CLAUDE CODE

Claude Code

Praxisanleitung für Entwickler

Stand: Mai 2026

Claude Code v2.1+ · Opus 4.7 · Sonnet 4.6 · Haiku 4.5

Autor Christian Drapatz

Plattform macOS · Apple Silicon · Linux · Windows

Version 1.0

Inhaltsverzeichnis

Einstieg

1. Was Claude Code ist — und was nicht

Claude Code ist ein KI-Agent der direkt im Terminal läuft. Kein Webinterface, kein Copy-Paste-Loop — du arbeitest mit ihm wie mit einem Entwickler der neben dir sitzt.

Was das Tool gut kann: Repositories durchsuchen, Bugs fixen, Code schreiben, Tests ausführen, Refactorings über viele Dateien hinweg. Was es nicht ist: ein Chatbot den du mit Prompts füttert und dem du dann hoffst.

Der Kern-Unterschied zu anderen KI-Tools: Claude Code nutzt lieber `grep`, `git log` und `ls` als KI-Aufrufe. Ein `grep -r "functionName" src/` kostet keine Tokens. Erst wenn Claude wirklich denken muss, kommt das Modell ins Spiel. Das spart Geld und reduziert Halluzinationen.

```
Du gibst eine Aufgabe
  → Claude sammelt Kontext mit echten Tools (grep, git, ls)
  → Claude plant mit KI
  → Claude führt aus
  → Claude verifiziert (Tests, Compiler, Build)
  → Wenn nicht erfolgreich: Schleife
```

Neben dem Terminal gibt es auch eine Desktop-App, VS Code und JetBrains Extensions sowie `claude.ai/code` im Browser. Die meiste Power steckt aber nach wie vor in der CLI.

2. Installation & erste Schritte

```bash
npm install -g @anthropic-ai/claude-code

# Version prüfen
claude --version

# Updates
claude update
```

Dann einfach ins Projektverzeichnis und loslegen:

```bash
cd mein-projekt
claude
```

Erster Schritt in jedem neuen Projekt: `/init` ausführen. Claude liest die Codebase, erkennt Stack und Struktur und schreibt eine CLAUDE.md. Die ist nie perfekt, aber ein guter Ausgangspunkt.

Wenn etwas nicht funktioniert: `/doctor` — das diagnostiziert kaputte Berechtigungen, MCP-Probleme und merkwürdiges Terminal-Verhalten.

---

3. Die drei Modelle: wann welches

```bash
/model haiku    # schnell, günstig
/model sonnet   # ausgewogen – für den Alltag
/model opus     # maximale Reasoning-Tiefe
```

Kurz: **Sonnet für fast alles, Opus für Planung und schwierige Architektur-Fragen, Haiku für einfache Q&A.** Wer Opus als Default eingestellt hat verbrennt Geld ohne nennenswerten Mehrwert.

Modell	Wann sinnvoll
Haiku 4.5	Einfache Fragen, Lernphase, kurze Lookups
Sonnet 4.6	Implementierung, Debugging, Standard-Code
Opus 4.7	Architektur-Entscheidungen, komplizierte Bugs, Planung

Unabhängig vom Modell gibt es noch die Effort-Einstellung — die steuert wie tief Claude denkt:

```bash
/effort low     # für einfache Edits und Formatierungen
/effort medium  # Normalfall
/effort high    # Architektur, größere Refactorings
/effort max     # nur wenn es wirklich schwierig ist – verbraucht viele Tokens
```

`max` gilt nur für die aktuelle Session. Alle anderen Levels bleiben über Sessions hinweg erhalten.

Konfiguration

4. CLAUDE.md — das Projektgedächtnis

`CLAUDE.md` liegt im Root des Projekts und wird bei jeder Session automatisch in den Kontext geladen. Claude liest sie bevor es irgendetwas anderes macht.

```
mein-projekt/
└── CLAUDE.md   ← hier
```

Für verschiedene Bereiche können eigene CLAUDE.md-Dateien existieren — Claude liest immer die nächstgelegene:

```
/CLAUDE.md                  ← globale Projektregeln
/app/CLAUDE.md              ← Frontend-spezifisch
/app/dashboard/CLAUDE.md    ← für datenlastige Seiten
```

Was hineingehört (und was nicht)

Die Datei sollte **unter 200 Zeilen** bleiben. Klingt wenig, reicht aber. Längere Dateien fressen Tokens und Claude verliert den Fokus auf die wichtigen Regeln.

Die ersten 30 Zeilen bekommen mehr Aufmerksamkeit als der Rest — kritische Regeln also nach oben.

Details auslagern statt alles reinquetschen. Ein Satz `Für Design-System-Details: docs/design-system.md` ist besser als 50 Zeilen Design-System-Regeln direkt in der CLAUDE.md.

Was sinnvoll ist:

MARKDOWN

```
# KK-App – Krankenkassen iOS-App

## Übersicht
Native iOS-App für eine private Krankenversicherungen.
Versicherte verwalten Anträge, Bescheinigungen, Postfach, Bonus-
programm und Kontakt zur Kasse – alles in einer App.
Zielgruppe: alle Versicherungsaltersgruppen, unterschiedliche
Tech-Affinität. Barrierefreiheit ist keine Option, sondern Pflicht.

## Stack
- Swift 5.9 + SwiftUI (iOS 17+, kein UIKit-Fallback)
- Xcode 16
- OAuth2/OIDC via IAM (KeycloakClient in Core/Auth/)
- REST API (Spec: api/openapi.yml)
- SwiftData für lokales Caching (offline-fähig)
```

```
- Keychain für Tokens und sensible Gesundheitsdaten

Nicht verwenden: Combine (async/await stattdessen),
UIKit direkt, Third-Party-Auth-Libraries.

## Ordnerstruktur
- `Features/` → Module: Login, Dashboard, Postfach, Antrag,
                Bonus, Bescheinigung, Gesundheit, Kontakt,
                MeineDaten, Einstellungen
- `Core/` → Netzwerk, Auth/IAM, Persistence, Keychain
- `Shared/` → Design-System, Extensions, Komponenten
- `Resources/` → Lokalisierung (de, en), Assets
API Keys → nie hardcoden → Config.xcconfig + Keychain

## Regeln die nie gebrochen werden
- Kein force-unwrap (!) außer bei @IBOutlet
- Gesundheitsdaten (SGB V §67ff) niemals in UserDefaults –
  nur Keychain oder verschlüsselte SwiftData-Datenbank
- Jedes interaktive UI-Element hat .accessibilityLabel()
  und .accessibilityHint()
- Keine neuen Dependencies ohne Rückfrage
- Nur die Dateien anfassen die zur Aufgabe gehören

## Was nie passieren soll
- Token, Versichertennummer oder Diagnosen in Logs (print/NSLog)
- Netzwerk-Requests ohne Certificate Pinning in Production
- Lorem Ipsum oder Dummy-Versichertendaten
- Direkt auf main pushen ohne PR

## Commands
- Build: `xcodebuild -scheme KKApp -destination
          'platform=iOS Simulator,name=iPhone 16'`
- Test:  `xcodebuild test -scheme KKAppTests`
- Lint:  `swiftlint lint --strict`

## Wichtiges
- IAM-Token läuft nach 15 min ab – RefreshToken-Logic in
  Core/Auth/TokenManager.swift, nicht neu implementieren
- Antrag-Upload: max. 10 MB, erlaubt: PDF, JPG, PNG
- Postfach muss offline-fähig sein (lokaler SwiftData-Cache)
- Bescheinigungen kommen als PDF via Signed URL, nicht Base64
```

Details auslagern — drei Varianten

MARKDOWN

```
# Option 1: Lazy Reference – Claude liest bei Bedarf
Für Design-System-Regeln: docs/design-system.md

# Option 2: @-Import – wird immer mitgeladen
@docs/kritische-regeln.md
```

```
# Option 3: .claude/rules/ – automatisch geladen, mit Kontext-Filter möglich
```

Variante	Wann laden	Token-Kosten
Freitext-Referenz	Claude entscheidet	Niedrig
`@datei.md`	Immer, sofort	Direkt
`.claude/rules/`	Automatisch, filterbar	Mittel

Was nach `/compact` erhalten bleiben soll

Das ist ein unterschätzter Abschnitt. Ohne ihn macht die Compaction eine generische Zusammenfassung:

MARKDOWN

```
## Compact Instructions
Erhalten:
- Aktuelle Architektur-Entscheidungen mit Begründung
- Fehlschlagende Tests und ihre Fehlermeldungen
- Offene Aufgaben dieser Session

Verwerfen:
- Explorative Versuche die nicht funktioniert haben
- Bereits gelöste Debugging-Sessions
```

Session sinnvoll abschließen

```
Aktualisiere CLAUDE.md mit den wichtigen Erkenntnissen aus dieser Session.
Nur was wirklich neu ist – nicht was schon drinsteht.
```

5. MEMORY.md — Auto-Memory

Seit v2.1.76 führt Claude ein eigenes Notizbuch über das Projekt. Kein Setup nötig — läuft automatisch.

Der Unterschied zu CLAUDE.md:

	CLAUDE.md	MEMORY.md
Wer schreibt?	Du	Claude
Inhalt	Regeln, Anweisungen	Claudes Notizen über das Projekt
Wann geladen?	Immer vollständig	Erste 200 Zeilen beim Start

Wo liegt es:

```
~/.claude/projects/<git-root>/memory/
├── MEMORY.md          ← Haupt-Index, erste 200 Zeilen = auto geladen
├── debugging.md       ← von Claude angelegte Themen-Dateien
└── api-conventions.md
```

Funktioniert nur in Git-Repos. Ohne `git init` greift Claude auf das aktuelle Verzeichnis zurück.

Verwalten:

BASH

```
/memory
```

Zeigt drei Optionen: globale User-Einstellungen, Projekt-CLAUDE.md, und den Auto-Memory-Ordner.

Ob es funktioniert testen:

BASH

```
# Session starten, etwas bauen, schließen
# Neue Session:
Was weißt du über dieses Projekt?
```

Deaktivieren:

BASH

```
# Einzelprojekt (.claude/settings.json)
{ "autoMemoryEnabled": false }

# Global (~/.claude/settings.json)
{ "autoMemoryEnabled": false }

# In CI/CD
export CLAUDE_CODE_DISABLE_AUTO_MEMORY=1
```

Die vier Bausteine

Commands, Skills, Agents und Hooks — das sind die vier Mechanismen mit denen du Claude Code wirklich kontrollierst. Für jeden gilt: **Schreiben → Speichern → Verwenden.**

---

6. Commands — eigene Slash-Befehle

Was ist ein Command?

Eine Markdown-Datei = ein `/slash-command`. Schnelle, einmalige Aktionen. Der Dateiname wird der Befehlsname.

```
commands/review-pr.md  →  /review-pr
commands/fix-bug.md    →  /fix-bug
```

Schreiben

Normales Markdown, Claude-Anweisungen drin. Nichts Magisches.

`.claude/commands/review-pr.md`

MARKDOWN

```
Reviewe die aktuellen Code-Änderungen für die KK-App.

Fokus:
- Swift-Korrektheit: Optionals, Memory-Management (ARC), Concurrency
- Sicherheit: Werden Gesundheitsdaten korrekt behandelt?
  (nur Keychain, nie UserDefaults/Logs)
- Barrierefreiheit: accessibilityLabel, accessibilityHint vorhanden?
- IAM/Auth: Token-Handling korrekt? Kein Token in Logs?
- Offline-Fähigkeit: Ist SwiftData-Caching berücksichtigt?

Ausgabe:
1. Kritische Probleme (blockieren Merge – z.B. Datenschutz, Datenverlust)
2. Mittlere Probleme (sollten vor Merge behoben werden)
3. Kleine Anmerkungen

Fang mit einem Satz Zusammenfassung an.
```

`.claude/commands/check-barrierefreiheit.md`

MARKDOWN

```
Prüfe alle SwiftUI-Views in $ARGUMENTS auf iOS-Barrierefreiheit.
```

```
Checkliste:
- Jeder Button/Tap-Bereich hat .accessibilityLabel()
- Beschreibende .accessibilityHint() wo die Aktion nicht
  offensichtlich ist
- Bilder mit Informationsgehalt haben accessibilityLabel,
  dekorative .accessibilityHidden(true)
- Mindest-Tap-Größe 44×44pt
- VoiceOver-Reihenfolge sinnvoll (.accessibilitySortPriority)
- Keine rein farbbasierten Status-Indikatoren

Ausgabe: Datei + Zeile + konkreter Fix.
Wenn Barrierefreiheit korrekt: kurze Bestätigung.
```

`.claude/commands/neuer-screen.md`

MARKDOWN

```
Erstelle ein Gerüst für einen neuen Feature-Screen: $ARGUMENTS

1. SwiftUI View in Features/$ARGUMENTS/$ARGUMENTSView.swift
2. ViewModel in Features/$ARGUMENTS/$ARGUMENTSViewModel.swift
   (ObservableObject, @Published Properties)
3. Navigation-Eintrag in AppRouter falls nötig
4. Lokalisierungs-Keys in Resources/de.lproj/Localizable.strings

Regeln:
- Kein UIKit
- Keine hardcodierten deutschen Texte im View – immer NSLocalizedString
- accessibilityLabel auf alle interaktiven Elemente
- Lade-Zustände (loading, error, empty) von Anfang an berücksichtigen

Zeig mir die Struktur zuerst, dann warte auf mein OK bevor du schreibst.
```

Mit Argument (`$ARGUMENTS`):

`.claude/commands/erklaer.md`

MARKDOWN

```
Erkläre $ARGUMENTS in einfachen Worten.
Gib ein konkretes Beispiel aus der KK-App.
Wenn es um Datenschutz oder GKV-Recht geht: zitiere den
relevanten Paragraphen (SGB V, DSGVO, TMG).
```

Aufrufen:

BASH

```
/neuer-screen Bescheinigung
/check-barrierefreiheit Features/Antrag/
/erklaer den IAM-TokenManager
```

Speichern

```
.claude/commands/        ← Projektspezifisch, via Git geteilt
~/.claude/commands/      ← Persönlich, alle Projekte
```

Verwenden

BASH

```
/review-pr
/check-barrierefreiheit Features/Antrag/
/neuer-screen Bescheinigung
/erklaer den IAM-TokenManager
```

Claude kann Commands auch selbst aufrufen wenn es passt.

Commands vs. Skills

Commands	Skills
Schnelle, einmalige Aktionen	Komplexe, mehrstufige Workflows
Eine .md-Datei	Ordner mit mehreren Dateien
Lädt komplett in Haupt-Kontext	Lazy-loaded
"Tue X, fertig"	"Bleib in diesem Modus"

7. Skills — komplexe Workflows verpacken

Was ist ein Skill?

Ein Skill ist ein paketiertes Verfahren — Claude lädt es und bleibt dann in diesem Modus. Anders als ein Command der einmalig feuert, hält ein Skill den Kontext für die gesamte Aufgabe aufrecht.

Analogie: Command = "Schau kurz drüber" Skill = "Du bist ab jetzt unser Security-Reviewer"

Schreiben

Pflichtdatei: `SKILL.md` mit YAML-Frontmatter im Ordner.

`.claude/skills/ios-ux-review/SKILL.md`

MARKDOWN

```
---
name: ios-ux-review
description: Prüft SwiftUI-Views auf iOS-HIG-Konformität und Barrierefreiheit
für die KK-App. Trigger: View-Review, UX-Audit, Onboarding-Flow, Antrag-Flow,
Barrierefreiheit. NICHT für: Swift-Logik, API-Design, DSGVO.
tools: Read, Edit
---

# Rolle
Senior iOS Designer, spezialisiert auf Gesundheits-Apps und
barrierefreie iOS-Anwendungen im GKV-Kontext.

# Verwende diesen Skill wenn
- SwiftUI-Views oder Flows reviewed werden sollen
- Barrierefreiheit für ältere oder eingeschränkte Nutzer geprüft wird
- Antrag- oder Formular-Flows bewertet werden
- Onboarding/Login-Flow analysiert wird

# Verwende ihn nicht für
- Swift-Logik oder Architektur-Fragen
- DSGVO-Compliance → dafür datenschutz-audit nutzen
- Backend-API-Design

# Ablauf (immer in dieser Reihenfolge)
1. Zielgruppe verstehen (wer nutzt diesen Screen?)
2. Visuelle Hierarchie und Information-Scent prüfen
3. Tap-Targets auf Mindestgröße 44×44pt prüfen
4. VoiceOver-Reihenfolge und Labels prüfen
5. Lade-, Fehler- und Leer-Zustände vorhanden?
6. iOS HIG: Navigation, Gesten, System-Komponenten korrekt?
7. Befunde nach Schweregrad sortieren

# Besondere Anforderungen KK-App
- Ältere Nutzer: Dynamic Type, klare Sprache, kein Jargon
- Antragsformulare: Fortschritts-Indikator, kein Datenverlust
  beim App-Wechsel, Bestätigungsdialog vor Absenden
```

```
- Bescheinigungen: klar erkennbar ob PDF verfügbar oder nicht

# Schweregrad
- Critical → Nutzer kann Aufgabe nicht abschließen
- Major → Usability deutlich beeinträchtigt
- Minor → kosmetisch oder verbesserungswürdig

# Ausgabe
## UX-Befunde

### Kritisch
- **Problem:** [Was genau, Datei:Zeile]
- **Impact:** [Wen betrifft es]
- **Fix:** [Konkreter SwiftUI-Code oder Anweisung]

### Mittel
### Klein

## Score: [X/10]

# Niemals
- Komponenten vorschlagen die nicht in Shared/ existieren
- Generisches Feedback ohne Belege

# Bei fehlenden Infos
Fragen:
- "Welche Nutzergruppe?"
- "Was ist der primäre Job-to-be-done auf diesem Screen?"
```

Skill mit mehreren Dateien:

`.claude/skills/datenschutz-audit/SKILL.md`

MARKDOWN

```
---
name: datenschutz-audit
description: DSGVO- und SGB-V-Compliance-Audit für Gesundheitsdaten in der
KK-App. Trigger: vor Merges mit Datenspeicherung, nach Auth-Änderungen, bei
neuen API-Endpoints die PII verarbeiten. NICHT für: allgemeines Code-Review,
UI-Fragen.
tools: Read, Grep, Bash
---

# Rolle
Datenschutzbeauftragter mit Swift/iOS-Kenntnissen.
Fokus: DSGVO Art. 9 (besondere Kategorien), SGB V §67–§75,
BSI TR-03161 (Gesundheits-Apps), OWASP Mobile Top 10.

# Ablauf
1. Datenpunkte identifizieren: was wird gespeichert, übertragen, geloggt?
2. Speicherorte prüfen → @speicherorte-checklist.md
3. Token- und Session-Handling prüfen
4. Netzwerk-Sicherheit prüfen (Certificate Pinning, TLS)
```

```
5. Logging auf Datenlecks prüfen
6. Einwilligungen und Betroffenenrechte prüfen

# Ausgabe
- Betroffene Datei und Zeile zitieren
- Rechtsgrundlage nennen (DSGVO Art. X, SGB V §Y)
- Risiko: Critical / High / Medium / Low
- Konkreten Fix nennen
```

`.claude/skills/datenschutz-audit/speicherorte-checklist.md`

MARKDOWN

```
# Datenspeicher-Checkliste iOS KK-App

## Was niemals in UserDefaults
- [ ] Versichertennummer, KVNR
- [ ] Diagnosen, Medikamente, Behandlungsinfos
- [ ] IAM-Tokens (Access Token, Refresh Token)

## Keychain (Pflicht für)
- [ ] Access Token + Refresh Token
- [ ] Benutzername / Versichertennummer
- [ ] Certificate-Pinning-Hashes

## SwiftData/Core Data (verschlüsselt)
- [ ] Postfach-Nachrichten (offline-Cache)
- [ ] Antrag-Entwürfe
- [ ] Bescheinigungen (nur Referenz, nicht Inhalt)

## Netzwerk
- [ ] Certificate Pinning für alle API-Endpoints
- [ ] TLS 1.3 erzwungen (kein Fallback auf ältere Versionen)
- [ ] Keine PII in URL-Parametern (→ Request Body)
- [ ] Keine PII in Crash-Reports (Sentry/Firebase)

## Logging
- [ ] Kein print() oder NSLog() mit Token, IDs, Diagnosen
- [ ] OSLog mit privacy: .private für alle PII
- [ ] Analytics-Events enthalten keine Klardaten
```

Speichern

```
.claude/
└── skills/
    ├── ios-ux-review/
    │   ├── SKILL.md
    │   └── examples/
    └── datenschutz-audit/
        ├── SKILL.md
        └── speicherorte-checklist.md
```

Skills sind Ordner. Commands sind einzelne Dateien.

Verwenden

```bash
# Explizit
/ios-ux-review

# In der Prompt
Nutze den ios-ux-review Skill für den Antrag-Screen

# Automatisch – wenn die description gut geschrieben ist
Kannst du mal den Onboarding-Flow reviewen?
# → Claude wählt ios-ux-review
```

Was eine gute `description` ausmacht: Trigger-Wörter + explizite Ausschlüsse. Claude entscheidet anhand der Description ob es den Skill automatisch lädt. Eine vage Description bedeutet: Claude nutzt ihn nie automatisch.

```markdown
# Schlecht
description: Hilft mit iOS-UX

# Gut
description: Prüft SwiftUI-Views auf HIG und Barrierefreiheit.
Trigger: UX-Review, Antrag-Flow, Onboarding. NICHT für Swift-Logik.
```

Skill-Checkliste:

- Eine Aufgabe pro Skill
- Klare Trigger und Ausschlüsse in der description
- Deterministischer Workflow (nummerierte Schritte)
- Exaktes Output-Format
- Was Claude nie tun soll
- Bei fehlenden Infos: fragen statt raten
- 100–250 Zeilen für SKILL.md

> **Sicherheitshinweis:** Skills von Drittanbietern können Prompt Injections enthalten. Nur Skills aus vertrauenswürdigen Quellen installieren.

8. Agents — spezialisierte Sub-Instanzen

Was ist ein Agent?

Ein Sub-Agent ist eine **isolierte Claude-Instanz** mit eigenem Kontext-Fenster. Der entscheidende Unterschied zu Skills und Commands: Sub-Agents laufen **unabhängig**, können **parallel** arbeiten und bringen nur das Ergebnis zurück. Das hält den Haupt-Kontext sauber.

	Command	Skill	Agent
Kontext	Haupt-Kontext	Haupt-Kontext	Eigener, isoliert
Parallel möglich?	Nein	Nein	Ja
Kostet mehr?	Nein	Mittel	Ja (eigene Instanz)
Eigenes Modell?	Nein	Nein	Ja

Schreiben

`.claude/agents/swift-reviewer.md`

MARKDOWN

```
---
name: swift-reviewer
description: Swift/SwiftUI Code-Review vor Merges in der KK-App. Trigger:
wenn Review ansteht, nach größeren Feature-Änderungen, bei neuen ViewModels
oder Core-Änderungen.
model: sonnet
tools: Read, Grep
---

Senior iOS Developer mit Fokus auf Swift-Korrektheit und
Wartbarkeit.

Prüft:
- Swift-Korrektheit: Optionals, ARC (retain cycles), Concurrency
- SwiftUI: View-Struktur, State-Management (@State vs @StateObject)
- Kein force-unwrap (!) außer @IBOutlet
- ViewModel-Trennung: keine Business-Logik in Views
- Tests vorhanden für kritische Logik (XCTest)
- Kein Over-Engineering

Vorgehen:
1. Geänderte Dateien lesen
2. Kontext in Features/ und Core/ verstehen
3. Strukturiert ausgeben

## Swift Code Review

**Zusammenfassung:** [1 Satz]

**Muss vor Merge behoben werden:**
```

```
- [Datei:Zeile] Problem – Empfehlung

**Sollte behoben werden:**
- ...

**Kleine Anmerkungen:**
- ...

**Fazit:** APPROVE / REQUEST_CHANGES
```

`.claude/agents/datenschutz-prufer.md`

MARKDOWN

```
---
name: datenschutz-prufer
description: DSGVO/SGB-V-Audit für KK-App-Releases. Liest nur, schreibt nie.
Trigger: vor jedem Release, nach Änderungen an Core/Auth, nach neuen
Datenspeicher- oder API-Änderungen.
model: opus
tools: Read, Grep
---

Datenschutzbeauftragter, spezialisiert auf iOS-Gesundheits-Apps.
Liest Code. Schreibt NIEMALS etwas.

Rechtsgrundlagen: DSGVO Art. 5/6/9, SGB V §67–§75,
BSI TR-03161, OWASP Mobile Top 10.

Prüft:
- Wo werden Gesundheitsdaten gespeichert? (Keychain? UserDefaults?)
- Tokens im Klartext in Logs?
- Certificate Pinning aktiv?
- PII in Analytics/Crash-Reports?
- Einwilligungen korrekt implementiert?

Für jeden Fund:
- Datei + Zeile zitieren
- Rechtsgrundlage (DSGVO Art. X / SGB V §Y)
- Risiko: Critical / High / Medium / Low
- Konkreten Fix nennen
```

`.claude/agents/api-doc-writer.md`

MARKDOWN

```
---
name: api-doc-writer
description: Schreibt OpenAPI-Dokumentation für KK-App-Backend-Endpunkte.
Trigger: nach API-Änderungen, wenn Endpunkte fehlen oder veraltet sind in
api/openapi.yml.
model: sonnet
tools: Read, Write, Edit
```

```
---

Technical Writer für REST-APIs im GKV-Kontext.

Stil:
- Klare, kurze Beschreibungen (kein Marketing-Sprech)
- Echte Request/Response-Beispiele mit realistischen Testdaten
  (keine echten Versichertendaten!)
- Fehler-Responses dokumentieren (401, 403, 422, 500)
- OpenAPI 3.0 Format

Workflow:
1. Vorhandene Einträge in api/openapi.yml prüfen
2. Relevanten Controller-Code lesen
3. Schema + Beschreibungen ergänzen
4. Sicherstellen: Keine PII in Beispiel-Responses
```

Orchestrator-Worker-Muster

MARKDOWN

```
---
name: release-orchestrator
description: Koordiniert Pre-Release-Checks für KK-App. Trigger: "Release
vorbereiten", "Pre-Release-Check".
model: opus
tools: Task, Read
---

Koordiniert den Release-Prozess:
1. Aufgaben aufteilen
2. Parallel ausführen:
   - swift-reviewer → Code-Qualität
   - datenschutz-prufer → DSGVO-Compliance
   - api-doc-writer → Docs aktuell?
3. Ergebnisse zusammenführen
4. Release-Checkliste ausgeben

Verfügbare Agents: swift-reviewer, datenschutz-prufer,
                   api-doc-writer
```

Speichern

```
.claude/
└── agents/
    ├── swift-reviewer.md
    ├── datenschutz-prufer.md
    ├── api-doc-writer.md
    └── release-orchestrator.md
```

Ein Agent = eine .md-Datei (nicht wie Skills ein Ordner).

Verwenden

```bash
# Explizit
@swift-reviewer Bitte reviewe Features/Antrag/

# Agent direkt starten
claude --agent=datenschutz-prufer

# Automatisch – wenn description passt
Kannst du den Code vor dem Release prüfen?
# → Claude wählt release-orchestrator
```

Agent-Teams (experimentell):

```bash
# In Claude Settings aktivieren, dann:
Erstelle ein Agent-Team für den KK-App-Release:
- Swift-Reviewer (Code)
- Datenschutz-Prüfer (DSGVO)
- UX-Reviewer (Barrierefreiheit)

# Shift+Down → zum nächsten Teammitglied
```

Agent-Teams sind mächtig aber teuer — jedes Mitglied ist eine eigene Instanz.

9. Hooks — Verhalten erzwingen

Was sind Hooks?

Hooks sind Shell-Befehle die zu festen Zeitpunkten im Claude-Lifecycle automatisch ausgeführt werden. CLAUDE.md kann Claude *erklären* keine .env-Dateien zu ändern. Ein Hook *erzwingt* es auf Code-Ebene — unabhängig davon ob Claude die Regel verstanden hat oder nicht.

Schreiben

Konfiguration in `settings.json`:

JSON

```
{
  "hooks": {
    "[EVENT]": [
      {
        "matcher": "[Tool-Muster]",
        "hooks": [
          {
            "type": "command",
            "command": "[Shell-Befehl]"
          }
        ]
      }
    ]
  }
}
```

Speichern

```
.claude/settings.json           ← Projekt (in Git)
.claude/settings.local.json     ← Lokal (nicht in Git)
~/.claude/settings.json         ← Global (alle Projekte)
```

Events

Event	Wann	Blockierend?
`SessionStart`	Session startet	Nein
`UserPromptSubmit`	Du sendest Prompt	Ja
`PreToolUse`	Vor Tool-Aufruf	Ja
`PostToolUse`	Nach Tool-Aufruf	Ja
`PermissionRequest`	Claude fragt um Erlaubnis	Ja
`Stop`	Claude ist fertig	Nein
`Notification`	Claude-Benachrichtigung	Nein

Praktische Beispiele — KK-App

SwiftLint nach jedem Swift-Edit:

JSON

```
{
  "hooks": {
    "PostToolUse": [{
      "matcher": "Edit|Write",
      "hooks": [{
        "type": "command",
        "command": "if [[ \"$CLAUDE_FILE_PATH\" == *.swift ]]; then swiftlint
lint --path \"$CLAUDE_FILE_PATH\" --quiet; fi"
      }]
    }]
  }
}
```

Gesundheitsdaten-Schreibschutz — Keychain-Umgehung verhindern:

`.claude/hooks/protect-health-data.sh`:

BASH

```
#!/usr/bin/env bash
INPUT=$(cat)
FILE=$(echo "$INPUT" | jq -r '.tool_input.file_path // ""')

# UserDefaults-Nutzung für sensible Daten blockieren
if echo "$FILE" | grep -qE "\.swift$"; then
  CONTENT=$(echo "$INPUT" | jq -r '.tool_input.content // ""')
  if echo "$CONTENT" | grep -qE
"UserDefaults.*KVNR|UserDefaults.*token|UserDefaults.*diagnos"; then
    echo '{"decision": "deny", "reason": "Gesundheitsdaten/Tokens dürfen
nicht in UserDefaults — nur Keychain verwenden. Siehe
Core/Auth/KeychainManager.swift"}'
    exit 0
  fi
fi
```

`settings.json`:

JSON

```
{
  "hooks": {
    "PreToolUse": [{
      "matcher": "Edit|Write",
      "hooks": [{"type": "command", "command": "bash .claude/hooks/protect-
health-data.sh"}]
```

```
    }]
  }
}
```

Config.xcconfig schreibschützen (API-Endpoints, Zertifikate):

`.claude/hooks/protect-config.sh`:

BASH

```
#!/usr/bin/env bash
INPUT=$(cat)
FILE=$(echo "$INPUT" | jq -r '.tool_input.file_path // ""')

if echo "$FILE" | grep -qE "Config\.xcconfig$|Certificates/|\.p12$|\.cer$";
then
  echo '{"decision": "deny", "reason": "Config/Zertifikat-Dateien sind
schreibgeschützt. Bitte manuell und mit Review ändern."}'
  exit 0
fi
```

Gefährliche Bash-Befehle blockieren:

`.claude/hooks/block-dangerous.sh`:

BASH

```
#!/usr/bin/env bash
INPUT=$(cat)
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command // ""')

if echo "$COMMAND" | grep -qE "rm -rf /|xcodebuild.*-destination.*device";
then
  echo '{"decision": "deny", "reason": "Gefährlicher Befehl blockiert. Kein
Deployment auf echte Geräte ohne manuellen Schritt."}'
  exit 0
fi
```

Sound wenn Claude fertig ist (macOS):

JSON

```
{
  "hooks": {
    "Stop": [{"hooks": [{"type": "command", "command": "afplay
/System/Library/Sounds/Glass.aiff"}]}]
  }
}
```

Unit-Tests automatisch nach dem Stop:

JSON

```
{
  "hooks": {
    "Stop": [{"hooks": [{"type": "command", "command": "xcodebuild test -
scheme KKAppTests -destination 'platform=iOS Simulator,name=iPhone 16' 2>&1 |
tail -10"}]}]
  }
}
```

Alle Swift-Änderungen loggen (für DSGVO-Audit-Trail):

JSON

```
{
  "hooks": {
    "PostToolUse": [{
      "matcher": "Edit|Write",
      "hooks": [{"type": "command", "command": "echo \"$(date):
$CLAUDE_FILE_PATH\" >> ~/.claude/kk-app-edit-log.txt"}]
    }]
  }
}
```

Was ein Hook zurückgeben kann

Rückgabe	Effekt
Nichts (exit 0)	Claude macht weiter
`{"decision": "deny", "reason": "..."}`	Geblockt, Claude akzeptiert es
`{"decision": "ask"}`	User wird gefragt
`exit 2`	Systemfehler — Claude sucht Umgehung!
`exit 1`	Wird ignoriert

`exit 2` bedeutet für Claude "das System ist kaputt" — es sucht dann nach einem anderen Weg. `{"decision": "deny"}` bedeutet "das ist Policy" — Claude akzeptiert die Ablehnung. Für Sicherheits-Hooks fast immer letzteres verwenden.

Merge-Logik bei mehreren Hooks

`deny > ask > allow` — Ein einziges deny gewinnt.

```
User-Level Hook     → allow
Projekt-Level Hook  → ask
Plugin-Hook         → deny
─────────────────────────────
Ergebnis            → deny
```

Verfügbare Variablen in Hooks

BASH

```
$CLAUDE_FILE_PATH      # Pfad zur bearbeiteten Datei
$TOOL_INPUT            # Vollständiger Tool-Call als JSON
$CLAUDE_SESSION_ID     # Aktuelle Session-ID
$CLAUDE_PLUGIN_ROOT    # Root des aktiven Plugins
```

Werkzeuge

10. Eingebaute Slash-Commands

Projekt & Kontext

Command	Funktion
`/init`	CLAUDE.md für aktuelles Projekt generieren
`/context`	Kontext-Nutzung anzeigen
`/compact [Anweisung]`	Kontext komprimieren
`/clear`	Kontext leeren
`/memory`	Memory-Verwaltung

Session

Command	Funktion
`/resume`	Vorherige Session auswählen
`/plan [Aufgabe]`	Plan-Modus starten
`/focus`	Zwischenschritte ausblenden
`/copy`	Output in Clipboard
`/cost`	Kosten der Session anzeigen

Modell

Command	Funktion
`/model sonnet`	Modell wechseln
`/effort low|medium|high|max`	Reasoning-Tiefe

Tools & Config

Command	Funktion
`/permissions`	Berechtigungen verwalten
`/agents`	Sub-Agent-Manager
`/hooks`	Hook-Verwaltung
`/mcp`	MCP-Server verwalten
`/doctor`	System-Diagnose
`/statusline`	Status-Zeile anpassen

Spezial

Command	Funktion
`/insights`	Muster aus Sessions analysieren
`/batch`	Mehrere Aufgaben gebündelt
`/voice`	Sprachsteuerung
`/security-review`	Eingebautes Security-Review
`/simplify`	Code vereinfachen
`/ultraplan`	Maximale Planungstiefe
`/install-github-app`	GitHub-App installieren
`/fewer-permission-prompts`	Häufige Befehle zur Allowlist hinzufügen

`/compact` richtig nutzen:

BASH

```
# Schlechte Variante (generische Zusammenfassung)
/compact

# Gute Variante
/compact Behalte: Auth-Refactor-Entscheidungen, fehlschlagende Tests.
         Verwirf: CSS-Experimente, alte Debug-Sessions.
```

11. CLI-Flags & versteckte Tricks

Beim Start

BASH

```
claude                          # Normal
claude --resume                 # Session auswählen
claude --continue               # Letzte Session direkt
claude --enable-auto-mode       # Auto Mode an
claude --model sonnet           # Modell beim Start setzen
```

Headless — Claude in Scripts

BASH

```
# Einmalig ausführen
claude -p "Fasse CHANGELOG.md zusammen"

# Mit Pipes
git diff main..HEAD | claude -p "Schreibe eine PR-Beschreibung"
cat error.log | claude -p "Klassifiziere jede Zeile: warning/error/info"

# In Datei schreiben
claude -p "Analysiere die Architektur" > architecture-review.md

# Strukturierter JSON-Output
claude -p "Liste alle TODOs" --output-format json

# Streaming JSON
claude -p "Analysiere..." --output-format stream-json
```

System-Prompt für einen Aufruf setzen

BASH

```
# Ändert nicht CLAUDE.md, gilt nur für diesen Aufruf
claude --append-system-prompt "Antworten unter 100 Wörtern. Kein Code außer
wenn gefragt."

# Kombiniert mit Headless
claude -p "Review this code" \
  --append-system-prompt "Output: severity, Zusammenfassung, Fix. Keine
Einleitung."
```

Tool-Zugriff einschränken

```bash
claude --allowedTools "Read,Grep,Bash"     # Nur diese Tools erlaubt
claude --disallowedTools "Bash"             # Bash deaktivieren
claude --disallowedTools "Write,Edit,Bash" # Read-only Session
```

Permission Modes

```bash
claude --permission-mode default           # Alles bestätigen (default)
claude --permission-mode acceptEdits       # Datei-Edits auto-erlauben
claude --permission-mode bypassPermissions # ALLES überspringen — nur in
isolierten Umgebungen!
claude --permission-mode plan              # Nur lesen
```

`bypassPermissions` gehört in Docker oder CI, nicht auf dem eigenen Rechner.

Schnellstart ohne Discovery

```bash
# Überspringt: Hooks, Skills, Plugins, MCP-Discovery, CLAUDE.md
# Für einfache Headless-Scripts deutlich schneller
claude --bare -p "Meine schnelle Aufgabe"
```

Das `#`-Präfix — sofort etwas merken

```bash
# Am Anfang eines Prompts: Claude speichert es als Memory-Eintrag
# Immer pnpm verwenden, nie npm
# Nie direkt auf main pushen ohne PR
```

Claude fragt welche Memory-Datei (User-global, Projekt, lokal).

Das `!`-Präfix — direkte Shell-Befehle ohne KI

```bash
! git status
! pnpm test
! docker ps
```

Output landet automatisch im Kontext.

@-Syntax — Datei in Prompt einbinden

BASH

```
@src/auth/login.ts Erkläre mir diese Funktion
@package.json Welche Dependencies könnten veraltet sein?
@CHANGELOG.md Was änderte sich in der letzten Version?

# Mehrere Dateien
@src/auth/login.ts @src/auth/types.ts Refactore die Auth-Logik
```

Shell-Aliases für den Alltag

BASH

```
# ~/.zshrc
alias cc="claude"
alias ccr="claude --resume"
alias ccc="claude --continue"
alias ccp="claude -p"

# Workflows
alias cc-review="git diff | claude -p 'Code-Review dieser Änderungen'"
alias cc-pr="git diff main..HEAD | claude -p 'Schreibe eine PR-Beschreibung'"
alias cc-commit="git diff --staged | claude -p 'Schreibe eine Commit-
Message'"
alias cc-tests="claude -p 'Schreibe fehlende Tests für die geänderten
Dateien'"
```

12. Keyboard Shortcuts

Interaktiver Modus

Shortcut	Aktion
`Enter`	Absenden
`Shift+Enter` oder `\` + `Enter`	Neue Zeile (ohne Absenden)
↑ / ↓	Prompt-History
`Ctrl+R`	History durchsuchen
`Tab`	Slash-Commands autocompleten
`Escape`	Input leeren
`Ctrl+C`	Aktuelle Operation abbrechen

Modi

Shortcut	Aktion
`Shift+Tab` (1×)	Plan Mode
`Shift+Tab` (2×)	Auto Mode
`Shift+Down`	Zum nächsten Agent-Teammitglied

13. MCP-Server

MCP (Model Context Protocol) verbindet Claude mit externen Systemen: Datenbanken, Browser, Slack, GitHub.

```bash
/mcp    # Server verwalten
```

Konfiguration

```json
{
  "mcpServers": {
    "postgresql": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres"],
      "env": { "POSTGRES_URL": "${POSTGRES_URL}" }
    },
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": { "GITHUB_TOKEN": "${GITHUB_TOKEN}" }
    }
  }
}
```

Wichtige Server

Kategorie	Server
Browser	Playwright, Puppeteer
Scraping	Firecrawl
Datenbanken	PostgreSQL, Supabase, SQLite
Dev	GitHub, Sentry, Linear
Produktivität	Slack, Notion, Figma

Faustregel: Ungenutzte MCPs deaktivieren. Jeder Server verbraucht beim Start Kontext-Tokens — selbst wenn er nicht verwendet wird.

Fortgeschrittenes

14. Auto Mode

Auto Mode ersetzt manuelle Bestätigungen durch einen KI-Klassifikator der jede Aktion auf Sicherheit prüft. Sinnvoll für lange autonome Tasks.

Aktivieren

BASH

```
claude --enable-auto-mode
# Im Terminal: Shift+Tab bis "▸▸ auto" erscheint
```

Konfiguration

> Auto Mode nur in **User-Settings** konfigurieren (`~/.claude/settings.json`), nicht im Repo — sonst könnten eingecheckte Settings eigene Regeln injizieren.

JSON

```
{
  "autoMode": {
    "environment": [
      "Organisation: Meine Firma, Software-Entwicklung",
      "Source Control: github.com/meine-org",
      "Cloud: AWS eu-central-1",
      "Vertrauenswürdige Domains: *.intern.meine-firma.de"
    ],
    "soft_deny": [
      "$defaults",
      "Niemals terraform destroy gegen prod-Workspaces"
    ],
    "hard_deny": [
      "$defaults",
      "Niemals rm -rf außerhalb des Working-Directory",
      "Niemals force-push auf main oder release/*"
    ],
    "allow": [
      "$defaults",
      "Deployment auf staging ist OK"
    ]
  }
}
```

`$defaults` immer einfügen:

```json
// Kaputt — überschreibt ALLE eingebauten Schutzregeln
{ "soft_deny": ["Niemals prod-DB anfassen"] }

// Richtig — eigene Regeln ergänzen statt ersetzen
{ "soft_deny": ["$defaults", "Niemals prod-DB anfassen"] }
```

Was standardmäßig gesperrt ist (muss nicht konfiguriert werden):

- Force-Push, Remote-Branches löschen, History umschreiben
- Direkt auf `main`/`master` pushen
- Production-Deploys und Migrationen
- `curl | bash` und Code von externen Quellen
- Credentials aus Umgebungsvariablen auslesen

Konfiguration überprüfen:

```bash
claude auto-mode defaults   # Was ist geblockt?
claude auto-mode config     # Effektive Konfiguration
claude auto-mode critique   # KI prüft deine Regeln
```

15. Think-Keywords

Claude hat verschiedene Denk-Intensitäten die per Keyword gesteuert werden:

Keyword	Intensität	Wann sinnvoll
(kein)	Standard	Alltag
`think`	Erhöht	Mittlere Komplexität
`think hard`	Hoch	Schwierige Probleme
`think harder`	Sehr hoch	Komplexe Architektur
`ultrathink`	Maximum	Härteste Fälle — viele Tokens

BASH

```
think: Analysiere mögliche Race Conditions in diesem Code

think hard: Entwirf die Datenbankarchitektur für dieses Feature

ultrathink: Finde alle Sicherheitslücken in diesem Auth-Flow
```

`ultrathink` sparsam einsetzen — es verbraucht deutlich mehr Tokens als normale Aufrufe.

16. Umgebungsvariablen & Cloud-Provider

Kern

```bash
export ANTHROPIC_API_KEY="sk-ant-..."
export ANTHROPIC_BASE_URL="http://localhost:11434"  # Proxy oder lokales LLM
export ANTHROPIC_MODEL="claude-sonnet-4-6"          # Modell überschreiben
```

Token-Kontrolle

```bash
export CLAUDE_CODE_MAX_OUTPUT_TOKENS=8192
export MAX_THINKING_TOKENS=10000
```

Features

```bash
export CLAUDE_CODE_DISABLE_AUTO_MEMORY=1    # Auto-Memory aus
export DISABLE_AUTOUPDATER=1                # Keine Auto-Updates
export CLAUDE_CODE_DISABLE_TELEMETRY=1      # Keine Telemetrie
```

AWS Bedrock

```bash
export CLAUDE_CODE_USE_BEDROCK=1
export AWS_REGION=us-east-1
# AWS Credentials müssen konfiguriert sein
claude
```

Google Vertex AI

```bash
export CLAUDE_CODE_USE_VERTEX=1
export CLOUD_ML_REGION=us-east5
export ANTHROPIC_VERTEX_PROJECT_ID=mein-gcp-projekt
claude
```

Best Practices

17. Die 4 Karpathy-Regeln

Andrej Karpathy beschrieb im Januar 2026 die häufigsten Fehler von KI-Coding-Agents. Ein GitHub-Repo das diese vier Regeln in eine CLAUDE.md destilliert hat bekam 60.000 Sterne innerhalb einer Woche.

```markdown
1. Nicht annehmen. Verwirrung nicht verstecken. Tradeoffs aufzeigen.
2. Minimaler Code der das Problem löst. Nichts Spekulatives.
3. Nur anfassen was nötig ist. Nur den eigenen Dreck aufräumen.
4. Erfolgskriterien definieren. Loop bis verifiziert.
```

Regel 1 — Nicht annehmen:

```
Ohne: Claude implementiert sofort mit eigenen Annahmen.
      export_users() schreibt JSON, alle User, auf Disk, alle Felder.

Mit: "Bevor ich anfange – welche User? Welches Format? Sensible Felder?"
```

Regel 2 — Minimaler Code:

```python
# Ohne: 40 Zeilen Strategy Pattern für eine Rechenoperation
class DiscountStrategy(ABC):
    @abstractmethod
    def calculate(self, amount: float) -> float: pass
# ...

# Mit: direkt
def calculate_discount(amount: float, percent: float) -> float:
    return amount * (percent / 100)
```

Regel 3 — Chirurgisch:

```
Aufgabe: Crash bei leerer E-Mail fixen.

Ohne Regel: Claude ändert auch Username-Validierung, formatiert Quotes um,
            fügt Type-Hints hinzu → 40-Zeilen-Diff für einen 3-Zeilen-Bug.

Mit Regel: Genau die drei betroffenen Zeilen, nichts anderes.
```

Regel 4 — Erfolgskriterien:

```
Ohne: "Ich werde das Auth-System fixen durch Review, Analyse,
Verbesserung..."

Mit:  1. Test: altes Token nach Passwort-Änderung → ungültig
         Verify: Test schlägt fehl ✓ (Bug reproduziert)
      2. Fix: Sessions invalidieren
         Verify: Test besteht ✓
      3. Regression: alle Auth-Tests grün ✓
```

Installieren:

BASH

```
# Plugin
/plugin marketplace add forrestchang/andrej-karpathy-skills
/plugin install andrej-karpathy-skills@karpathy-skills

# Direkt
curl -o CLAUDE.md https://raw.githubusercontent.com/forrestchang/andrej-
karpathy-skills/main/CLAUDE.md
```

Das Konfigurations-Paradox: Mehr Regeln klingen nach mehr Kontrolle. Ab einer bestimmten Menge produzieren sie aber verwirrte Agents statt disziplinierte. Claude Code limitiert Regel-Dateien auf 6.000 Zeichen, alle zusammen auf 12.000. Die vier Karpathy-Regeln funktionieren weil sie Verhalten formen statt einzelne Dinge zu verbieten.

---

18. Kontext & Kosten im Griff behalten

Wo die Tokens hinfließen

```bash
/context    # Zeigt Kontext-Nutzung
/cost       # Kosten der aktuellen Session
```

Beim Start einer Session fließen bereits Tokens:

```
Basis-System-Prompt:        ~2.000 Token
CLAUDE.md:                  ~500–2.000
@-Imports:                  variabel
MEMORY.md:                  ~1.000–3.000
MCP-Server-Definitionen:    ~500–2.000 pro Server
Skill/Agent-Descriptions:   ~100–500
```

Jeder Tool-Aufruf im Loop kostet zusätzlich. Die History wächst linear — deshalb ist `/compact` keine Notlösung sondern normales Arbeiten.

Praktisch

```bash
# Eine Session = eine Aufgabe
/clear          # Bei neuer unzusammenhängender Aufgabe

# Komprimieren wenn Kontext voll wird
/compact Behalte: [was wichtig ist]. Verwirf: [was weg kann]

# Gezielt Dateien einbinden statt alles laden
@src/auth/login.ts    # besser als: "Schau dir die gesamte src/ an"
```

Token sparen

Maßnahme	Einsparung
Haiku statt Sonnet für Q&A	~5× günstiger
Sonnet statt Opus	~5× günstiger
`/effort low` für einfache Aufgaben	Weniger Thinking-Tokens
Ungenutzte MCPs deaktivieren	Weniger Start-Kontext
`--bare` für Headless-Scripts	Kein Discovery-Overhead

Maschinenlesbar vs. menschenlesbar

Claude parst JSON deterministisch, Markdown probabilistisch. Für Pipelines:

BASH

```
# In Markdown denken, dann als JSON extrahieren
claude -p "
Analysiere die Fehler-Logs (Markdown).
Schreibe dann:
\`\`\`json
{ \"root_cause\": \"...\", \"severity\": \"critical|high|medium|low\" }
\`\`\`
" < error.log
```

19. Praktische Empfehlungen

Aufgaben aufteilen:

```
❌ "Fixe die Bugs, verbessere die UI und optimiere die Performance"

✅ Eine Aufgabe pro Session:
   "Fixe den Crash in UserForm bei leerer E-Mail.
   Danach warte auf mein OK."
```

Verifizierung einbauen:

```
Fixe den Bug und schreibe einen Test der beweist dass er weg ist.
Starte danach `pnpm build` und zeige mir das Ergebnis.
```

Plan Mode für komplexe Aufgaben:

```bash
# Shift+Tab → Plan Mode
/plan Stripe-Subscriptions mit Webhooks integrieren
# Plan reviewen → genehmigen → Claude führt aus
```

Git Worktrees für parallele Agents:

```bash
git worktree add ../feature-auth feature/auth
git worktree add ../feature-payments feature/payments
# Jeder Agent in seinem eigenen Verzeichnis – keine Konflikte
```

TypeScript statt JavaScript: Typfehler kommen sofort — weniger Debugging-Iterationen, weniger Tokens.

Deployment

20. GitHub-Integration

```bash
/install-github-app
```

Danach:

- `@claude` in PRs oder Issues → Claude macht Änderungen und Commits
- GitHub Actions für automatische Reviews

```yaml
# .github/workflows/claude-review.yml
name: Claude Code Review
on:
  pull_request:
    types: [opened, synchronize]
jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: anthropics/claude-code-action@v1
        with:
          anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
          mode: review
```

PR-Beschreibung automatisch:

```bash
git diff main..HEAD | claude -p "Schreibe eine PR-Beschreibung"
```

21. Produktion & CI/CD

Queue + Worker für lange Tasks:

PYTHON

```
async def worker():
    while True:
        task = await queue.get()
        agent = Agent(session_id=task["session_id"])
        result = await agent.run(task["prompt"])
        await notify_webhook(task["callback_url"], result)
```

Docker für Isolation:

DOCKERFILE

```
FROM node:20-alpine
RUN npm install -g @anthropic-ai/claude-code
```

BASH

```
docker run --rm \
  --network none \
  --read-only \
  --tmpfs /tmp \
  -v $(pwd):/workspace:ro \
  claude-agent \
  claude --bare -p "Analysiere diesen Code"
```

In GitHub Actions:

BASH

```
claude -p "Security-Review der geänderten Dateien" \
  --permission-mode bypassPermissions \
  --output-format json
```

Internals & Experten-Wissen

22. Wie Claude Code intern arbeitet

Wer versteht was under the hood passiert, kann seine Konfiguration gezielt optimieren statt blind zu probieren.

22.1 Startup-Sequenz

Wenn du `claude` tippst:

```
1. Claude Code startet (Node.js)
        ↓
2. Settings laden (Priorität hoch → niedrig):
   Managed Policy > User-Settings > Local-Settings > Project-Settings
        ↓
3. MCP-Server starten
        ↓
4. Skills & Agents entdecken
   → Nur NAME und DESCRIPTION werden geladen — nicht der volle Inhalt
        ↓
5. Commands entdecken
        ↓
6. CLAUDE.md laden:
   a) User-global (~/.claude/CLAUDE.md)
   b) Projekt-Root CLAUDE.md + @-Imports sofort einfügen
   c) Verzeichnis-spezifische CLAUDE.md überschreibt übergeordnete
        ↓
7. MEMORY.md laden (erste 200 Zeilen)
        ↓
8. SessionStart-Hooks ausführen
        ↓
9. Bereit
```

22.2 Wie der System-Prompt aufgebaut wird

Claude sieht keinen "leeren" Kontext. Bevor dein erster Prompt kommt, ist schon ein langer System-Prompt zusammengebaut:

```
[1] Anthropics Basis-Anweisungen (unveränderlich)
[2] User-CLAUDE.md (~/.claude/CLAUDE.md)
[3] Projekt-CLAUDE.md + @-Imports
[4] Relevante .claude/rules/ Dateien
[5] MEMORY.md (erste 200 Zeilen)
[6] Aktive Skill-Descriptions (nur Namen, kein Inhalt)
[7] Aktive Agent-Descriptions (nur Namen)
[8] --append-system-prompt (falls gesetzt)
```

```
[9] DEIN PROMPT
```

Die ersten 30 Zeilen deiner CLAUDE.md stehen an Position [3] relativ weit oben — deshalb bekommen sie mehr Gewicht als der Rest.

22.3 Der Agentic Loop im Detail

```
Du: "Fixe den Bug in login.ts"
                ↓
KI-Aufruf #1: "Ich brauche Kontext. Lese login.ts."
                → gibt zurück: tool_use: Read
                ↓
[PreToolUse Hook läuft]
[Permission Check]
                ↓
Tool: Read("login.ts") → gibt Dateiinhalt zurück
                ↓
[PostToolUse Hook läuft]
                ↓
KI-Aufruf #2: sieht Inhalt. "Zeile 47 ist das Problem. Grep für ähnliche
Muster."
                → gibt zurück: tool_use: Bash(grep)
                ↓
[... Loop wiederholt sich ...]
                ↓
KI-Aufruf #N: "Genug Kontext. Ich schreibe den Fix."
                → gibt zurück: tool_use: Edit("login.ts", ...)
                ↓
[PreToolUse Hook: protect-env.sh?]
[PermissionRequest: User bestätigt]
                ↓
Edit ausgeführt
                ↓
[PostToolUse Hook: prettier läuft]
                ↓
Finaler KI-Aufruf: "Fertig. Bug in Zeile 47 gefixt..."
                → gibt Text zurück, kein tool_use
                ↓
[Stop Hook: Sound, Tests]
                ↓
Du siehst das Ergebnis
```

Jedes `tool_use` ist ein eigener KI-Aufruf. Claude entscheidet selbst welches Tool es als nächstes braucht.

22.4 Skill- und Agent-Discovery

```
Session startet
```

```
      ↓
Claude lädt NUR Metadaten:
  skills/ios-ux-review/SKILL.md  → name + description
  agents/swift-reviewer.md       → name + description
      ↓
Du: "Kannst du mal den Antrag-Screen reviewen?"
      ↓
Claude vergleicht Anfrage mit Descriptions:
  ios-ux-review: "Prüft SwiftUI-Views auf HIG. Trigger: UX-Review..." → MATCH
      ↓
Erst JETZT: SKILL.md vollständig geladen (lazy loading)
      ↓
Skill-Hooks aktivieren (temporär)
Claude arbeitet im Skill-Modus
      ↓
Skill endet → Hooks entfernt, Kontext gesäubert
```

Deshalb ist die `description` im Frontmatter so wichtig — sie ist das einzige was Claude kennt bevor es entscheidet ob der Skill passt.

22.5 Wie Hooks den Fluss unterbrechen

```
Claude will Edit("prod.env") aufrufen
            ↓
Alle passenden PreToolUse-Hooks laufen parallel:
  Hook 1 (user-level):    → allow
  Hook 2 (project-level): → ask
  Hook 3 (plugin):        → deny
            ↓
Stringstes Ergebnis gewinnt: DENY
            ↓
Edit wird nicht ausgeführt
Claude: "Operation abgelehnt: prod.env ist schreibgeschützt"
Claude passt sein Verhalten an
```

22.6 Kontext-Hierarchie

```
~/.claude/CLAUDE.md          ← persönliche Gewohnheiten
        +
Projekt CLAUDE.md            ← Team-Standards
        +
.claude/rules/ [passend]     ← Bereichsspezifisch
        +
MEMORY.md (200 Zeilen)       ← Claudes Notizen
        =
Was Claude über das Projekt "weiß"

Priorität bei Konflikten:
Managed Policy > User > Local > Project
```

```
Nähere CLAUDE.md überschreibt entferntere:
/app/dashboard/CLAUDE.md > /app/CLAUDE.md > /CLAUDE.md
```

22.7 Was passiert bei `/compact`

```
/compact [Fokus]
      ↓
Separater KI-Aufruf (außerhalb des aktuellen Kontexts)
      ↓
Dieser "Compaction-Agent" liest Session-History
und schreibt eine Zusammenfassung
      ↓
Ohne Fokus: generische Zusammenfassung
Mit Fokus: priorisierte Zusammenfassung
      ↓
Zusammenfassung ersetzt History
Neue Session startet damit als Basis
```

Da der Compaction-Agent kein Wissen über deine Prioritäten hat, ist die Fokus-Anweisung wichtig.

Alternativ selbst kontrollieren:

```
Erstelle ein Handoff-Dokument als handoff.md mit:
- Was wurde gemacht
- Getroffene Entscheidungen und Gründe
- Offene Aufgaben
- Bekannte Probleme
```

22.8 Alles auf einen Blick

```
KONFIGURATION (statisch, vor Session):
  CLAUDE.md + rules/ + settings.json
         ↓
KONTEXT-AUFBAU (beim Start):
  User-CLAUDE.md + Projekt-CLAUDE.md + MEMORY.md + Skill-Descriptions (lazy)
         ↓
LAUFZEIT (während Session):
  Prompt → KI denkt → Tool-Call → PreHook → Permission
         → Ausführung → PostHook → Ergebnis → Nächster KI-Call
         ↓
LAZY LOADING (bei Bedarf):
  Skill-Inhalt wenn erkannt
  Agent wenn aufgerufen
  Topic-Memory wenn gebraucht
         ↓
SESSION-ENDE:
  Stop-Hooks laufen
```

```
Memory aktualisiert
Session auf Disk (für /resume)
```

23. Tipps die kaum jemand kennt

23.1 Conditional Hooks mit `if`

JSON

```
{
  "hooks": {
    "PostToolUse": [{
      "matcher": "Edit|Write",
      "hooks": [
        {
          "type": "command",
          "if": "echo $CLAUDE_FILE_PATH | grep -q '\\.py$'",
          "command": "ruff format $CLAUDE_FILE_PATH"
        },
        {
          "type": "command",
          "if": "echo $CLAUDE_FILE_PATH | grep -qE '\\.(ts|tsx)$'",
          "command": "pnpm prettier --write $CLAUDE_FILE_PATH"
        }
      ]
    }]
  }
}
```

23.2 `once: true` — Hook nur einmal pro Session

JSON

```
{
  "hooks": {
    "SessionStart": [{
      "hooks": [{
        "type": "command",
        "command": "echo 'Session gestartet' | say",
        "once": true
      }]
    }]
  }
}
```

23.3 `statusMessage` — Claude zeigt Fortschritt

JSON

```
{
  "hooks": {
    "PreToolUse": [{
```

```
      "matcher": "Bash",
      "hooks": [{
        "type": "command",
        "command": "./validate-cmd.sh",
        "statusMessage": "Befehl wird auf Sicherheit geprüft..."
      }]
    }]
  }
}
```

23.4 `IMPORTANT:` und `YOU MUST` in CLAUDE.md

Für Regeln die wirklich nie gebrochen werden sollen:

MARKDOWN

```
IMPORTANT: Verwende niemals `any` in TypeScript.

YOU MUST ask for confirmation before deleting any file.

CRITICAL: API Keys gehören in .env — niemals hartcodiert.
```

Diese Formulierungen erhöhen die Priorität im Modell.

23.5 `.claude/rules/` mit Glob-Filter

MARKDOWN

```
---
# .claude/rules/migrations.md
globs: ["**/migrations/**", "**/db/**"]
---

# Migrations-Regeln
- Immer reversible Migrations (up + down)
- Keine Spalten löschen ohne vorherige nullable-Phase
```

Lädt nur wenn Claude in `migrations/` oder `db/` Dateien arbeitet. Spart Kontext für alles andere.

23.6 Claude als Git Pre-Commit-Hook

BASH

```
# .git/hooks/pre-commit
#!/usr/bin/env bash
STAGED=$(git diff --cached --name-only | grep -E '\.(ts|tsx|py)$')
if [ -n "$STAGED" ]; then
  echo "$STAGED" | claude -p \
    "Security-Check. Bei kritischen Issues: exit 1. Sonst: 'LGTM' und exit
```

```
0." \
    --bare \
    --permission-mode bypassPermissions
fi
```

23.7 `CLAUDE.local.md` für persönliche Overrides

MARKDOWN

```
# CLAUDE.local.md — automatisch in .gitignore, nie in Git

## Meine lokale Umgebung
- Lokale DB: postgresql://localhost:5432/myapp_dev
- Test-User: test@example.com / passwort123
- Mein Staging: https://staging-xyz.myapp.dev

## Meine Preferences
- Immer verbose Ausgaben beim Debugging
- Zeig mir den Diff bevor du eine Datei änderst
```

23.8 Alle Hook-Input-Felder per `jq` auslesen

BASH

```
#!/usr/bin/env bash
# Tool-Call-Daten aus stdin lesen
INPUT=$(cat)
echo "Tool:    $(echo $INPUT | jq -r '.tool_name')"
echo "Datei:   $(echo $INPUT | jq -r '.tool_input.file_path // "n/a"')"
echo "Command: $(echo $INPUT | jq -r '.tool_input.command // "n/a"')"
```

Verfügbare Felder:

```
Read:  tool_input.file_path
Write: tool_input.file_path, tool_input.content
Edit:  tool_input.file_path, tool_input.old_string, tool_input.new_string
Bash:  tool_input.command
Grep:  tool_input.pattern, tool_input.path
```

23.9 Mehrstufige Pipeline-Workflows

BASH

```
# Git-Workflow automatisiert
git diff --staged | \
  claude -p "Schreibe eine Commit-Message" --output-format json | \
  jq -r '.result' | \
```

```
  git commit -F -

# Code → Review → Issue
cat src/auth.ts | \
  claude -p "Security-Review als JSON: {issues: [{line, severity,
description}]}" \
         --output-format json | \
  jq '.result' | \
  gh issue create --title "Security Review: auth.ts" --body-file -
```

23.10 Plan Mode + Worktree = sichere Experimente

BASH

```
git worktree add ../experiment experiment/new-feature
cd ../experiment
claude
/plan Refactore das gesamte Auth-System auf JWT

# Plan schlecht: Worktree löschen, kein Schaden
git worktree remove ../experiment
```

23.11 Das Handoff-Dokument-Muster

Gibt mehr Kontrolle als `/compact`:

BASH

```
# Am Ende einer Session:
Erstelle handoff.md mit:
1. Was diese Session gemacht hat
2. Architektur-Entscheidungen mit Begründung
3. Offene Aufgaben (priorisiert)
4. Bekannte Fallstricke

# Neue Session:
@handoff.md Was ist als nächstes zu tun?
```

23.12 Claude verbessert seine eigene Konfiguration

BASH

```
# Patterns aus der Session in CLAUDE.md aufnehmen
Analysiere unsere heutige Session und schlage Ergänzungen für CLAUDE.md vor.
Fokus: Konventionen die mehrfach aufgetaucht sind, Fallstricke die wir
hatten.

# Rules automatisch erstellen
Erstelle .claude/rules/testing.md aus den Test-Diskussionen dieser Session.
```

```
# Muster erkennen
/insights
# Dann: Welche meiner häufigen Fragen gehören direkt in CLAUDE.md?
```

23.13 `max-Turns` setzen

JSON

```
{ "maxTurns": 20 }
```

Zu niedrig: Claude kommt nicht zum Ziel. Zu hoch: unkontrolliertes Loopen und Token-Verschwendung.

23.14 Agent SDK für vollautomatische Pipelines

PYTHON

```
from anthropic import Agent
import asyncio

async def auto_review(pr_diff: str) -> dict:
    agent = Agent(
        system="Code-Reviewer. Analysiere den Diff und gib JSON zurück.",
        tools=["Read", "Grep"],
    )
    return await agent.run(
        f"Review:\n{pr_diff}\nOutput: {{issues: [], verdict:
'approve'|'request_changes'}}"
    )

diff = subprocess.check_output(["git", "diff", "main..HEAD"])
review = asyncio.run(auto_review(diff.decode()))
```

23.15 Schnellste Diagnose bei Problemen

BASH

```
/doctor                          # System-Check
/context                         # Was ist im Kontext?
/permissions                     # Welche Tools sind erlaubt?
CLAUDE_DEBUG=1 claude            # Debug-Output
claude --bare --disallowedTools Bash  # Isoliert testen
```

Anhang

A — .claude/ Ordnerstruktur

```
KK-App/
├── CLAUDE.md                          ← Projektanweisungen (in Git)
├── CLAUDE.local.md                    ← Persönlich (NICHT in Git)
│
├── Features/                          ← Feature-Module
│   ├── Login/
│   ├── Dashboard/
│   ├── Postfach/
│   ├── Antrag/
│   ├── Bescheinigung/
│   ├── Bonus/
│   ├── Gesundheit/
│   ├── Kontakt/
│   ├── MeineDaten/
│   └── Einstellungen/
│
├── Core/                              ← Auth, Netzwerk, Persistence
├── Shared/                            ← Design-System, Extensions
├── Resources/                         ← Lokalisierung, Assets
├── api/openapi.yml                    ← API-Spezifikation
│
└── .claude/
    ├── settings.json                  ← Hooks, MCP (in Git)
    ├── settings.local.json            ← Lokal (NICHT in Git)
    │
    ├── rules/                         ← Modulare Regeln
    │   ├── swift-style.md             ← Swift-Coding-Standards
    │   ├── datenschutz.md             ← DSGVO/SGB-V-Regeln
    │   └── barrierefreiheit.md        ← Accessibility-Anforderungen
    │
    ├── commands/                      ← Eigene Slash-Commands
    │   ├── review-pr.md               → /review-pr
    │   ├── check-barrierefreiheit.md → /check-barrierefreiheit
    │   ├── neuer-screen.md            → /neuer-screen
    │   └── erklaer.md                 → /erklaer
    │
    ├── skills/                        ← Komplexe Workflows
    │   ├── ios-ux-review/
    │   │   ├── SKILL.md
    │   │   └── examples/
    │   └── datenschutz-audit/
    │       ├── SKILL.md
    │       └── speicherorte-checklist.md
    │
    ├── agents/                        ← Sub-Agenten
    │   ├── swift-reviewer.md
    │   ├── datenschutz-prufer.md
```

```
    │   ├── api-doc-writer.md
    │   └── release-orchestrator.md
    │
    └── hooks/                          ← Automation-Scripts
        ├── protect-health-data.sh
        ├── protect-config.sh
        └── block-dangerous.sh
```

Entscheidungshilfe:

Bedarf	Lösung
Globale Projektregeln	`CLAUDE.md`
Bereichsspezifische Regeln	`.claude/rules/[bereich].md`
Schneller wiederverwendbarer Befehl	`.claude/commands/[name].md`
Komplexer mehrstufiger Workflow	`.claude/skills/[name]/SKILL.md`
Spezialisierte isolierte Rolle	`.claude/agents/[name].md`
Erzwungenes Verhalten	`.claude/settings.json` → hooks
Externe Tool-Verbindung	`.claude/settings.json` → mcpServers

B — Umgebungsvariablen Referenz

BASH

```
ANTHROPIC_API_KEY=sk-ant-...            # API Key (erforderlich)
ANTHROPIC_BASE_URL=...                  # Alternativer Endpoint
ANTHROPIC_MODEL=claude-sonnet-4-6     # Standard-Modell
CLAUDE_CODE_MAX_OUTPUT_TOKENS=8192    # Max Output-Tokens
MAX_THINKING_TOKENS=10000             # Max Thinking-Tokens
CLAUDE_CODE_DISABLE_AUTO_MEMORY=1     # Auto-Memory aus
DISABLE_AUTOUPDATER=1                 # Keine Auto-Updates
CLAUDE_CODE_DISABLE_TELEMETRY=1       # Keine Telemetrie
CLAUDE_CODE_USE_BEDROCK=1             # AWS Bedrock
CLAUDE_CODE_USE_VERTEX=1              # Google Vertex AI
AWS_REGION=us-east-1                  # Für Bedrock
CLOUD_ML_REGION=us-east5              # Für Vertex
ANTHROPIC_VERTEX_PROJECT_ID=...       # GCP Projekt
CLAUDE_DEBUG=1                        # Debug-Output
```

C — Checkliste

Neues Projekt:

- [ ] `claude` + `/init` ausführen
- [ ] CLAUDE.md reviewen und erweitern
- [ ] Stack, Architektur, Commands, Hard Rules rein
- [ ] Unter 200 Zeilen halten
- [ ] `git init` (für Auto-Memory)

Täglich:

- [ ] Richtiges Modell und Effort für die Aufgabe
- [ ] `/plan` für komplexe Aufgaben
- [ ] Eine Aufgabe pro Session
- [ ] `@Features/Antrag/AntragView.swift` für gezielten Kontext
- [ ] `/clear` bei neuem Thema
- [ ] `/compact` mit Fokus wenn Kontext voll

Team-Setup (KK-App):

- [ ] `.claude/commands/` mit Team-Workflows
- [ ] `.claude/rules/` für Swift-Style, Datenschutz, Accessibility
- [ ] Hooks für SwiftLint und Gesundheitsdaten-Schutz
- [ ] `settings.json` in Git (ohne `settings.local.json`)
- [ ] `datenschutz-prufer` Agent als Pre-Release-Pflicht definiert

Sicherheit:

- [ ] Auto Mode konfiguriert mit `$defaults`
- [ ] `hard_deny` für kritische Aktionen
- [ ] Skills nur aus vertrauenswürdigen Quellen

D — CLAUDE.md Template (KK-App)

Das ist das vollständige Template für die Krankenkassen iOS-App. Als Startpunkt kopieren, dann projektspezifisch anpassen.

MARKDOWN

```
# KK-App – [Name der Krankenkasse] iOS

## Übersicht
Native iOS-App für Versicherte der [Kasse].
Kernfunktionen: IAM-Login, Dashboard, Postfach, Anträge,
Bescheinigungen, Bonusprogramm, Gesundheit, Kontakt.
Zielgruppe: alle Versicherungsaltersgruppen.
Barrierefreiheit ist gesetzlich vorgeschrieben (BGG, BITV 2.0).

## Stack
- Swift 5.9 + SwiftUI (iOS 17+)
- Xcode 16
- OAuth2/OIDC via [IAM-System, z.B. Keycloak]
- REST API (Spec: api/openapi.yml)
- SwiftData für lokalen Cache (offline-fähig)
- Keychain für Tokens und PII

Nicht verwenden: Combine, UIKit direkt, Third-Party-Auth-Libs.

## Ordnerstruktur
- `Features/` → Login, Dashboard, Postfach, Antrag,
                Bescheinigung, Bonus, Gesundheit,
                Kontakt, MeineDaten, Einstellungen
- `Core/` → Netzwerk, Auth/IAM, Persistence, Keychain
- `Shared/` → Design-System, Komponenten, Extensions
- `Resources/` → Lokalisierung (de, en), Assets
Konfiguration → Config.xcconfig (nie in Git mit Secrets!)

## Regeln die nie gebrochen werden
- Kein force-unwrap (!) außer @IBOutlet
- Gesundheitsdaten (SGB V) nur in Keychain / verschlüsselt
- Alle UI-Elemente mit .accessibilityLabel() + .accessibilityHint()
- Keine neuen Dependencies ohne Rückfrage
- Nur die Dateien anfassen die zur Aufgabe gehören

## Was nie passieren soll
- Token, KVNR, Diagnosen in Logs (print/NSLog/os_log ohne .private)
- Netzwerk ohne Certificate Pinning in Production-Builds
- Echte Versichertendaten als Testdaten im Code
- Direkt auf main pushen ohne PR und Review

## Commands
- Build:  `xcodebuild -scheme KKApp -destination
           'platform=iOS Simulator,name=iPhone 16'`
- Test:   `xcodebuild test -scheme KKAppTests`
- Lint:   `swiftlint lint --strict`
```

```
## Wichtige Hinweise
- IAM-Token läuft nach 15 min ab → RefreshToken in
  Core/Auth/TokenManager.swift (nicht neu implementieren!)
- Antrag-Upload: max. 10 MB, nur PDF/JPG/PNG
- Postfach offline-fähig: SwiftData-Cache in Core/Persistence/
- Bescheinigungen: als Signed URL, kein Base64 im Response

## Compact Instructions
Erhalten:
- Offene IAM/Auth-Fragen und Token-Refresh-Entscheidungen
- Laufende Feature-Implementation mit State
- Fehlschlagende Tests und ihre Fehlermeldungen
- Datenschutz-relevante Entscheidungen

Verwerfen:
- Explorative Versuche die verworfen wurden
- Gelöste Debugging-Sessions
- CSS/Layout-Experimente ohne Ergebnis
```

Stand: Mai 2026 · Claude Code v2.1+