Einzelfunktion zum Gesamtsystem.

Agentic Coding Tools – Cursor, GitHub Copilot, Claude Code, Windsurf, Cline, Aider. Was können sie? Wie unterscheiden sie sich? Und: Wie promptest du innerhalb dieser Tools optimal?

Die LLM-APIs – Anthropic, OpenAI, Google, Open Source. Preise, Modelle, Features. Structured Output, Vision, Streaming, Caching. Alles, was du als Entwickler wissen musst.

Programmatisches Prompting – Prompt-Templates, Variablen, Antwortverarbeitung, Fehlerbehandlung, Retry-Logik. Mit echtem Code in Python und TypeScript.

RAG (Retrieval Augmented Generation) – Deine eigenen Daten in die KI bringen. Embeddings, Vektor-Datenbanken, Chunking-Strategien. Das Fundament jeder ernsthaften KI-Anwendung.

Tool Use und Function Calling – KI, die nicht nur Text generiert, sondern Funktionen aufruft: Datenbanken abfragen, APIs ansprechen, Dateien bearbeiten. Die Brücke zwischen Text und Aktion.

Agentische Systeme – KI-Agenten, die planen, handeln und iterieren. MCP (Model Context Protocol), Multi-Step-Agents, Orchestrierung. Der aktuellste und spannendste Bereich.

Fine-Tuning vs. Prompting – Wann reicht gutes Prompting? Wann lohnt sich Fine-Tuning? Die Entscheidungsmatrix mit konkreten Kosten-Nutzen-Analysen.

Context Engineering – Der Nachfolger von Prompt Engineering. Wie du den gesamten Kontext eines LLM-Aufrufs designst – nicht nur den Prompt, sondern System-Prompts, Tool-Definitionen, Retrieval-Ergebnisse, Konversationshistorie und Caching.

Die KI-Entwickler-Landschaft 2026

Bevor wir einsteigen, ein Überblick über den Stand der Dinge:

Bereich	Stand 2024	Stand 2026
Code-Assistenten	GitHub Copilot dominiert	Dutzende Tools, agentic by default
APIs	Text rein, Text raus	Multimodal, Tool Use, Structured Output
RAG	Experimentell, komplex	Standard-Pattern, viele Frameworks
Agenten	Demo-Stage, fragil	Produktionsreif für definierte Workflows
Fine-Tuning	Teuer, komplex	Günstiger, aber Prompting oft ausreichend
Context Engineering	Buzzword	Etablierte Disziplin
Open Source	Llama 2, Mixtral	Llama 4, DeepSeek-V3, Qwen 3

Die Entwicklung in den letzten zwei Jahren war exponentiell. Was 2024 als Research-Paper existierte, ist 2026 ein npm-Paket. Was 2024 ein Startup war, ist 2026 ein Feature von VS Code.

Dieses Buch gibt dir das Wissen, um in dieser Landschaft nicht nur mitzuschwimmen, sondern zu navigieren.

Los geht’s. Wir bauen.

Belkis Aslani, März 2026

Kapitel 1: Code-Generierung mit KI – Von der Funktion zum System

“Schreib mir eine Python-Funktion, die prüft, ob eine Zahl eine Primzahl ist.” Das kann jeder. Und es ist ungefähr so nützlich wie “Schreib mir ein Gedicht über Liebe” – du bekommst ein generisches Ergebnis, das meistens funktioniert, aber selten genau das ist, was du brauchst.

In diesem Kapitel lernst du, wie du KI für professionelle Software-Entwicklung nutzt – von der Einzelfunktion bis zur Systemarchitektur.

Warum die meisten Code-Prompts schlecht sind

Problem 1: Zu wenig Kontext

```
SCHLECHT:
"Schreib eine Login-Funktion."

GUT:
"Schreib eine Login-Funktion für eine Express.js-API.
Tech-Stack: Node.js 22, TypeScript 5.4, PostgreSQL mit
Prisma ORM.
Auth: JWT mit Access + Refresh Token (RS256).
Passwort-Hashing: bcrypt, 12 Rounds.
Rate Limiting: Max. 5 Versuche pro 15 Min pro IP.
Input-Validierung: Zod-Schema.
Error Handling: Custom AppError-Klasse mit HTTP-Statusco-
des.
Logging: Pino-Logger, keine sensiblen Daten loggen."
```

Problem 2: Kein existierender Codebase-Kontext

KI weiß nicht, wie dein Projekt aufgebaut ist. Ohne Kontext schreibt sie Code, der nicht in dein Projekt passt.

```
Hier ist mein Projekt-Kontext:

ARCHITEKTUR: Hexagonal Architecture (Ports & Adapters)
ORDNERSTRUKTUR:
  src/
    domain/       # Business-Logik, Entities, Value Objects
    application/  # Use Cases, Ports (Interfaces)
    infrastructure/ # Adapter (DB, API, etc.)
    api/          # REST Controller, DTOs

KONVENTIONEN:
- Error Handling über Result-Type (kein throw)
- Dependency Injection via Constructor
- Alle DB-Zugriffe über Repository-Interfaces
- Tests: Vitest, AAA-Pattern, keine Mocks für Domain-Logik

Schreibe [AUFGABE] so, dass sie in diese Architektur passt.
```

Problem 3: Keine Qualitätsanforderungen

```
SCHLECHT:
"Schreib eine Funktion zum Sortieren."

GUT:
"Schreib eine Sortierfunktion mit folgenden Anforderungen:
- TypeScript, generisch (funktioniert mit jedem Typ)
- Stabile Sortierung
- O(n log n) im Average Case
- Immutable: Gibt neues Array zurück, verändert das Origi-
nal nicht
- Vergleichsfunktion als Parameter (wie Array.sort)
- JSDoc-Dokumentation
- Edge Cases: Leeres Array, ein Element, bereits sortiert
- Unit Tests mit Vitest (min. 5 Testfälle)"
```

Der professionelle Code-Prompt

Mein Framework für Code-Generierung:

```
AUFGABE: [Was soll gebaut werden?]
KONTEXT: [Wo wird es eingebaut? Existierender Code?]
TECH-STACK: [Sprache, Framework, Versionen]
ARCHITEKTUR: [Patterns, Strukturen, Konventionen]

ANFORDERUNGEN:
- Funktional: [Was muss es tun?]
- Nicht-funktional: [Performance, Sicherheit, Skalierbar-
keit]
- Edge Cases: [Was kann schiefgehen?]

QUALITÄT:
- Error Handling: [Strategie]
- Logging: [Was loggen, was nicht?]
- Tests: [Test-Framework, Testfälle]
- Dokumentation: [JSDoc/Docstring/README]

EINSCHRÄNKUNGEN:
- [Was NICHT verwenden: bestimmte Libraries, Patterns]
- [Max. Dateigröße, Komplexität]
- [Kompatibilität: Browser, Node-Versionen]
```

Code-Generierung nach Aufgabentyp

Neue Features implementieren

```
Implementiere folgendes Feature:

FEATURE: [Beschreibung]
USER STORY: "Als [Rolle] möchte ich [Aktion],
damit [Nutzen]."
AKZEPTANZKRITERIEN:
1. [Kriterium 1]
2. [Kriterium 2]
3. [Kriterium 3]

EXISTIERENDER CODE:
"""[Relevante Code-Abschnitte einfügen]"""

DATENMODELL: [Welche Daten sind beteiligt?]

IMPLEMENTIERE:
1. Datenbank-Migration (wenn nötig)
2. Backend-Logik (Service + Repository)
3. API-Endpoint (Controller + Validierung)
4. Tests (Unit + Integration)

Keine Frontend-Änderungen in diesem PR.
```

Bugs fixen

```
BUG: [Beschreibung des Problems]
ERWARTETES VERHALTEN: [Was sollte passieren?]
TATSÄCHLICHES VERHALTEN: [Was passiert stattdessen?]
REPRODUKTION: [Schritte zum Reproduzieren]

FEHLERMELDUNG:
"""[Error-Log einfügen]"""

RELEVANTER CODE:
"""[Code-Abschnitt einfügen]"""

AUFGABE:
1. Identifiziere die Ursache (Root Cause Analysis)
2. Schlage einen Fix vor (erkläre warum)
3. Implementiere den Fix
4. Schreibe einen Regressions-Test, der den Bug abdeckt
5. Prüfe: Kann der Fix andere Stellen beeinflussen?
```

Refactoring

```
Refactore folgenden Code:

"""[Code einfügen]"""

ZIEL: [z.B. Lesbarkeit, Performance, Testbarkeit,
Separation of Concerns, DRY]

REGELN:
- Verhalten darf sich NICHT ändern (pure Refactoring)
- Existierende Tests müssen weiterhin grün sein
- Keine neuen Dependencies einführen
- Commits in logische Schritte aufteilen

ERKLÄRE für jede Änderung:
- Was wurde geändert?
- Warum?
- Welches Design-Pattern wurde angewandt?
```

Code Reviews

```
Reviewe folgenden Code:

"""[Code einfügen / PR-Diff]"""

PRÜFE AUF:
1. KORREKTHEIT: Tut der Code, was er soll?
2. SICHERHEIT: SQL Injection, XSS, Auth-Lücken?
3. PERFORMANCE: N+1 Queries? Unnötige Berechnungen?
4. LESBARKEIT: Verständliche Variablennamen? Kommentare nö-
tig?
5. FEHLERBEHANDLUNG: Werden Fehler abgefangen und sinnvoll
behandelt?
6. TESTS: Ausreichend abgedeckt? Edge Cases?
7. KONVENTIONEN: Passt es zum Rest des Projekts?

FORMAT:
Für jedes Finding:
- Datei + Zeile
- Schwere: 🔴 Blocker / 🟡 Verbesserung / 🟢 Nitpick
- Problem
- Vorgeschlagener Fix
```

Test-Generierung

```
Schreibe Tests für folgenden Code:

"""[Code einfügen]"""

TEST-FRAMEWORK: [Jest/Vitest/pytest/Go testing/...]
TEST-ARTEN:
- Unit Tests (isoliert, gemockte Dependencies)
- Integration Tests (mit DB/API)
- Edge Case Tests

NAMING: describe/it-Pattern (BDD-Style)
PATTERN: Arrange-Act-Assert (AAA)

ABDECKUNG:
- Happy Path (normaler Ablauf)
- Error Cases (was kann schiefgehen?)
- Boundary Values (Grenzwerte)
- Null/Undefined/Empty
- Concurrent Access (wenn relevant)

Mindestens [X] Testfälle. Ziel: >90% Branch Coverage.
```

Architektur-Prompts

System Design

```
Designe die Architektur für folgendes System:

SYSTEM: [Beschreibung]
ANFORDERUNGEN:
- Funktional: [Was muss es können?]
- Nicht-funktional:
  - Erwartete Last: [Requests/Sekunde]
  - Verfügbarkeit: [99.9%? 99.99%?]
  - Latenz: [P95 < Xms]
  - Datenmenge: [Wie viel Daten?]
  - Compliance: [DSGVO? PCI-DSS?]

ERSTELLE:
1. High-Level-Architektur (Komponenten + Interaktion)
2. Datenfluss (Wie fließen Daten durch das System?)
3. Datenmodell (Entitäten, Beziehungen)
4. API-Design (Endpoints, Verben, Payloads)
5. Technologie-Empfehlung (mit Begründung)
6. Trade-offs (Was gewinnen wir? Was verlieren wir?)
7. Skalierungsstrategie (Wie wächst das System?)
```

Datenbank-Design

```
Designe ein Datenbankschema für [ANWENDUNG].

DATENBANK: [PostgreSQL/MySQL/MongoDB/...]
ENTITIES: [Liste der Entitäten]

FÜR JEDE ENTITY:
- Felder (Name, Typ, Constraints)
- Beziehungen (1:1, 1:N, N:M)
- Indizes (für erwartete Queries)

BERÜCKSICHTIGE:
- Normalisierung (3. Normalform oder bewusste Denormalisierung?)
- Soft Deletes (deleted_at statt physischem Löschen?)
- Audit-Trail (created_at, updated_at, created_by)
- Multi-Tenancy (wenn relevant)

ERSTELLE:
1. ER-Diagramm (als Text/Mermaid)
2. SQL CREATE TABLE Statements
3. Migrations-Dateien (für [Prisma/TypeORM/Alembic/...])
4. Seed-Daten (Testdaten)
```

Sprach-spezifische Tipps

Python

```
STIL: PEP 8, Type Hints (Python 3.12+), f-Strings
IMPORTS: Absolute Imports, keine Star-Imports
ASYNC: asyncio mit async/await wenn I/O-bound
DATENKLASSEN: Pydantic v2 für Validierung, dataclass für interne Strukturen
TESTING: pytest, parametrize für Varianten, fixtures für Setup
```

TypeScript

```
STIL: strict Mode, kein any, keine type assertions (as)
IMPORTS: Named Imports, Barrel-Files sparsam
ASYNC: Promises, async/await, keine Callbacks
VALIDIERUNG: Zod für Runtime-Validierung, TypeScript für
Compile-Time
TESTING: Vitest, describe/it, expect-Syntax
LINTING: ESLint + Prettier (konfiguriert)
```

Go

```
STIL: gofmt, Effective Go, keine generischen Interface{}-
Typen
ERROR HANDLING: Explicit error returns (if err != nil),
keine panic()
CONCURRENCY: Goroutines + Channels, context.Context für
Cancellation
TESTING: table-driven tests, testify für Assertions
STRUCTURE: cmd/ pkg/ internal/ nach Standard Go Layout
```

Rust

```
STIL: clippy, rustfmt, idiomatic Rust
ERROR HANDLING: Result<T, E>, thiserror für Custom Errors,
anyhow für Applications
OWNERSHIP: Borrowing statt Cloning wo möglich
ASYNC: tokio Runtime, async/await
TESTING: #[cfg(test)], unit tests im gleichen File,
integration tests in tests/
```

Anti-Patterns in der Code-Generierung

Anti-Pattern 1: Blind Copy-Paste

KI-generierter Code kann Bugs enthalten. Immer reviewen, nie blind einfügen.

Anti-Pattern 2: Zu große Prompts

“Schreib mir die komplette Anwendung” → funktioniert nicht. Bau sie Modul für Modul.

Anti-Pattern 3: Keine Iteration

Der erste Output ist selten perfekt. Iteriere:

```
"Der Code funktioniert, aber:
1. Die Error-Messages sind zu generisch
2. Das Logging fehlt
3. Die Funktion ist zu lang (>50 Zeilen)
Verbessere diese 3 Punkte."
```

Anti-Pattern 4: KI-Code nicht testen

KI-generierter Code muss genauso (oder strenger) getestet werden wie handgeschriebener Code. Vertraue nicht – verifiziere.

Anti-Pattern 5: Veraltete Patterns akzeptieren

KI-Modelle haben Wissens-Cutoffs. Code-Patterns können veraltet sein. Prüfe:
- Werden aktuelle API-Versionen verwendet?
- Sind deprecated Methoden im Code?
- Entspricht der Code aktuellen Best Practices?

Übungen

Übung 1: Der volle Code-Prompt

Wähle eine Aufgabe aus deinem Arbeitsalltag und schreibe einen vollständigen Code-Prompt (Aufgabe, Kontext, Tech-Stack, Architektur, Anforderungen, Qualität). Vergleiche das Ergebnis mit einem einfachen “Schreib mir X”-Prompt.

Übung 2: Code Review

Nimm ein eigenes Code-Stück (oder Open-Source-Code) und lass es reviewen. Stimmen die Findings? Gibt es False Positives?

Übung 3: Test-Generierung

Lass für eine bestehende Funktion Tests generieren. Laufen sie? Decken sie die wichtigen Edge Cases ab?

Übung 4: Bug-Analyse

Nimm einen echten Bug (z.B. aus einem GitHub Issue) und lass die KI eine Root Cause Analysis durchführen. Liegt sie richtig?

Kapitel 2: Agentic Coding Tools – Die neue Generation der Entwicklerwerkzeuge

2024 war das Jahr der Coding-Assistenten. 2025 war das Jahr der Coding-Agenten. 2026 ist das Jahr, in dem sie zum Standard geworden sind.

Der Unterschied: Ein Assistent schlägt Code vor, den du einfügst. Ein Agent schreibt Code, führt ihn aus, liest Fehlermeldungen, fixt Bugs und commitet – alles autonom. Du gibst die Richtung vor. Der Agent erledigt die Schrittarbeit.

Die Landschaft (Stand März 2026)

Tier 1: Agentic-First

Tool	Modell	Stärke	Preis
Claude Code (Anthropic)	Claude Opus 4 / Sonnet 4	Terminal-Agent, plant & handelt autonom, Git-integriert	API-basiert (pay per token)
Cursor	Multi-Model (Claude, GPT, Gemini)	IDE mit Agent-Mode, Composer, multi-file Edits	$20/Monat (Pro)
Windsurf (Codeium)	Multi-Model	Cascade-Flow, kontextuelles Verständnis	$15/Monat (Pro)
Devin (Cognition)	Proprietär	Vollautonomer Agent, eigene Sandbox	Enterprise

Tier 2: Assistenten mit Agent-Fähigkeiten

Tool	Modell	Stärke	Preis
GitHub Copilot	GPT-4o + Claude	IDE-integriert, Agent-Mode (Preview), Workspace-Kontext	$10-$39/Monat
Cline	Multi-Model (via API)	Open Source, VS-Code-Extension, volle Kontrolle	Kostenlos (+ API-Kosten)
Aider	Multi-Model (via API)	Terminal-basiert, Git-integriert, diff-basiert	Kostenlos (+ API-Kosten)
Continue	Multi-Model	Open Source, IDE-Extension, anpassbar	Kostenlos (+ API-Kosten)

Tier 3: Spezialisiert

Tool	Fokus	Stärke
Bolt.new	Full-Stack Web-Apps	Generiert komplette Apps im Browser
v0 (Vercel)	Frontend/UI	React-Komponenten aus Beschreibungen
Lovable	Full-Stack	Produkt-Building für Nicht-Entwickler
Replit Agent	Full-Stack	Deployment inklusive

Wie du in Coding-Agenten optimal promptest

Claude Code

Claude Code ist ein Terminal-basierter Agent, der direkt in deinem Repo arbeitet:

```
# Starten
claude

# In Claude Code prompten:
"Lies die README und erkläre mir die Architektur dieses
Projekts."

"Implementiere eine REST-API für User-Management:
- CRUD-Endpoints (GET, POST, PUT, DELETE)
- Zod-Validierung
- Prisma ORM
- Schreibe Tests mit Vitest
- Committe am Ende mit aussagekräftiger Commit-Message."
```

Best Practices für Claude Code:

1. **CLAUDE.md nutzen** – Erstelle eine `CLAUDE.md`-Datei im Root deines Projekts mit Architektur, Konventionen und Befehlen. Claude Code liest sie automatisch.
2. **Kleine, klare Aufgaben** – "Implementiere Feature X" ist besser als "Baue die ganze App".
3. **Lass ihn planen** – "Erstelle zuerst einen Plan, bevor du implementierst."
4. **Tests zuerst** – "Schreibe zuerst die Tests, dann die Implementierung."
5. **Git nutzen** – Claude Code commitet automatisch. Prüfe die Commits.

CLAUDE.md – Das Projektgedächtnis

Die wichtigste Datei für agentic Coding:

```
# Projektname

## Architektur
- Monorepo mit Turborepo
- Backend: Node.js + Express + TypeScript
- Frontend: Next.js 15 + React 19
- DB: PostgreSQL + Prisma

## Befehle
- `npm run build` – Alles bauen
- `npm run test` – Tests ausführen
- `npm run lint` – Linting
- `npm run dev` – Entwicklungsserver

## Konventionen
- Alle Dateien: kebab-case
- Alle Funktionen: camelCase
- Alle Typen: PascalCase
- Error Handling: Result-Pattern, kein throw
- Commits: Conventional Commits (feat:, fix:, chore:)
- Branch-Naming: feature/kurze-beschreibung

## Vermeide
- any-Types
- console.log in Production-Code
- Star-Imports
- Class Components (nur Functional Components)
```

Cursor

Cursor hat drei Modi: Tab-Completion, Chat und Composer (Agent).

Tab-Completion:
Cursor vervollständigt Code inline. Optimiere durch:
- Gute Variablennamen (die Completion wird besser)
- Kommentare als Hints: `// TODO: Validate email format`
- Typ-Annotationen (TypeScript/Python Type Hints)

Chat (Cmd+L):

```
Kontext: Markiere relevanten Code, dann frage:
"Erkläre, warum dieser Code bei mehr als 1000 Items
langsam wird und schlage eine Optimierung vor."
```

Composer / Agent Mode (Cmd+I):

```
"Erstelle eine neue API-Route /api/v2/users mit:
- Pagination (cursor-based)
- Filtering (nach Name, Email, Status)
- Sorting (nach jedem Feld, ASC/DESC)
- Rate Limiting (100 req/min)
Nutze die bestehende Middleware aus src/middleware/.
Schreibe Tests."
```

Cursor Rules (.cursorrules):

```
# .cursorrules
You are a senior TypeScript developer.
Always use strict TypeScript - no 'any'.
Use Zod for runtime validation.
Follow the repository's existing patterns.
Write tests for every new function.
Use descriptive variable names.
Prefer composition over inheritance.
```

GitHub Copilot

Copilot hat sich von einem Autocomplete-Tool zu einem Agent entwickelt:

Copilot Chat (in VS Code):

```
@workspace Wie ist die Authentifizierung in diesem
Projekt implementiert?

@workspace Schreibe eine Migration, die eine
"notifications"-Tabelle hinzufügt.
```

Copilot Agent Mode:

```
Implementiere folgendes GitHub Issue: #123
Lies das Issue, verstehe die Anforderungen,
implementiere, teste und erstelle einen PR.
```

Cline (Open Source)

Cline ist der Open-Source-Champion: Volle Kontrolle, jedes Modell, keine Vendor-Lock-in.

```
# In Cline (VS Code Extension):
"Analysiere die Performance dieses Projekts:
1. Lies alle API-Routes
2. Identifiziere N+1 Query-Probleme
3. Schlage Optimierungen vor
4. Implementiere die wichtigste Optimierung
5. Schreibe einen Benchmark-Test"
```

Cline Best Practices:
- `.clinerules` für Projekt-Konventionen
- Auto-approve für sichere Operationen (read, list)
- Manual-approve für Schreiboperationen
- API-Budget setzen (Token-Limit pro Task)

Aider

Terminal-basiert, Git-integriert, minimal:

```
# Starten mit einem Modell
aider --model claude-sonnet-4-20250514

# In Aider:
/add src/api/users.ts src/models/user.ts
> Füge Pagination zur Users-API hinzu.
> Nutze cursor-based Pagination mit Prisma.
```

Aider-Stärken:
- Versteht Git nativ (zeigt Diffs, commitet)
- Funktioniert mit jedem Modell via API
- Lightweight, keine IDE nötig
- Architect-Modus: Erst planen (mit starkem Modell), dann implementieren (mit schnellem Modell)

Welches Tool für welchen Zweck?

Situation	Empfehlung
Kleine Änderungen, quick Fixes	Cursor Tab-Completion, Copilot
Neue Features (multi-file)	Cursor Composer, Claude Code
Große Refactorings	Claude Code, Aider
Code verstehen (onboarding)	Claude Code, Cursor Chat
Prototyping (0 → 1)	Bolt.new, v0, Cursor Composer
Open Source / Budget	Cline, Aider, Continue
Enterprise / Compliance	GitHub Copilot Enterprise, Cursor Business

Effektive Strategien für alle Tools

1. Kontext ist King

Alle Tools profitieren von Kontext. Je mehr relevante Dateien du referenzierst oder in den Kontext gibst, desto besser das Ergebnis.

2. Inkrementell arbeiten

Nicht “baue mir die ganze App” – sondern Schritt für Schritt. Ein Feature nach dem anderen. Ein Modul nach dem anderen.

3. Tests als Qualitätsgate

Lass den Agent Tests schreiben. Dann: Tests ausführen. Wenn sie rot sind: Agent fixen lassen. Die Feedback-Loop ist der Schlüssel.

4. Reviewe alles

Auch agentic-generierter Code muss reviewed werden. Git Diff lesen. Verstehen, was geändert wurde. Nicht blind mergen.

5. Versionskontrolle als Safety Net

Git ist dein Undo-Button. Commite oft. Branch für experimentelle Änderungen. Wenn der Agent Mist baut: `git checkout .`

Übungen

Übung 1: CLAUDE.md erstellen

Erstelle eine CLAUDE.md für ein eigenes Projekt (oder ein Open-Source-Projekt, das du kennst). Welche Informationen braucht ein KI-Agent, um guten Code zu schreiben?

Übung 2: Tool-Vergleich

Nimm dieselbe Aufgabe und löse sie mit zwei verschiedenen Tools (z.B. Cursor + Aider oder Copilot + Claude Code). Vergleiche die Ergebnisse und den Workflow.

Übung 3: Agent-Modus testen

Nutze den Agent-Modus eines Tools (Cursor Composer, Claude Code, Copilot Agent) für ein echtes Feature. Wie viel musstest du manuell nacharbeiten?

Übung 4: Fehlerbehebung

Gib einem Coding-Agent einen Bug-Report und lass ihn autonom debuggen. Wie weit kommt er ohne dein Eingreifen?

Kapitel 3: Die LLM-APIs – Dein Zugang zur KI-Power

Bisher hast du KI über Chat-Interfaces genutzt. Jetzt nutzt du sie programmatisch – über APIs. Das bedeutet: Du schickst HTTP-Requests und bekommst strukturierte Responses. Automatisiert. Skalierbar. Integriert in deine Software.

Die großen Drei (und die Challenger)

Anthropic (Claude)

Modell	Kontext	Stärke	Input / Output (pro 1M Token)
Claude Opus 4.6	200K (1M beta)	#1 Reasoning, 80.8% SWE-bench, Code	$5 / $25
Claude Sonnet 4.5	200K	Bestes Preis-Leistung, 77-82% SWE-bench	$3 / $15
Claude Haiku 4.5	200K	Schnell, günstig, agentic Loops	$0.25 / $1.25

Besondere Features:
- **Extended Thinking** – Das Modell "denkt nach" bevor es antwortet. Steuerbar über `effort`-Parameter. Thinking-Tokens als Output abgerechnet.

- **Tool Use** – Natives Function Calling mit JSON-Schema. Plus: Web Search, Code Execution, Computer Use, programmatic Tool Calling.
- **Prompt Caching** – Wiederholt genutzte Prompt-Teile werden gecacht (90% günstiger).
- **Batches API** – Massenhaft Requests mit 50% Rabatt (24h SLA).
- **Vision** – Bilder, PDFs, Charts, Diagramme als Input.
- **Compaction API** (beta) – Server-seitige Kontext-Zusammenfassung für endlose Konversationen.
- **1M Token Context Window** (beta für Opus 4.6).

OpenAI (GPT)

Modell	Kontext	Stärke	Input / Output (pro 1M Token)
GPT-5.2 (xhigh)	1M	#1 Coding-Benchmarks (89% LiveCodeBench)	$1.75 / $14
GPT-5 mini	200K	Schnell, günstig, starkes Allround	$0.25 / $2
GPT-5 nano	128K	Ultra-günstig, Edge-Deployment	$0.05 / $0.40
GPT-4o	128K	Multimodal (Text + Bild + Audio)	$2.50 / $10
o3	200K	Deep Reasoning, Mathematik	$10 / $40

Besondere Features:
- **Responses API** – Ersetzt Chat Completions für neue Projekte. Agentic by default.
- **Structured Outputs** – Garantiert valides JSON mit `strict: true`.

- **Agents SDK** – Open-Source Multi-Agent-Orchestrierung mit MCP-Support.
- **Realtime API** – Audio-Input und -Output in Echtzeit.

Google (Gemini)

Modell	Kontext	Stärke	Input / Output (pro 1M Token)
Gemini 3.1 Pro	1M	Reasoning, Cross-Language	$2 / $12
Gemini 3 Flash	1M	Schnell, günstig	$0.50 / $3
Gemini 2.5 Pro	1M	Bewährt, riesiger Kontext	$1.25 / $10
Gemini 2.0 Flash-Lite	1M	Ultra-günstig	$0.075 / $0.30

Besondere Features: 1M Token Kontext, großzügigstes Free Tier (1.000 Anfragen/Tag kostenlos), Code Execution in Sandbox, Thinking Budget, nativer MCP-Support.

Open Source / Alternative

Modell	Parameter	Stärke	Zugang
Llama 4 (Meta)	Scout 17B aktiv / Maverick 400B	Multimodal, Open Source	Lokal / Together / Groq
DeepSeek-V3.2	685B (37B aktiv)	Coding, Reasoning, $0.28/1M In	API / Lokal
Qwen 3	Diverse	Multimodal, mehrsprachig	API / Lokal
Mistral Large 2	123B	Europa, mehrsprachig	API / Lokal

Dein erster API-Call

Die Grundstruktur ist bei allen Anbietern ähnlich: Client erstellen, Modell wählen, Nachricht senden:

```
import anthropic

client = anthropic.Anthropic()  # ANTHROPIC_API_KEY aus Env
message = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    messages=[{"role": "user", "content": "Was ist eine
API?"}]
)
print(message.content[0].text)
```

Bei OpenAI sieht es fast identisch aus – `from openai import OpenAI`, `client.chat.completions.create()`, Ergebnis in `response.choices[0].message.content`. Die SDKs für TypeScript folgen demselben Muster.

System-Prompts

System-Prompts definieren das Verhalten des Modells über die gesamte Konversation. Bei Anthropic ein `system`-Parameter, bei OpenAI eine Message mit `role: "system"`:

```
message = client.messages.create(
    model="claude-sonnet-4-6",
    system="Du bist ein Senior Python Developer. PEP 8,
typisiert, Tests.",
    messages=[{"role": "user", "content": "Implementiere
Retry-Logik."}]
)
```

Streaming

Für UX-freundliche Responses – Text erscheint Wort für Wort. Bei Anthropic: `client.messages.stream()` mit einem Context Manager und `stream.text_stream`. Bei OpenAI: `stream=True` als Parameter und Iteration über `chunk.choices[0].delta.content`.

Structured Output

Das Modell gibt garantiert valides JSON zurück. Bei Anthropic nutzt du **Tool Use** – du definierst ein JSON-Schema als Tool, und das Ergebnis kommt als validiertes JSON:

```
message = client.messages.create(
    model="claude-sonnet-4-6",
    tools=[{
        "name": "analyze_sentiment",
        "description": "Analysiere das Sentiment",
        "input_schema": {
            "type": "object",
            "properties": {
                "sentiment": {"type": "string", "enum": ["p
ositiv", "neutral", "negativ"]},
                "confidence": {"type": "number"}
            },
            "required": ["sentiment", "confidence"]
        }
    }],
    tool_choice={"type": "tool", "name": "analy-
ze_sentiment"},
    messages=[{"role": "user", "content": "Analysiere: 'Su-
per Produkt, aber zu teuer.'"}]
)
result = message.content[0].input  # Valides JSON nach
Schema
```

Bei OpenAI: `response_format` mit `type: "json_schema"` und `strict: true`.

Prompt Caching

Spart bis zu 90% bei wiederholten Prompts. Bei Anthropic fügst du `cache_control: {"type": "ephemeral"}` zu System-Prompt-Blöcken hinzu. Beim zweiten Call mit demselben System-Prompt trifft der Cache – 90% günstiger für den gecachten Teil.

Vision (Bilder als Input)

Bilder als Base64 oder URL in den Content-Array einfügen. Das Modell kann Screenshots, Diagramme, PDFs analysieren und z.B. HTML/CSS-Code daraus generieren.

Kosten optimieren

Fünf Strategien, die zusammen bis zu 95% sparen:

1. **Richtiges Modell:** Einfache Aufgaben → Haiku/nano, Mittlere → Sonnet/mini, Komplexe → Opus/o3
2. **Prompt Caching:** 90% Ersparnis auf wiederholt genutzte Kontexte
3. **Batches:** Nicht zeitkritische Anfragen sammeln, 50% Rabatt
4. **Token-Budget:** `max_tokens` auf das Minimum setzen
5. **Prompt-Optimierung:** Kürzere Prompts = weniger Input-Tokens

Preisentwicklung: LLM-API-Preise sind zwischen Anfang 2025 und Anfang 2026 um ca. 80% gefallen. Claude Opus 4.6 kostet 67% weniger als sein Vorgänger.

Übungen

Übung 1: Erster API-Call

Erstelle einen API-Key bei Anthropic oder OpenAI. Schreibe deinen ersten API-Call in Python oder TypeScript.

Übung 2: System-Prompt

Schreibe einen System-Prompt für einen spezialisierten Assistenten. Teste ihn mit verschiedenen User-Messages.

Übung 3: Structured Output

Nutze Tool Use, um das Sentiment von 10 Produktbewertungen als JSON zu analysieren.

Übung 4: Kosten berechnen

Schätze die API-Kosten für 1.000 Anfragen/Tag bei ~500 Input + ~200 Output Tokens. Vergleiche Claude Sonnet, GPT-5 mini und Gemini Flash.

Kapitel 4: Programmatisches Prompting – Prompts als Code

In Kapitel 3 hast du API-Calls gemacht. Einzelne Requests, einzelne Antworten. Jetzt bauen wir Systeme: Prompt-Templates, Variablen, Pipelines, Fehlerbehandlung und Multi-Turn-Konversationen.

Prompt-Templates

Hartcodierte Prompts sind wie hartcodierte Strings – sie funktionieren, aber sie skalieren nicht. Die Lösung: Templates mit Variablen.

Die einfachste Variante sind Python f-Strings mit einer Funktion, die `text`, `language` und `detail_level` als Parameter nimmt. Für mehr Struktur nutzt du eine `PromptTemplate`-Dataclass mit `system`, `user_template`, `model`, `max_tokens` und `temperature`. Die `render`-Methode füllt die Platzhalter:

```
@dataclass
class PromptTemplate:
    system: str
    user_template: str
    model: str = "claude-sonnet-4-6"
    max_tokens: int = 1024

    def render(self, **kwargs) -> str:
        return self.user_template.format(**kwargs)
```

Du definierst Templates einmal (z.B. `SENTIMENT_TEMPLATE`) und nutzt sie überall mit `.render(count=3, texts="...")`.

Multi-Turn-Konversationen

LLM-APIs sind zustandslos. Für Konversationen verwaltest du die History selbst. Eine `Conversation`-Klasse speichert die `messages`-Liste und hängt bei jedem `send()` die User-Message an, macht den API-Call und speichert die Antwort:

```
class Conversation:
    def __init__(self, client, system, model="claude-son-
net-4-6"):
        self.client = client
        self.system = system
        self.messages = []

    def send(self, user_message: str) -> str:
        self.messages.append({"role": "user", "content": us
er_message})
        response = self.client.messages.create(
            model=self.model, system=self.system,
            messages=self.messages, max_tokens=2048
        )
        text = response.content[0].text
        self.messages.append({"role": "assistant", "con-
tent": text})
        return text
```

So baut jede Antwort auf dem vorherigen Kontext auf.

Fehlerbehandlung und Retry-Logik

APIs können fehlschlagen: Rate Limits, Timeouts, Server-Fehler. Robuster Code fängt das ab mit **Exponential Backoff** – 1s, 2s, 4s Wartezeit bei `RateLimitError` und `APITimeoutError`. Server-Fehler (500+) werden retried, Client-Fehler (400, 401) nicht. Nach `max_retries` Versuchen: Exception werfen.

Batch-Verarbeitung

Wenn du 1.000 Items verarbeiten musst: `asyncio` mit Semaphore für kontrollierte Parallelität. Ein `AsyncAnthropic`-Client, ein Semaphore mit `concurrency=5-10`, und `asyncio.gather()` für parallele Verarbeitung. Haiku ist hier ideal – schnell und günstig.

Antwort-Parsing

Zwei Ansätze für strukturierte Antworten:

JSON-Parsing aus Freitext: Ein Fallback-System – erst `json.loads()` direkt, dann nach JSON-Block in Markdown suchen, dann nach `{...}` im Text. Funktioniert, ist aber fragil.

Besser: Structured Output (siehe Kapitel 3). Tool Use oder Structured Outputs geben garantiert valides JSON zurück – kein Parsing nötig.

Prompt-Chaining programmatisch

Mehrere LLM-Calls hintereinander, wobei der Output des einen der Input des nächsten ist. Beispiel: Recherche → Outline → Artikel. Drei `async`-Calls nacheinander, jeder nutzt das Ergebnis des vorherigen als Kontext.

Das Muster ist immer gleich:

1. **Schritt 1:** Breite Recherche (LLM generiert Aspekte)
2. **Schritt 2:** Struktur (LLM erstellt Outline basierend auf Schritt 1)
3. **Schritt 3:** Ausführung (LLM schreibt basierend auf Schritt 2)

Evaluierung und Monitoring

Prompt-Qualität messen

Ein `EvalResult`-Dataclass speichert Input, Expected, Actual, Correct-Flag, Latenz und Token-Verbrauch. Eine `evaluate_prompt`-Funktion iteriert über Testfälle, misst Latenz und Accuracy:

```
Accuracy: 87.5% | Avg Latency: 234ms
```

Mindestens 20 Testfälle pro Template. Bei Änderungen: Regression testen.

Best Practices

1. **Prompt und Code trennen** – Prompts in eigene Dateien (`.txt`, `.md`, YAML), nicht inline
2. **Versionierung** – Prompts in Git versionieren, A/B-Testing ermöglichen
3. **Logging** – Jeden API-Call loggen: Input, Output, Tokens, Latenz, Modell, Timestamp
4. **Testfälle pflegen** – Mindestens 20 pro Template
5. **Graceful Degradation** – Bei API-Ausfall: Fallback auf Cache oder günstigeres Modell

Übungen

Übung 1: Template-Bibliothek

Erstelle eine PromptTemplate-Klasse und 3 Templates für verschiedene Aufgaben (Sentiment, Zusammenfassung, Übersetzung).

Übung 2: Konversations-Manager

Implementiere eine Conversation-Klasse mit History, Token-Counting und Max-History-Länge.

Übung 3: Batch-Verarbeitung

Verarbeite 50 Texte parallel mit asyncio. Miss die Gesamtdauer vs. sequentielle Verarbeitung.

Übung 4: Evaluierung

Erstelle 10 Testfälle für ein Sentiment-Analyse-Template. Wie hoch ist die Accuracy?

Kapitel 5: RAG – Retrieval Augmented Generation

Das wichtigste Pattern für professionelle KI-Anwendungen. Punkt.

RAG löst das fundamentale Problem von LLMs: Sie wissen nur, was in ihren Trainingsdaten steht. Dein Unternehmens-Wiki? Deine Produktdokumentation? Deine Verträge? Davon weiß das Modell nichts.

RAG ändert das. Du reichst dem Modell die relevanten Informationen zusammen mit der Frage – und es antwortet basierend auf DEINEN Daten.

Wie RAG funktioniert

Drei Phasen:

1. **Indexierung** (einmalig): Dokumente → Chunks → Embeddings → Vektor-DB
2. **Abfrage** (bei jeder Frage): User-Frage → Embedding → Ähnlichkeitssuche → Top-K Chunks
3. **Generation** (bei jeder Frage): System-Prompt + Chunks + Frage → LLM → Antwort

Organisationen berichten von 60-80% weniger Halluzinationen und 3x besserer Antwortgenauigkeit bei domain-spezifischen Fragen.

Schritt 1: Dokumente chunken

Chunking ist die wichtigste Entscheidung im RAG-Stack. **80% aller RAG-Fehler liegen im Chunking-Layer, nicht beim LLM.**

Chunking-Strategien

Strategie	Beschreibung	Wann nutzen
Fixed Size	Alle 500 Tokens schneiden	Einfach, aber naiv
Recursive	Versuche große Teiler (§, \n\n, \n, .)	**Standard-Empfehlung**
Semantic	Split bei Themenwechsel per Embedding	Beste Qualität, teuerste Berechnung
Document-aware	Markdown-Header, HTML-Tags	Ideal für strukturierte Formate

Empfohlener Standard: Recursive Character Splitting, **512 Tokens**, 50-100 Tokens Overlap. Erzielte 69% Accuracy im größten Real-Document-Test 2026.

Häufige Fehler:
- Zu kleine Chunks (unter 100 Tokens) verlieren Kontext
- Zu große Chunks (über 2.500 Tokens) verwässern die Relevanz
- Semantisches Chunking klingt besser als es ist – rechtfertigt selten die höheren Kosten

Die Implementierung ist einfach: `RecursiveCharacterTextSplitter` aus LangChain mit `chunk_size`, `chunk_overlap` und einer Liste von Separatoren. Metadata pro Chunk hinzufügen (Quelle, Seite, Abschnitt).

Schritt 2: Embeddings erstellen

Embeddings wandeln Text in Vektoren um – mathematische Repräsentationen der Bedeutung. Ähnliche Texte haben ähnliche Vektoren.

Embedding-Modelle (Stand 2026)

Modell	Anbieter	Stärke	Kosten
`text-embedding-3-large`	OpenAI	Bestes Allround	$0.13 / 1M Token
`voyage-3-large`	Voyage AI	#1 MTEB Retrieval	$0.18 / 1M Token
Gemini Embedding 2	Google	#1 ELO-Ranking	Kostenlos (Free Tier)
`embed-v4.0`	Cohere	Multilingual stark	$0.10 / 1M Token
BGE-M3	Open Source	Bestes Budget/ Self-Hosted	Kostenlos (lokal)

Kritisch: Query und Chunks müssen das gleiche Embedding-Modell verwenden. Modell-Upgrades erfordern Re-Embedding des gesamten Corpus.

Die Implementierung: OpenAI-Client, `client.embeddings.create(model="text-embedding-3-small", input=texts)` – gibt eine Liste von Vektoren zurück.

Schritt 3: Vektor-Datenbank

DB	Typ	Stärke	Wann nutzen
Pinecone	Cloud	Sub-10ms Latenz, SOC2/HIPAA	Enterprise, Skalierung
Weaviate	Cloud + Self-hosted	Hybrid-Suche, Multi-Modal	Wenn Hybrid Search wichtig
Qdrant	Cloud + Self-hosted	Performance (Rust), Filter	Hohe Anforderungen an Filter
ChromaDB	Lokal / Embedded	Einfachster Start	**Prototypen**
pgvector	PostgreSQL Extension	In bestehende DB integriert	Wenn du Postgres hast

Für den Start: ChromaDB. Collection erstellen, Dokumente mit `collection.add()` hinzufügen, mit `collection.query()` suchen. Drei Zeilen Setup, sofort einsatzbereit.

Für Produktion: pgvector (75% günstiger als Pinecone bei bestehender Postgres-Infrastruktur) oder Weaviate (wenn Hybrid Search nötig).

Schritt 4: RAG-Pipeline zusammenbauen

Die Pipeline in vier Schritten:

1. Frage embedden
2. Ähnliche Chunks finden (Top-K, typisch 3-5)
3. Kontext zusammenbauen (Chunks als String)
4. LLM mit System-Prompt + Kontext + Frage befragen

Der System-Prompt ist entscheidend: *"Beantworte Fragen basierend auf dem bereitgestellten Kontext. Wenn die Antwort nicht im Kontext steht, sag das ehrlich. Zitiere relevante Stellen."*

Fortgeschrittene RAG-Techniken

Hybrid Search (Vektor + Keyword)

Pflicht für Produktion. Reine Vektor-Suche verpasst exakte Treffer (z.B. "RFC 7231"). BM25 verpasst semantische Queries. Produktion braucht beides. Weaviate bietet das nativ mit einem `alpha`-Parameter (0 = nur Keyword, 1 = nur Vektor).

Reranking

Höchster ROI-Upgrade in RAG. Erst breit suchen (Top 20), dann mit einem Reranker-Modell (z.B. Cohere `rerank-v3.5`) auf Top 5 sortieren. Deutlich bessere Ergebnisqualität.

Query Expansion

Die User-Frage umformulieren für bessere Treffer. Ein schnelles LLM (Haiku) generiert 3 alternative Formulierungen. Suche mit allen Varianten, dedupliziere Ergebnisse.

Contextual Retrieval

Vor dem Embedden: Jedem Chunk einen Kontextsatz voranstellen, der erklärt, wo sich der Chunk im Gesamtdokument einordnet. Bessere Embeddings, bessere Suchergebnisse.

RAG Anti-Patterns

1. **Zu große Chunks** → Modell wird von irrelevanter Information verwirrt
2. **Kein Overlap** → Informationen über Chunk-Grenzen gehen verloren
3. **Blindes Vertrauen** → Modell kann Chunks falsch interpretieren – immer Quellen angeben
4. **Zu viel Kontext** → 20 Chunks à 1000 Tokens = Nadel im Heuhaufen. Top 3-5 reichen

---

Übungen

Übung 1: Einfaches RAG

Nimm 3-5 PDFs, chunke sie, erstelle Embeddings und baue eine RAG-Pipeline mit ChromaDB.

Übung 2: Chunking-Vergleich

Chunke dasselbe Dokument mit verschiedenen Strategien. Vergleiche die Suchergebnisse.

Übung 3: Hybrid Search

Implementiere Hybrid-Suche (Vektor + Keyword). Bei welchen Fragen ist Hybrid besser?

Übung 4: Evaluierung

Erstelle 10 Frage-Antwort-Paare für deine Dokumente. Wie oft gibt das RAG-System die richtige Antwort?

Kapitel 6: Tool Use und Function Calling – KI greift in die echte Welt

Bisher generierte KI Text. Jetzt greift sie in die Welt ein: Datenbanken abfragen, APIs aufrufen, E-Mails senden, Dateien erstellen. Das ist Tool Use – und es verwandelt ein Sprachmodell in einen handlungsfähigen Agenten.

Was ist Tool Use?

Du definierst Funktionen (Tools) mit Schema. Das Modell entscheidet, wann welches Tool aufgerufen wird, und generiert die Parameter. Du führst die Funktion aus und gibst das Ergebnis zurück. Das Modell verarbeitet das Ergebnis und antwortet dem User.

```
User: "Wie ist das Wetter in Berlin?"
      ↓
LLM entscheidet: Tool "get_weather" mit {"city": "Berlin"}
      ↓
Dein Code führt get_weather("Berlin") aus → {"temp": 12}
      ↓
LLM: "In Berlin sind es 12°C und bewölkt."
```

Tool-Definition

Ein Tool besteht aus drei Teilen: **Name**, **Beschreibung** und **Input-Schema** (JSON-Schema). Das sieht so aus:

```
tools = [{
    "name": "get_weather",
    "description": "Ruft das Wetter für eine Stadt ab. "
                   "Nutze dieses Tool wenn der User nach
Wetter fragt.",
    "input_schema": {
        "type": "object",
        "properties": {
            "city": {"type": "string", "description": "Stad
tname"},
            "unit": {"type": "string", "enum": ["celsius",
"fahrenheit"]}
        },
        "required": ["city"]
    }
}]
```

Bei OpenAI heißt das `functions` statt `tools`, die Struktur ist fast identisch. Bei Google `function_declarations`.

Der Tool-Use-Loop

Das Herzstück: Eine Schleife, die so lange läuft, bis das Modell keine Tools mehr aufruft:

1. Sende Message an LLM (mit Tools)
2. Wenn `stop_reason == "end_turn"` → fertig, gib Text zurück
3. Wenn Tool-Call → führe Tool aus, hänge Ergebnis an Messages
4. Zurück zu Schritt 1

Das Modell kann dabei **mehrere Tools gleichzeitig** aufrufen (parallele Tool-Calls). Bei "Vergleiche das Wetter in Berlin, München und Hamburg" generiert es drei `get_weather`-Calls auf einmal.

Tool-Design: Best Practices

1. Klare, präzise Beschreibungen

- **Schlecht:** `"Sucht nach Sachen"`
- **Gut:** `"Durchsucht den Produktkatalog. Gibt Name, Preis und Verfügbarkeit zurück. Nutze dieses Tool wenn der User nach Produkten oder Preisen fragt."`

Die Beschreibung ist das Wichtigste. Das Modell entscheidet anhand der Beschreibung, ob es ein Tool nutzt.

2. Parameter-Beschreibungen mit Beispielen

Nicht nur den Typ angeben, sondern erklären was erwartet wird: `"Suchbegriff. Kann Produktname, Kategorie oder Beschreibung sein. Beispiele: 'rotes Kleid', 'iPhone 16'"`.

3. Enums für begrenzte Optionen

Wenn ein Parameter nur bestimmte Werte annehmen kann, nutze `enum`. Das verhindert ungültige Eingaben und hilft dem Modell.

4. Wenige, mächtige Tools statt vieler kleiner

Statt 10 einzelne Tools (`get_user_name`, `get_user_email`, …) lieber ein flexibles `get_user_info` mit einem `fields`-Array. Weniger Tools = weniger Kontext-Verbrauch = bessere Tool-Auswahl.

Sicherheit bei Tool Use

Bestätigungen für kritische Aktionen

Jedes Tool bekommt eine Sicherheitsstufe:

- **SAFE** (z.B. `search_database`) → automatisch ausführen
- **CONFIRM** (z.B. `send_email`) → User fragen
- **CRITICAL** (z.B. `delete_user`) → Admin-Bestätigung nötig

Input-Validierung

Nie die LLM-generierten Parameter blind an dein System weiterreichen. Validiere mit Pydantic oder eigenen Checks: Keine SQL-Injection-Zeichen in Stadtname, keine Pfadangaben wo keine hingehören, Enum-Werte prüfen.

Tool Use in der Praxis

Ein typischer **Kunden-Support-Bot** hat drei bis fünf Tools:

- `lookup_order` – Bestellung nachschlagen
- `check_inventory` – Verfügbarkeit prüfen
- `create_ticket` – Support-Ticket erstellen

Jedes Tool mit klarem Schema, sinnvollen Defaults und Required-Feldern. Das Modell wählt selbstständig das richtige Tool basierend auf der User-Frage.

Übungen

Übung 1: Eigenes Tool

Definiere ein Tool für einen Anwendungsfall deiner Wahl. Implementiere den Tool-Use-Loop.

Übung 2: Multi-Tool

Erstelle 3 Tools, die zusammenarbeiten (z.B. Suche → Details → Bestellung).

Übung 3: Sicherheit

Implementiere eine Sicherheitsschicht mit Bestätigungen für kritische Tool-Calls.

Übung 4: Parallele Calls

Teste parallele Tool-Calls: Lass das Modell 5 verschiedene Informationen gleichzeitig abrufen.

Kapitel 7: Agentische Systeme – KI, die handelt

Tool Use (Kapitel 6) gibt KI Hände. Agentische Systeme geben ihr einen Plan.

Ein Agent ist ein LLM, das:
1. Ein Ziel bekommt
2. Einen Plan erstellt
3. Tools nutzt, um Schritte auszuführen
4. Ergebnisse bewertet
5. Den Plan anpasst
6. Iteriert, bis das Ziel erreicht ist

Der Unterschied zu einem einfachen Tool-Use-Call: Ein Agent arbeitet autonom über mehrere Schritte. Er entscheidet selbst, welches Tool als Nächstes kommt.

Agent-Architektur

Das Grundmuster ist eine Schleife: **Planen → Handeln → Beobachten → Bewerten → (Wiederholen oder Fertig)**. In der Literatur heißt das **ReAct** (Reason + Act).

```
ZIEL (User-Input)
    ↓
PLANEN (LLM denkt) ←──────┐
    ↓                     │
HANDELN (Tool nutzen)     │
    ↓                     │
BEOBACHTEN (Ergebnis)     │
    ↓                     │
BEWERTEN ── Ziel erreicht? ─┘
    ↓ Ja
ANTWORT
```

Der einfachste Agent: ReAct-Loop

Der ReAct-Loop ist im Kern der Tool-Use-Loop aus Kapitel 6 – nur mit einem System-Prompt, der dem Modell sagt: *"Arbeite Schritt für Schritt. Überlege, was du als Nächstes tun musst. Nutze ein Tool. Beobachte das Ergebnis. Entscheide: fertig oder weitermachen?"*

Plus ein `max_steps`-Parameter als Sicherheitsnetz gegen Endlosschleifen. Bei jedem Schritt wird geloggt, welches Tool aufgerufen wird – so siehst du, wie der Agent "denkt".

MCP – Model Context Protocol

MCP (Model Context Protocol) ist der offene Standard für die Verbindung von Agenten mit externen Systemen. Im November 2024 von Anthropic eingeführt, im Dezember 2025 an die Linux Foundation übergeben. Stand März 2026: 97 Millionen monatliche SDK-Downloads, adoptiert von Anthropic, OpenAI, Google, Microsoft und Amazon.

Was MCP löst

Ohne MCP schreibst du für jedes externe System (Datenbank, GitHub, Slack, Jira) eigenen Integrations-Code. Mit MCP definierst du einmal einen MCP-Server – und jeder MCP-fähige Client (Claude Code, Cursor, deine App) kann ihn nutzen.

MCP-Architektur

Drei Komponenten: **MCP Host** (Claude, Cursor, deine App), **MCP Client** (SDK) und **MCP Server** (dein Code). Der Server exponiert drei Dinge: **Tools** (Funktionen, die der Agent aufrufen kann), **Resources** (Daten, die der Agent lesen kann) und **Prompts** (vordefinierte Prompt-Templates).

MCP vs. Function Calling

MCP-Server halten State über Aufrufe hinweg, bieten dynamische Tool-Discovery (Agent erkennt automatisch verfügbare Tools) und exponieren Resources – das ist reicher als stateless Function Calling.

Einen MCP-Server schreiben

Mit dem Python SDK (`FastMCP`) sind es wenige Zeilen:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("Firmen-Server")

@mcp.tool()
def search_knowledge_base(query: str, max_results: int =
5) -> str:
    """Durchsucht die interne Wissensdatenbank."""
    results = db.search(query, limit=max_results)
    return json.dumps(results)

@mcp.resource("file://docs/{path}")
def read_document(path: str) -> str:
    """Liest ein Dokument aus dem Docs-Verzeichnis."""
    return (Path("docs") / path).read_text()

mcp.run()
```

In der Claude-Code-Konfiguration (`settings.json`) registrierst du den Server mit `command` und `args`. Dann kann der Agent automatisch deine Wissensdatenbank durchsuchen.

Multi-Agent-Systeme

Für komplexe Aufgaben: Mehrere spezialisierte Agenten, die zusammenarbeiten.

Orchestrator-Pattern

Ein Orchestrator analysiert die Aufgabe, erstellt einen Plan mit Teilaufgaben und delegiert an Spezialisten (Research-Agent, Coding-Agent, Review-Agent). Am Ende führt ein Synthese-Agent die Ergebnisse zusammen.

Supervisor-Pattern

Ein Supervisor wählt dynamisch den nächsten Agent und überwacht den Fortschritt. Im Gegensatz zum Orchestrator ist der Plan nicht statisch – der Supervisor entscheidet bei jedem Schritt neu, basierend auf den bisherigen Ergebnissen.

Agent-Frameworks

Framework	Stärke	Ideal für
LangGraph	Graph-basiert, Checkpointing	Komplexe Workflows mit Schleifen
CrewAI	Rollen-basierte Agents, Visual Editor	Schnelles Prototyping
OpenAI Agents SDK	Open-Source, MCP-Support	Moderate Komplexität

Praxis-Empfehlung: Für einfache Workflows (1-2 Tools) brauchst du kein Framework – direkte API-Calls reichen. Framework erst ab 3+ Agents oder komplexen Abläufen.

Agent-Guardrails

Agenten, die autonom handeln, brauchen Sicherheitsmechanismen:

1. Token-Budget

Setze ein maximales Token-Limit pro Aufgabe. Wenn überschritten: Stoppe den Agent und melde den Fehler.

2. Erlaubte Aktionen

Definiere pro Tool eine Sicherheitsstufe: `read_file` immer erlaubt, `write_file` nur mit Bestätigung, `delete_file` nie, `send_email` nie.

3. Sandbox

Agenten sollten in isolierten Umgebungen laufen: Docker-Container, Read-only Dateisysteme, Netzwerk-Beschränkungen, Zeitlimits.

4. Human-in-the-Loop

Für kritische Entscheidungen: Pausiere und frage den Menschen.

Wann Agenten, wann nicht?

Situation	Agent?	Warum
Klar definierte Aufgabe, 1 Schritt	Nein	Einfacher API-Call reicht
Multi-Step mit klarer Reihenfolge	Prompt-Chain	Deterministische Pipeline reicht
Multi-Step mit unklarer Reihenfolge	Ja	Agent entscheidet dynamisch
Offene Recherche-Aufgabe	Ja	Agent iteriert bis zufrieden
Sicherheitskritische Aufgabe	Vorsichtig	Starke Guardrails nötig
Echtzeit / Low-Latency	Nein	Agenten sind langsam (mehrere LLM-Calls)

Übungen

Übung 1: ReAct-Agent

Baue einen einfachen ReAct-Agent mit 2-3 Tools. Lass ihn eine mehrstufige Frage beantworten.

Übung 2: MCP-Server

Schreibe einen MCP-Server für einen Anwendungsfall deiner Wahl (z.B. SQLite-DB oder lokale API).

Übung 3: Guardrails

Implementiere Token-Budget, erlaubte Aktionen und Human-in-the-Loop. Teste: Hält er sich daran?

Übung 4: Multi-Agent

Baue ein 2-Agent-System: Einer recherchiert, der andere schreibt. Koordiniere sie mit einem Orchestrator.

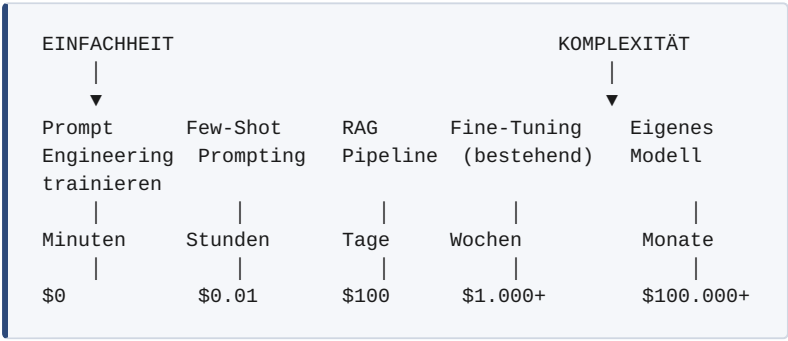
Kapitel 8: Fine-Tuning vs. Prompting – Die Entscheidungsmatrix

“Sollte ich das Modell fine-tunen?” Diese Frage höre ich ständig. Und die Antwort ist fast immer: Nein. Zumindest nicht zuerst.

Fine-Tuning ist mächtig. Aber es ist teuer, komplex und oft unnötig. In den meisten Fällen erreichst du mit gutem Prompting, RAG und Tool Use dasselbe Ergebnis – schneller und günstiger.

Dieses Kapitel zeigt dir, wann welcher Ansatz der richtige ist.

Die Optionen im Überblick



Wann reicht Prompting?

Anforderung	Prompting reicht?
Ton/Stil anpassen	✅ Ja – System-Prompt mit Beispielen
Fachsprache verwenden	✅ Ja – Fachbegriffe im Prompt erklären
Ausgabe-Format steuern	✅ Ja – Structured Output / Tool Use
Auf eigene Daten zugreifen	✅ Ja – RAG
Eigene Tools nutzen	✅ Ja – Function Calling
Komplexe Reasoning-Aufgaben	✅ Ja – Chain-of-Thought, Extended Thinking
95%+ Accuracy bei Klassifikation	⚠️ Vielleicht – Few-Shot oft ausreichend
Spezifische Domäne (Jura, Medizin)	⚠️ Vielleicht – RAG + spezialisierter Prompt
Eigene “Stimme” / Brand Voice	⚠️ Vielleicht – System-Prompt + Beispiele
Extrem hohe Konsistenz	❌ Vielleicht Fine-Tuning nötig
Latenz-kritisch (jede ms zählt)	❌ Fine-Tuning auf kleinem Modell
Offline-Betrieb	❌ Eigenes/lokales Modell nötig

Wann Fine-Tuning?

Fine-Tuning ist sinnvoll, wenn:

1. **Konsistenz über tausende Anfragen** – Du brauchst exakt dasselbe Format, denselben Stil, immer. Nicht “meistens”, sondern “immer”.

2. **Latenz** – Du willst ein kleines, schnelles Modell, das eine spezifische Aufgabe so gut kann wie ein großes.

3. **Kosten bei Volumen** – 1 Million Requests/Tag: Ein fine-getuntes GPT-4o-mini ist günstiger als ein Prompt mit 2000 Token Kontext bei Sonnet.

4. **Spezifisches Verhalten** – Das Modell soll Dinge tun, die schwer per Prompt zu beschreiben sind (z.B. einen sehr spezifischen Coding-Style).

5. **Datenschutz** – Deine Trainingsdaten bleiben bei dir (lokales Fine-Tuning mit Open Source).

Fine-Tuning-Optionen (Stand 2026)

Cloud-basiert

Anbieter	Modell	Methode	Kosten (Training)	Kosten (Inference)
OpenAI	GPT-4o-mini	Supervised FT	$3.00 / 1M Token	$0.30 / 1M Token (out)
OpenAI	GPT-4o	Supervised FT	$25.00 / 1M Token	$3.75 / 1M Token (out)
Google	Gemini Flash	Supervised FT	Variabel	Reduziert vs. Base
Together AI	Llama, Mistral etc.	LoRA, Full FT	Ab $0.50/Stunde GPU	Pay-per-token
Fireworks AI	Diverse Open Source	LoRA	Ab $0.50/Stunde GPU	Pay-per-token

Lokal

Tool	Methode	Stärke
Unsloth	QLoRA	2x schneller, 60% weniger VRAM, Open Source
Axolotl	LoRA, QLoRA, Full FT	Flexibel, viele Modelle
Hugging Face TRL	SFT, DPO, RLHF	Standard-Framework
MLX (Apple)	LoRA	Optimiert für Apple Silicon
Ollama + Modelfile	Basis-Anpassung	Einfachster Einstieg

Fine-Tuning: Der Prozess

Schritt 1: Trainingsdaten erstellen

```
// Format: JSONL (eine Zeile pro Beispiel)
{"messages": [
    {"role": "system", "content": "Du bist ein Support-Bot
für TechCorp."},
    {"role": "user", "content": "Wie setze ich mein Pass-
wort zurück?"},
    {"role": "assistant", "content": "Gehe auf techcorp.de/
passwort, klicke auf 'Passwort vergessen', gib deine E-Mail
ein. Du erhältst innerhalb von 5 Minuten einen Reset-
Link."}
]}
{"messages": [
    {"role": "system", "content": "Du bist ein Support-Bot
für TechCorp."},
    {"role": "user", "content": "Wie kündige ich mein
Abo?"},
    {"role": "assistant", "content":
"Gehe zu Einstellungen > Abo > Kündigen. Die Kündigung wird
zum Ende des aktuellen Abrechnungszeitraums wirksam. Du
behältst bis dahin vollen Zugang."}
]}
```

Wie viele Beispiele?

Aufgabe	Minimum	Empfohlen	Optimum
Stil/Ton anpassen	20	50-100	200+
Klassifikation	50 pro Klasse	100 pro Klasse	500+ pro Klasse
Domänen-spezifisch	100	500	1.000+
Komplexe Aufgaben	200	1.000	5.000+

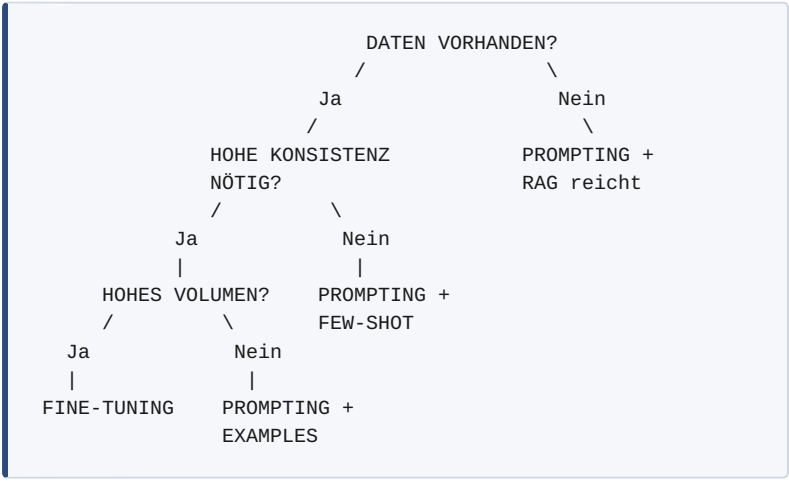
Schritt 2: Datenqualität prüfen

```
# Checkliste für Trainingsdaten:
# ✅ Konsistente Formatierung
# ✅ Keine Widersprüche zwischen Beispielen
# ✅ Repräsentativ für echte Anfragen
# ✅ Korrekte Antworten (manuell geprüft)
# ✅ Diverse Formulierungen (nicht alle gleich)
# ✅ Edge Cases enthalten
# ✅ Keine sensiblen Daten (PII, Passwörter)
```

Schritt 3: Fine-Tuning starten (OpenAI Beispiel)

```
from openai import OpenAI
client = OpenAI()

# Trainingsdatei hochladen
file = client.files.create(
    file=open("training_data.jsonl", "rb"),
    purpose="fine-tune"
)

# Fine-Tuning starten
job = client.fine_tuning.jobs.create(
    training_file=file.id,
    model="gpt-4o-mini-2024-07-18",
    hyperparameters={
        "n_epochs": 3,
        "batch_size": "auto",
        "learning_rate_multiplier": "auto"
    }
)

# Status prüfen
status = client.fine_tuning.jobs.retrieve(job.id)
print(f"Status: {status.status}")
# → "running" → "succeeded"

# Fine-tuned Modell nutzen
response = client.chat.completions.create(
    model=status.fine_tuned_model,  # ft:gpt-4o-
mini:org:custom-name:id
    messages=[{"role": "user", "content": "Wie setze ich
mein Passwort zurück?"}]
)
```

Die Entscheidungsmatrix



Praktische Faustregel

Starte IMMER mit Prompting. Investiere zuerst 2-3 Tage in Prompt-Optimierung, bevor du über Fine-Tuning nachdenkst. In 90% der Fälle wirst du feststellen: Prompting reicht.

Wenn es nicht reicht, probiere diese Reihenfolge:
1. Besserer System-Prompt
2. Few-Shot-Beispiele im Prompt
3. RAG (eigene Daten)
4. Prompt-Chaining
5. Fine-Tuning (wenn 1-4 nicht reichen)

Prompt-Distillation: Der goldene Mittelweg

Eine clevere Technik: Nutze ein großes, teures Modell (Opus) um hochqualitative Antworten zu generieren. Nutze diese Antworten als Trainingsdaten für ein kleines, günstiges Modell (Haiku, GPT-4o-mini).

```
# Phase 1: Großes Modell generiert Trainingsbeispiele
for query in production_queries:
    response = opus_model.generate(query)  # Teuer, lang-
sam, aber gut
    training_data.append({"input": query, "output": respon-
se})

# Phase 2: Kleines Modell wird auf diesen Daten fine-tuned
fine_tune(small_model, training_data)

# Phase 3: Kleines Modell in Produktion (günstig, schnell)
result = fine_tuned_small_model.generate(new_query)
```

Ergebnis: 90% der Qualität des großen Modells bei 10% der Kosten und 5x der Geschwindigkeit.

Übungen

Übung 1: Prompt vs. Fine-Tuning

Wähle eine Aufgabe (z.B. E-Mail-Klassifikation) und löse sie mit Few-Shot-Prompting. Wie hoch ist die Accuracy? Würde Fine-Tuning helfen?

Übung 2: Trainingsdaten erstellen

Erstelle 20 Trainingsdaten-Paare für eine Aufgabe deiner Wahl. Prüfe sie auf Konsistenz und Qualität.

Übung 3: Kosten-Analyse

Berechne die Kosten für: 100.000 Anfragen/Monat, einmal mit Sonnet (mit langem System-Prompt) und einmal mit fine-tuned GPT-4o-mini (ohne System-Prompt). Was ist günstiger?

Übung 4: Distillation

Wenn du API-Zugang hast: Generiere 50 Antworten mit einem großen Modell. Nutze sie als Few-Shot-Beispiele für ein kleines Modell. Wie nah kommt das kleine Modell?

Kapitel 9: Context Engineering – Die Evolution des Prompt Engineering

Prompt Engineering war Band 1. Context Engineering ist Band 7.

Der Unterschied: Prompt Engineering optimiert den Text, den du an ein LLM schickst. Context Engineering optimiert *alles*, was das Modell sieht – System-Prompt, Tools, Retrieval-Ergebnisse, Konversationshistorie, Caching, und wie all diese Teile zusammenspielen.

Andrej Karpathy brachte es auf den Punkt: *"Das LLM ist eine CPU, das Context Window ist RAM, und dein Job ist es, das Betriebssystem zu sein – den Arbeitsspeicher mit genau dem richtigen Code und den richtigen Daten für jede Aufgabe zu laden."*

Was ist Context Engineering?

Jeder API-Call an ein LLM hat fünf Kontextschichten:

1. **System-Prompt** – Rolle, Verhalten, Regeln, Output-Format
2. **Tool-Definitionen** – Verfügbare Tools mit Beschreibungen
3. **Retrieval-Kontext (RAG)** – Relevante Dokumente und Suchergebnisse
4. **Konversationshistorie** – Bisherige Messages und Tool-Ergebnisse
5. **Aktuelle Nachricht** – Der User-Prompt

Context Engineering optimiert jede dieser Schichten – und ihr Zusammenspiel.

Schicht 1: System-Prompt Engineering

Die Anatomie eines Produktions-System-Prompts

Ein produktionsreifer System-Prompt hat fünf Abschnitte:

- **Rolle:** Wer bist du, für welches Unternehmen, welche Expertise
- **Verhalten:** Sprache, Ton, Stil, Verbote
- **Fähigkeiten:** Welche Tools, was kannst du, was NICHT
- **Dynamischer Kontext:** Datum, User-Info, aktuelle Promotions, bekannte Probleme (per Variable eingefügt)
- **Output-Format:** Länge, Struktur, Formatierung

System-Prompt-Patterns

Identity + Constraints: Wer bist du → Was kannst du → Was NICHT.

Dynamic Context: Kontext, der sich bei jedem Call ändert – Datum, User-Name, Plan, offene Tickets. Per f-String oder Template eingefügt.

Graduated Response: Für einfache Fragen kurz antworten, für komplexe strukturiert, für unbekannte weiterleiten.

Schicht 2: Tool-Kontext-Optimierung

Die Beschreibungen deiner Tools sind Teil des Kontexts – und sie beeinflussen stark, wie gut das Modell die Tools nutzt.

- **Schlecht:** `"Sucht Dinge"`

- **Gut:** `"Durchsucht die interne Wissensdatenbank von TechCorp. Enthält: Produktdokumentation, FAQ, Troubleshooting. Enthält NICHT: Kundendaten, Finanzdaten. Nutze dieses Tool BEVOR du Produktfragen beantwortest."`

Die Investition in gute Tool-Beschreibungen zahlt sich mehr aus als fast jede andere Optimierung.

Schicht 3: RAG-Kontext-Optimierung

Kontext-Reihenfolge

LLMs haben ein "Lost in the Middle"-Problem: Accuracy ist am höchsten, wenn relevante Information am **Anfang** oder **Ende** des Kontexts steht. Über 30% Accuracy-Verlust für Informationen in der Mitte. Strategie: Relevantestes an den Anfang UND ans Ende.

Kontext-Kompression

Zu viel Kontext ist schlechter als zu wenig. Forschung zeigt, dass LLM-Reasoning ab ~3.000 Tokens degradiert. Sweet Spot für Prompts: 150-300 Wörter. Bei langen Dokumenten: Vorher mit einem schnellen Modell (Haiku) zusammenfassen.

Schicht 4: Konversationshistorie managen

Lange Konversationen sprengen das Kontext-Fenster. Zwei Strategien:

Trimmen: Älteste Messages entfernen, System-Prompt behalten. Einfach, aber Kontext geht verloren.

Zusammenfassen: Alte Messages durch eine Zusammenfassung ersetzen. Behält den Kontext, braucht einen Extra-LLM-Call.

Die beste Lösung ist oft eine Kombination: System-Prompt + Zusammenfassung der älteren History + die letzten 4-6 Messages vollständig.

Anthropics **Compaction API** (beta) macht das serverseitig – du gibst ihr die volle History, sie gibt dir eine komprimierte Version zurück.

Schicht 5: Prompt Caching

Anthropics Prompt Caching spart bis zu 90% auf wiederholt genutzte Kontexte. Alles, was sich zwischen Calls nicht ändert (System-Prompt, Tool-Definitionen, Konversations-Prefix), markierst du mit `cache_control`. Beim ersten Call wird gecacht, alle weiteren Calls lesen aus dem Cache.

Was cachen?

Kontext-Teil	Cachen?	Warum
System-Prompt	Immer	Ändert sich selten
Tool-Definitionen	Immer	Ändert sich nie
RAG-Dokumente	Manchmal	Nur bei wiederholt gleichen Docs
Konversationshistorie	Ja	Prefix bleibt gleich
User-Prompt	Nein	Ändert sich bei jedem Call

Context Engineering Checkliste

Für jeden produktionsreifen LLM-Call prüfe:

System-Prompt: Rolle klar? Fähigkeiten und Grenzen explizit? Dynamischer Kontext eingefügt? Output-Format spezifiziert? Gecacht?

Tools: Beschreibungen detailliert? Beispiele enthalten? Negative Beispiele (“NICHT für…”)? Gecacht?

RAG: Relevante Chunks (nicht zu viele)? Reranking? Quellen referenzierbar? Reihenfolge optimiert?

History: Getrimmt oder zusammengefasst? Token-Budget eingehalten? Prefix gecacht?

Prompt: Klar und eindeutig? Alle nötigen Infos? Nicht zu lang?

Von Prompt Engineering zu Context Engineering

Prompt Engineering	Context Engineering
“Wie schreibe ich den Prompt?”	“Wie designe ich den gesamten Kontext?”
Einzelner Text-Input	System + Tools + RAG + History + Prompt
Statisch	Dynamisch (pro User, pro Anfrage)
Trial and Error	Systematisch, messbar, versioniert
Ein Skill	Eine Disziplin

Performance-Gewinne kommen zunehmend nicht von besseren Modellen, sondern von smarterem Kontext. Context Engineering ist die Zukunft. Wer nur Prompts schreibt, schöpft 20% des Potenzials aus. Wer den gesamten Kontext designt, schöpft 100% aus.

Übungen

Übung 1: System-Prompt Audit

Nimm einen bestehenden System-Prompt und prüfe ihn gegen die Checkliste. Was fehlt?

Übung 2: Caching implementieren

Implementiere Prompt Caching für einen Anwendungsfall mit langem System-Prompt. Miss den Kostenunterschied.

Übung 3: Konversationsmanager

Baue einen ConversationManager mit Zusammenfassungs-Strategie. Behält das Modell den Kontext?

Übung 4: Full Context Design

Designe den kompletten Kontext für einen produktionsreifen Chatbot: System-Prompt, Tools, RAG, History-Management, Caching.

Kapitel 10: Zusammenfassung und Ausblick

Sieben Bände. Du bist jetzt in der Profi-Liga.

In diesem Band hast du den Sprung gemacht: Vom Chat-Interface zur API. Vom Endnutzer zum Entwickler. Vom einzelnen Prompt zum System.

Was du jetzt kannst

1. **Code-Generierung** – Du schreibst Prompts, die professionellen Code erzeugen: mit Kontext, Architektur, Qualitätsanforderungen und Testabdeckung. Du kennst die Unterschiede zwischen Feature-Implementierung, Bug-Fixing, Refactoring und Code Reviews.

2. **Agentic Coding Tools** – Du kennst die Landschaft: Claude Code, Cursor, GitHub Copilot, Windsurf, Cline, Aider. Du weißt, wie CLAUDE.md und .cursorrules funktionieren. Du promptest innerhalb dieser Tools optimal.

3. **LLM-APIs** – Du beherrschst die APIs von Anthropic, OpenAI und Google. Du kennst Streaming, Structured Output, Vision, Prompt Caching und Batches. Du weißt, was jedes Modell kostet und wann du welches nutzt.

4. **Programmatisches Prompting** – Du baust Prompt-Templates, Multi-Turn-Konversationen, Fehlerbehandlung, Batch-Verarbeitung und Evaluierungs-Pipelines. Mit echtem Code in Python und TypeScript.

5. **RAG** – Du baust vollständige RAG-Pipelines: Chunking, Embeddings, Vektor-Datenbanken, Hybrid Search, Reranking und Contextual Retrieval. Du bringst deine eigenen Daten in die KI.

6. **Tool Use** – Du definierst Tools, implementierst den Tool-Use-Loop, designst sichere Tool-Schemas und baust Sicherheitsschichten. KI greift in die echte Welt.

7. **Agentische Systeme** – Du baust ReAct-Agenten, nutzt MCP, orchestrierst Multi-Agent-Systeme und implementierst Guardrails. KI plant, handelt und iteriert.

8. **Fine-Tuning vs. Prompting** – Du kennst die Entscheidungsmatrix. Du weißt, dass Prompting fast immer ausreicht. Und wenn nicht: Du weißt, wie Fine-Tuning funktioniert und was es kostet.

9. **Context Engineering** – Du designst nicht nur Prompts, sondern komplette Kontexte: System-Prompt, Tools, RAG, History, Caching. Du denkst in Systemen, nicht in Einzelprompts.

Checkliste: Bin ich bereit für Band 8?

- [ ] Ich habe mindestens einen API-Call in Python oder TypeScript geschrieben
- [ ] Ich verstehe den Unterschied zwischen Chat-Interface und API-Nutzung
- [ ] Ich habe ein Coding-Tool (Cursor, Claude Code, Copilot) produktiv eingesetzt
- [ ] Ich habe eine RAG-Pipeline gebaut (oder zumindest verstanden)
- [ ] Ich habe Tool Use / Function Calling implementiert
- [ ] Ich verstehe, wann Fine-Tuning sinnvoll ist (und wann nicht)
- [ ] Ich kenne den Unterschied zwischen Prompt und Context Engineering
- [ ] Ich habe Prompt Caching verstanden und kann es einsetzen

- [ ] Ich habe einen einfachen Agent gebaut (oder den Ablauf verstanden)
- [ ] Ich kann API-Kosten für ein Projekt kalkulieren

Bei 7 von 10? Weiter zu Band 8.

Was dich in Band 8 erwartet

Band 8 heißt “Business & Produktivität” – und bringt alles Bisherige in den Arbeitsalltag.

Workflow-Automatisierung

KI in bestehende Business-Prozesse integrieren. Nicht als Spielerei, sondern als produktives Werkzeug. E-Mail-Verarbeitung, Dokumenten-Workflows, Datenaufbereitung – automatisiert.

Team-Standards für KI

Wie du KI-Nutzung im Team etablierst. Prompt-Libraries, Best Practices, Governance. Von “jeder promptet wie er will” zu “wir haben Standards”.

Berichte und Kommunikation

KI für die Textarbeit im Büroalltag: E-Mails, Berichte, Protokolle, Präsentationen. Nicht generisch, sondern in deinem Ton, für deine Zielgruppe.

ROI von KI messen

Wie du den Nutzen von KI-Integration bezifferst. Zeitersparnis, Qualitätsverbesserung, Kostensenkung – in Zahlen.

Wie die Reihe weitergeht

Anfänger (Band 1–3) ✓
- Band 1: Grundlagen ✓
- Band 2: Prompt-Frameworks ✓
- Band 3: Fortgeschrittene Basics ✓

Fortgeschritten (Band 4–6) ✓
- Band 4: Reasoning-Techniken ✓
- Band 5: Kreatives Prompting ✓
- Band 6: Spezialisiertes Prompting ✓

Profi (Band 7–9)
- Band 7: Prompting für Entwickler ✓ *(du bist hier)*
- Band 8: Business & Produktivität ← *als Nächstes*
- Band 9: Sicherheit & Ethik

Experte (Band 10)
- Band 10: Die Zukunft

Ein Gedanke zum Schluss

Dieser Band war der technischste der Reihe. Code, APIs, Vektor-Datenbanken, Agenten-Architekturen – schwerer Stoff. Aber notwendiger Stoff.

Denn die Zukunft von KI liegt nicht in Chat-Interfaces. Sie liegt in der Integration. KI als Schicht in jeder Software. KI als API-Call in jedem Workflow. KI als Agent, der Aufgaben erledigt.

Wer nur chattet, kratzt an der Oberfläche. Wer programmiert, formt die Technologie.

Und genau das hast du in diesem Band gelernt: KI formen. Nicht als passiver Nutzer, sondern als Entwickler, Architekt und Ingenieur.

In Band 8 bringen wir das in den Business-Alltag. Weniger Code, mehr Strategie. Weniger API-Calls, mehr Workflow-Design. Weniger “Wie funktioniert die Technik?” und mehr “Wie verändert die Technik meine Arbeit?”

Bis dahin: Bau etwas. Der beste Weg, Context Engineering zu lernen, ist Context Engineering zu machen.

Belkis Aslani

Ressourcen und weiterführende Links

APIs und Dokumentation:
- Anthropic API Docs: docs.anthropic.com
- OpenAI API Docs: platform.openai.com/docs
- Google AI Studio: ai.google.dev
- Claude Agent SDK: github.com/anthropics/claude-code

Coding-Tools:
- Cursor: cursor.com
- Claude Code: claude.ai/code
- GitHub Copilot: github.com/features/copilot
- Cline: github.com/cline/cline
- Aider: aider.chat

RAG und Vektor-Datenbanken:
- ChromaDB: docs.trychroma.com
- Pinecone: pinecone.io
- pgvector: github.com/pgvector/pgvector
- LangChain: python.langchain.com

MCP (Model Context Protocol):
- Spezifikation: modelcontextprotocol.io
- MCP Server Registry: github.com/modelcontextprotocol/servers

Fine-Tuning:

- OpenAI Fine-tuning Guide: platform.openai.com/docs/guides/fine-tuning
- Unsloth: github.com/unslothai/unsloth
- Hugging Face TRL: huggingface.co/docs/trl

Open Source Modelle:

- Ollama (lokale Modelle): ollama.com
- Together AI (gehostete Open Source): together.ai
- Hugging Face: huggingface.co