Claude Code als Senior-Dev einrichten

Für Entwickler und Solopreneure, die Claude Code bereits installiert haben und es jetzt produktiv als autonomen Engineering-Agenten nutzen wollen

UPRO Capital · white-label Kurs

Inhalt

1. Fundament: CLAUDE.md als Projektbriefing

1.1 Was CLAUDE.md ist und warum sie entscheidend ist

Claude Code startet jede Session ohne Erinnerung. Kein Kontext, kein Stilwissen, keine Kenntnis deines Stacks — es sei denn, du lieferst beides explizit. Genau das ist die Aufgabe der CLAUDE.md: Sie ist das schriftliche Briefing, das Claude Code vor jeder Antwort automatisch liest.

Ohne CLAUDE.md verhält sich Claude wie ein fähiger, aber völlig uninstruierter Freelancer am ersten Tag. Er kennt deinen Coding-Stil nicht, weiß nicht welches Framework du nutzt, kennt keine Deploy-Kommandos und macht Annahmen — die meistens falsch sind. Mit einer gut geschriebenen CLAUDE.md ist er ein eingearbeiteter Senior-Dev: Er weiß, dass du TypeScript strict ohne `any` schreibst, dass du pnpm nutzt, dass Deploy via `wrangler pages deploy dist/` läuft und dass du Kommentare auf Deutsch erwartest.

Es gibt zwei Ebenen: Die globale CLAUDE.md liegt unter `~/.claude/CLAUDE.md` und gilt für alle Projekte. Dort definierst du einmalig deine persönlichen Präferenzen — Sprache, Coding-Stil, Tonfallregeln, Dinge die du grundsätzlich nie willst. Die projektspezifische CLAUDE.md liegt im Wurzelverzeichnis des Repos und enthält alles Projekt-konkrete: Stack, Datenbankschema-Hinweise, welche Ports lokal laufen, exakte Build- und Deploy-Befehle.

Beide Dateien werden von Claude Code automatisch kombiniert geladen. Die projektspezifische überschreibt im Konfliktfall die globale. Der Lese-Aufwand für dich ist null — einmal geschrieben, permanent aktiv. Wer diese Datei leer lässt oder ignoriert, verschenkt den größten Hebel für konstante Ausgabequalität.

1. 1. Prüfe ob `~/.claude/CLAUDE.md` existiert: `ls ~/.claude/CLAUDE.md`. Falls nicht, erstelle sie mit `touch ~/.claude/CLAUDE.md`.
2. 2. Trage in die globale CLAUDE.md deine persönlichen Präferenzen ein: bevorzugte Sprache für Kommentare, Regeln die immer gelten (kein `any`, kein `console.log` in Commits), Dinge die du nie willst.
3. 3. Prüfe ob im aktuellen Projekt eine projektspezifische CLAUDE.md im Wurzelverzeichnis liegt: `ls CLAUDE.md`.
4. 4. Verifiziere, dass Claude Code beide Dateien erkennt, indem du im Chat fragst: 'Welchen Stack nutzt dieses Projekt laut deiner Konfiguration?' — Claude muss die Infos aus deiner CLAUDE.md korrekt nennen.

> **PROMPT**
>
> ```
> Lies meine CLAUDE.md und bestätige, was du über dieses Projekt weißt. Liste
> Stack, Deploy-Kommando und die wichtigsten Coding-Regeln auf.
> ```

PROMPT

```
Überprüfe ob die bestehende CLAUDE.md vollständig ist. Was fehlt für einen
neuen Contributor, der sofort produktiv werden soll?
```

ÜBUNG

Öffne dein aktuelles Projekt. Erstelle eine projektspezifische CLAUDE.md im Wurzelverzeichnis. Trage folgende Punkte ein: (1) genutzter Stack mit exakten Versionen, (2) Package-Manager-Befehl für Installation, (3) den exakten Befehl zum lokalen Starten der Dev-Umgebung, (4) den exakten Deploy-Befehl, (5) eine Regel zum Coding-Stil (z.B. 'Immer TypeScript strict, kein any'). Teste danach im Claude Code Chat: 'Wie deploye ich dieses Projekt?' — Claude soll den korrekten Befehl aus deiner CLAUDE.md nennen.

▶ Lösung

1.2 CLAUDE.md mit /init und eigenem Prompt aufbauen

Das `/init`-Kommando ist der schnellste Einstieg für neue Projekte: Claude scannt selbstständig das Verzeichnis — Package.json, Konfigurationsdateien, vorhandene Skripte — und generiert daraus eine erste CLAUDE.md. Das dauert wenige Sekunden und liefert eine strukturierte Basis, die deutlich besser ist als eine leere Datei.

Der Haken: Die auto-generierte CLAUDE.md ist generisch. Sie enthält, was Claude aus den Dateien lesen kann — aber nicht, was du bevorzugst, was dein Deploy-Prozess ist oder welche Konventionen in deinem Team gelten. Sie ist ein Startpunkt, kein Endprodukt. Wer sie unverändert lässt, bekommt generische Ausgaben statt personalisierter.

Der richtige Workflow ist zweiteilig: Erst `/init` ausführen lassen, dann sofort einen personalisierten Prompt hinterherschicken, der die Datei ergänzt. Dieser Prompt enthält alles, was Claude nicht aus den Projektdateien ablesen kann: deinen bevorzugten Coding-Stil, welche Patterns du verbietest, wie dein Deployment-Prozess aussieht und welche Fehler Claude in diesem Projekt besonders vermeiden soll.

Praktische Faustregel: Alles, was du einem neuen Entwickler im Onboarding erklärt hättest, gehört in die CLAUDE.md. Alles, was selbstverständlich klingt ('natürlich nutzen wir TypeScript strict'), ist genau das, was Claude Code ohne explizite Anweisung weglässt.

1. 1. Öffne Claude Code im Wurzelverzeichnis deines Projekts.
2. 2. Tippe `/init` und bestätige. Claude scannt das Verzeichnis und schreibt eine erste CLAUDE.md.
3. 3. Lies die generierte Datei durch: Was stimmt, was fehlt, was ist falsch?
4. 4. Schicke direkt im Anschluss einen Personalisierungs-Prompt (siehe Prompts unten), der alle fehlenden Punkte ergänzt.
5. 5. Prüfe die aktualisierte CLAUDE.md: `cat CLAUDE.md`. Teste mit einer konkreten Frage im Chat, ob Claude die neuen Regeln kennt.
6. 6. Committe die CLAUDE.md ins Repo — sie gehört in die Versionskontrolle, nicht in `.gitignore`.

PROMPT

```
Ergänze die bestehende CLAUDE.md mit folgenden Informationen, die du nicht
aus den Projektdateien lesen konntest:

Stack-Details: [dein Stack mit Versionen]
Package-Manager: [pnpm/npm/yarn]
Dev-Kommando: [z.B. pnpm dev]
Build-Kommando: [z.B. pnpm build]
Deploy-Kommando: [z.B. wrangler pages deploy dist/]
Coding-Regeln: [deine Regeln]
Verbotene Patterns: [was du nie willst]
Projekt-Konventionen: [wie Komponenten benannt werden, etc.]

Halte die Formatierung der bestehenden Datei bei.
```

PROMPT

```
Generiere eine vollständige CLAUDE.md für dieses Projekt. Nutze alle Infos,
die du aus den vorhandenen Dateien lesen kannst, und frage mich für alles
weitere, was du nicht ableiten kannst.
```

ÜBUNG

Führe `/init` in einem Projekt aus, das du regelmäßig nutzt. Lies die generierte CLAUDE.md. Identifiziere mindestens 3 Punkte, die fehlen oder falsch sind. Schicke dann den Personalisierungs-Prompt aus dem Prompts-Abschnitt und ergänze alle fehlenden Punkte. Teste abschließend mit der Frage 'Was sind die wichtigsten Coding-Regeln in diesem Projekt?' — Claude soll deine eigenen Regeln nennen, nicht generische.

▶ Lösung

2. Modell und Effort gezielt steuern

2.1 Modell per Slash-Command im aktiven Chat wechseln

Claude Code startet standardmäßig mit einem mittleren Modell — sinnvoll für Routineaufgaben, aber nicht die stärkste Option für Architekturentscheidungen oder schwieriges Debugging. Das Modell lässt sich jederzeit im laufenden Chat per Slash-Command wechseln, ohne den Context zu verlieren.

Die Entscheidung, wann welches Modell sinnvoll ist, folgt einer klaren Regel: Wenn du weißt, was zu tun ist, und Claude es nur ausführen soll — Standard-Modell. Wenn du nicht weißt, wie du ein Problem angehen sollst, wenn Architekturentscheidungen mit mehreren gültigen Optionen anstehen oder wenn Claude zweimal hintereinander eine falsche Lösung geliefert hat — dann wechsle auf das stärkste Modell mit maximalem Effort.

Ein konkretes Signal für den Modellwechsel: Wenn Claude dieselbe Frage, die du gerade gestellt hast, beim zweiten Anlauf mit einer komplett anderen Herangehensweise beantwortet — das zeigt, dass das Standardmodell an seiner Kapazitätsgrenze arbeitet und stärker rät als denkt. Das stärkste Modell auf Effort Max kostet ca. 5× mehr pro Token, liefert aber bei komplexen Problemen deutlich präzisere Analysen und seltener 'Halluzinations-Lösungen'.

Für API-Billing-Nutzer ist das Kostenargument real: Einfache Code-Ergänzungen, Refactoring bekannter Patterns, Testschreiben nach vorhandenem Schema — Standard-Modell. Unbekannte Bugs in verteilten Systemen, neue Architekturschichten, sicherheitsrelevante Entscheidungen — stärkstes Modell, gezielt eingesetzt.

1. 1. Öffne Claude Code und starte eine neue Session.
2. 2. Tippe `/` im Chat und schau dir die verfügbaren Modell-Optionen an.
3. 3. Wähle für eine einfache Aufgabe das Standard-Modell — z.B. 'Füge einen Export zu dieser Datei hinzu'.
4. 4. Wechsle per Slash-Command auf das stärkste Modell für eine komplexe Aufgabe — z.B. 'Analysiere warum dieser Auth-Flow bei parallelen Requests einen Race Condition hat'.
5. 5. Beobachte den Unterschied in Antworttiefe und Problemanalyse.
6. 6. Wechsle nach der komplexen Aufgabe zurück auf das Standard-Modell für folgende Routine-Tasks.

PROMPT

```
Wechsle auf das stärkste verfügbare Modell mit maximalem Effort. Bestätige
den Wechsel und analysiere dann: [dein komplexes Problem]
```

PROMPT

```
Analysiere diesen Bug mit maximaler Sorgfalt. Nenne die Ursache, warum sie
auftritt, und mindestens zwei mögliche Fixes mit ihren jeweiligen Tradeoffs:
[Bug-Beschreibung + relevanter Code]
```

ÜBUNG

Stelle Claude Code dieselbe Frage zweimal: einmal mit dem Standard-Modell, einmal mit dem stärksten Modell + Effort Max. Wähle eine Frage, bei der Qualität messbar ist — z.B. 'Erkläre mir die Vor- und Nachteile von Server Components vs. Client Components in Next.js für eine Anwendung mit häufig wechselnden Daten und Auth-Gate.' Vergleiche die beiden Antworten: Welche nennt konkrete Schwellen und Tradeoffs, welche bleibt vage?

▶ Lösung

2.2 Automatischen Modell-Wechsel in CLAUDE.md hinterlegen

Statt bei jeder komplexen Aufgabe manuell das Modell zu wechseln, lässt sich diese Logik direkt in die CLAUDE.md schreiben. Der Mechanismus ist einfach: Du definierst ein Trigger-Kommando (z.B. `/ultrareview`) und die Regel, dass Claude bei diesem Kommando intern auf das stärkste Modell mit maximalem Effort wechseln soll.

Das ist besonders nützlich für wiederkehrende High-Stakes-Momente im Workflow: Code-Reviews vor einem Merge, Security-Checks vor Deploys, Architektur-Reviews bei neuen Features. Du musst dann nicht mehr daran denken, welches Modell aktiv ist — das Trigger-Kommando erledigt beides: den Wechsel und die Aufgabe.

Kosten-Einschätzung pro Nutzungsmodell: Wer Claude Code auf einem Flatrate-Plan nutzt (max. Subscription), hat keinen direkten Token-Kostenvorteil durch das Standard-Modell. Der Vorteil des stärksten Modells überwiegt fast immer — es lohnt sich, es standardmäßig zu nutzen. Wer per API-Billing zahlt, zahlt pro Token: Dort ist die Strategie 'Standard-Modell für Routine, stärkstes nur gezielt für Reviews und schwierige Debugging-Sessions' wirtschaftlich sinnvoll. Das mittlere Modell mit Effort Max ist in diesem Fall oft der beste Preis-Leistungs-Spot — 80% der Kapazität des stärksten Modells zu ~40% des Preises.

Wichtig: CLAUDE.md-Regeln steuern Verhalten, nicht technische Modellparameter direkt. Der Wechsel passiert auf Basis von Claudes Interpretation der Regeln. Formuliere die Regel klar und prüfe sie einmalig mit einer Test-Session.

1. 1. Öffne die CLAUDE.md des Projekts oder die globale `~/.claude/CLAUDE.md`.
2. 2. Füge einen Abschnitt '## Modell-Regeln' ein.
3. 3. Definiere das Trigger-Kommando und die zugehörige Modell-Regel — Beispiel: 'Wenn ich /ultrareview tippe, führe den Review mit dem stärksten verfügbaren Modell und maximalem Effort durch.'
4. 4. Definiere optional eine Regel für Routine-Tasks: 'Für einfache Code-Ergänzungen und Refactoring reicht das Standard-Modell.'
5. 5. Speichere die CLAUDE.md und starte eine neue Claude Code Session (damit die aktualisierte Datei geladen wird).
6. 6. Teste: Tippe `/ultrareview` und beobachte ob Claude den Modellwechsel bestätigt und eine tiefere Analyse liefert.

PROMPT

```
/ultrareview – Analysiere den Code in [Dateiname] mit maximaler Sorgfalt. Prüfe auf: (1) Logikfehler, (2) Performance-Probleme, (3) Sicherheitslücken, (4) fehlende Fehlerbehandlung. Liefere für jeden Fund den genauen Ort und einen konkreten Fix.
```

PROMPT

```
/arch – Ich stehe vor folgender Architekturentscheidung: [Beschreibung]. Liefere zwei konkrete Optionen, deren Tradeoffs, und eine klare Empfehlung für meinen spezifischen Kontext.
```

ÜBUNG

Erweitere deine CLAUDE.md um einen 'Modell-Regeln'-Abschnitt. Definiere mindestens zwei Regeln: (1) ein Trigger-Kommando für das stärkste Modell + Effort Max, (2) eine explizite Regel wann das Standard-Modell ausreicht. Teste die Trigger-Regel in einer neuen Session mit einer echten Code-Review-Aufgabe aus deinem aktuellen Projekt.

▶ Lösung

3. Die 4 Kern-Skills installieren und anwenden

3.1 Skills als SKILL.md-Dateien verstehen und ablegen

Skills sind in Claude Code einfache Markdown-Textdateien mit der Dateiendung SKILL.md. Kein Binary, keine Abhängigkeiten, kein Compilieren. Du legst die Datei in einen Ordner ab, Claude Code liest sie automatisch beim Start — fertig. Das ist der gesamte Installationsprozess.

Der Ablageort entscheidet über den Gültigkeitsbereich: Globale Skills, die in jedem Projekt gelten sollen, kommen nach `~/.claude/skills/<skill-name>/SKILL.md`. Projekt-spezifische Skills, die nur in einem bestimmten Repo gelten sollen, kommen nach `./.claude/skills/<skill-name>/SKILL.md` — also in den `.claude`-Ordner im Projektverzeichnis.

Der Inhalt einer SKILL.md definiert: welche Slash-Commands der Skill einführt, wie Claude sich bei diesen Commands verhalten soll, welche Ausgabeformate erwartet werden und welche Regeln gelten. Claude Code lädt alle gefundenen SKILL.md-Dateien beim Sessionstart und merkt sich die darin definierten Verhaltensregeln für den gesamten Chat.

Verifizierung nach der Installation: Der einfachste Test ist `ls -la ~/.claude/skills/`. Damit siehst du alle global installierten Skills. Für einen funktionstüchtigen Test in der Chat-Session: Tippe das Trigger-Wort des installierten Skills und prüfe, ob Claude das erwartete Verhalten zeigt — wenn ja, wurde der Skill korrekt geladen. Wenn Claude das Trigger-Wort nicht erkennt, prüfe den Dateinamen: Er muss exakt `SKILL.md` heißen, nicht `skill.md` oder `Skill.md`.

1. 1. Erstelle den Skills-Ordner falls er noch nicht existiert: `mkdir -p ~/.claude/skills/`.
2. 2. Für einen globalen Skill: Erstelle den Unterordner `mkdir -p ~/.claude/skills/<skill-name>/` und lege die SKILL.md dort ab.
3. 3. Für einen projekt-spezifischen Skill: `mkdir -p ./.claude/skills/<skill-name>/` und SKILL.md dort ablegen.
4. 4. Starte eine neue Claude Code Session (bestehende Sessions erkennen nachträglich abgelegte Skills nicht immer zuverlässig).
5. 5. Verifiziere die Installation: `ls -la ~/.claude/skills/` für globale, `ls -la ./.claude/skills/` für lokale Skills.
6. 6. Teste den Skill im Chat: Tippe das im Skill definierte Trigger-Wort und prüfe ob Claude korrekt reagiert.

PROMPT

```
Lies alle SKILL.md-Dateien die du in dieser Session geladen hast und liste
alle verfügbaren Slash-Commands auf, die durch diese Skills definiert
werden.
```

PROMPT

```
Erstelle eine SKILL.md für folgenden Skill und lege sie unter
~/.claude/skills/[name]/SKILL.md ab: [Skill-Beschreibung]. Der Skill soll
durch das Trigger-Wort '[trigger]' ausgelöst werden.
```

ÜBUNG

Erstelle deinen ersten eigenen Mini-Skill. Ziel: Ein Skill, der beim Trigger '/checklist' eine Deployment-Checkliste für dein Projekt ausgibt. Erstelle die SKILL.md unter `~/.claude/skills/deploy-checklist/SKILL.md` mit folgendem Inhalt: Trigger `/checklist`, Ausgabe: eine nummerierte Liste mit deinen 5 wichtigsten Pre-Deploy-Checks (z.B. 'TypeScript-Fehler prüfen', 'Tests laufen lassen', 'ENV-Variablen gesetzt?'). Teste den Skill in einer neuen Claude Code Session.

▶ Lösung

3.2 Superpowers: Von /brainstorm bis atomarem Commit

Der Superpowers-Skill ist ein agentic Framework mit über 20 vorgefertigten Slash-Commands — er verwandelt Claude Code von einem Chatbot, der Code auf Anfrage schreibt, in einen Software-Ingenieur, der erst plant und dann baut. Der entscheidende Unterschied: Ohne Framework greift Claude sofort zum Code. Mit Superpowers startet er mit Rückfragen, baut einen strukturierten Plan und implementiert erst, wenn der Plan genehmigt ist.

Der häufigste Fehler beim Einstieg: Den Plan überspringen und direkt `/execute` schicken. Das Ergebnis ist strukturell falscher Code, weil Claude die Anforderungen nie vollständig verstanden hat. Der richtige Einstieg ist immer `/brainstorm [Feature-Beschreibung]`. Claude stellt dann konkrete Rückfragen — nicht vage ('Was willst du?'), sondern gezielt ('Soll die Auth stateless oder session-based sein?'). Diese Rückfragen sind wertvoller als man denkt: Sie decken Annahmen auf, die du selbst noch nicht explizit gemacht hast.

Nach dem Brainstorming schreibt Claude eine `PLAN.md` im Projektverzeichnis. Lies sie kritisch. Ein häufiges Problem: Der Plan ist strukturell korrekt, aber in der falschen Reihenfolge — z.B. API-Endpunkt wird vor dem Datenbankschema gebaut. Das korrigierst du im Review-Schritt, bevor ein einziger Code-Commit passiert.

Nach dem Review: `Execute den Plan mit TDD.` Claude baut dann Test-first, implementiert nach dem Rot-Grün-Refaktor-Zyklus und committed atomar — jeder Commit enthält genau eine abgeschlossene Einheit. Das ist der Unterschied zu 'hier ist der gesamte Feature-Code' in einem einzigen Commit, der schwer zu reviewen und bei Fehlern schwer zu reverten ist.

1. 1. Installiere den Superpowers-Skill global (Magic-Prompt oder direktes Ablegen der SKILL.md unter `~/.claude/skills/superpowers/SKILL.md`).
2. 2. Starte eine neue Claude Code Session im Projektverzeichnis.
3. 3. Tippe `/brainstorm [dein Feature in einem Satz]` — z.B. '/brainstorm Nutzer-Login mit Email/Passwort und JWT-Session'.
4. 4. Beantworte Claudes Rückfragen präzise. Vage Antworten produzieren vagen Plan.
5. 5. Lies die generierte `PLAN.md`. Prüfe: Stimmt die Reihenfolge? Sind alle Abhängigkeiten korrekt modelliert? Fehlt etwas?
6. 6. Gibt es Korrekturbedarf, antworte direkt mit dem Feedback: 'Ändere Schritt 3 — erst das DB-Schema, dann der API-Endpunkt.' Claude aktualisiert die PLAN.md.
7. 7. Wenn der Plan stimmt: 'Execute den Plan mit TDD.' Claude startet die Implementierung Test-first.
8. 8. Prüfe nach jedem Commit mit `git log --oneline`, ob die Commits atomar sind.

PROMPT

```
/brainstorm [Feature-Beschreibung in einem Satz]. Stelle mir alle
Rückfragen, die du brauchst, bevor du den Plan schreibst. Warte auf meine
Antworten.
```

PROMPT

```
Ich habe die PLAN.md gelesen. Ändere folgendes bevor du anfängst: [konkrete
Korrekturen]. Zeige mir danach die aktualisierte PLAN.md.
```

PROMPT

```
Execute den Plan mit TDD. Committe nach jeder abgeschlossenen Einheit
atomar. Nenne mir nach jedem Commit kurz was implementiert und getestet
wurde.
```

ÜBUNG

Nutze `/brainstorm` für ein echtes Feature in deinem aktuellen Projekt — auch ein kleines Feature ist geeignet (z.B. 'Passwort-Reset via Email'). Lass Claude die PLAN.md generieren, identifiziere einen Punkt der verbesserungswürdig ist und korrigiere ihn vor der Ausführung. Dokumentiere: Was hat der ursprüngliche Plan falsch angenommen, was hat deine Korrektur geändert?

▶ Lösung

3.3 Caveman: Token-Ausgabe um 65–75 % reduzieren

Claude Code hat eine Tendenz zu ausschweifenden Antworten: Einleitungen, Zusammenfassungen, ausführliche Erklärungen zu Code der eigentlich offensichtlich ist. In kurzen Sessions ist das tolerierbar. In langen Sessions — mehrere Stunden, viele Datei-Operationen, komplexe Features — wird es zum Problem: Der Kontext-Window füllt sich mit erklärendem Text, der nichts zur Lösung beiträgt. Wenn das Kontext-Window voll läuft, sinkt die Antwortqualität messbar.

Der Caveman-Skill löst das mit einem einzigen Prinzip: Maximale Information, minimale Tokens. Caveman-Modus bedeutet: kein einleitender Satz, kein Zusammenfassungs-Absatz, kein 'Great idea! Let me...'. Nur das, was direkt gefragt wurde — Code, Pfade, Kommandos, Fakten.

Die Token-Ersparnis ist real: 65–75% weniger Ausgabe-Tokens bedeutet, dass derselbe Kontext-Window 3–4× mehr genutzten Arbeitsfortschritt aufnimmt, bevor die Qualität absinkt. Für tageslanges Coding auf einer Feature-Branch ist das entscheidend.

Wann Caveman aktivieren: Ab dem Moment, wo du merkst, dass Claude viel erklärt und wenig liefert. Oder präventiv zu Beginn einer langen Session. Wann abschalten: Wenn du eine Erklärung für ein komplexes Konzept brauchst, das du noch nicht kennst — dann ist ausführlichere Ausgabe wertvoll. Caveman ist kein Dauermodus, sondern ein Effizienz-Werkzeug für Phasen hoher Arbeitsfrequenz.

1. 1. Installiere den Caveman-Skill unter `~/.claude/skills/caveman/SKILL.md`.
2. 2. Starte eine neue Claude Code Session.
3. 3. Stelle Claude eine normale Frage und notiere die Antwortlänge.
4. 4. Aktiviere Caveman: Tippe das Trigger-Wort des Skills (typischerweise 'caveman mode on' oder ein definiertes Kommando).
5. 5. Stelle dieselbe Frage erneut. Vergleiche Antwortlänge und Informationsdichte.
6. 6. Miss die Differenz: Wie viel Prozent kürzer ist die Caveman-Antwort? Enthält sie alle relevanten Informationen?
7. 7. Deaktiviere den Modus wieder mit dem entsprechenden Kommando wenn du eine ausführliche Erklärung brauchst.

PROMPT

```
Caveman mode on. Ab jetzt: kein einleitender Satz, keine Zusammenfassung,
keine Erklärung was du gleich tust. Direkte Antworten, minimal Tokens.
Bestätige mit einem Wort.
```

PROMPT

```
Caveman mode off. Antworte wieder normal ausführlich.
```

ÜBUNG

Führe einen direkten A/B-Vergleich durch: (1) Frage Claude ohne Caveman-Modus: 'Erkläre wie React useCallback funktioniert und wann man es nutzen sollte.' Zähle die Wörter der Antwort. (2) Aktiviere Caveman-Modus. Stelle dieselbe Frage. Zähle die Wörter. (3) Berechne die Reduktion in Prozent. (4) Prüfe: Fehlt in der Caveman-Antwort etwas Wichtiges, oder ist sie nur kürzer formuliert?

▶ Lösung

3.4 LLM Council: 5 Sub-Agenten für schwierige Entscheidungen

Eine einzelne KI-Antwort hat ein strukturelles Problem: Du bekommst eine Perspektive ohne zu wissen, wie sicher sie ist, was gegen sie spricht oder was sie übersieht. Der LLM Council löst das durch einen strukturierten Prozess: 5 Sub-Agenten mit unterschiedlichen Denkwinkeln analysieren dieselbe Frage, reviewen sich anonym gegenseitig und liefern ein Chairman-Fazit, das Konsens und Konflikte klar trennt.

Die 5 Rollen sind bewusst komplementär und teilweise gegensätzlich gewählt: Der Contrarian sucht aktiv den fatalen Fehler in deiner Idee. Der Expansionist findet das übersehene Upside. Der First-Principles-Thinker zerlegt alle Grundannahmen. Der Outsider bringt den Blick von jemandem ohne Expertenblindheit. Der Executor fragt nur: 'Was konkret tust du Montag?' Diese Spannung ist die Methode — nicht jeder wird zustimmen, und das ist gut so.

Der Council ist kein Allzweck-Werkzeug. Er lohnt sich ausschließlich bei Entscheidungen mit echten Tradeoffs und hohen Kosten bei Fehler — Pricing-Entscheidungen, Produkt-Pivot, Positionierung, Architektur-Wahl zwischen zwei gleichwertigen Optionen. Bei Faktenfragen ('Wie funktioniert useEffect?') oder Erstellungsaufgaben ('Schreib mir einen Test') ist der Council Overkill und liefert keinen Mehrwert.

Das Chairman-Fazit ist so strukturiert, dass du sofort handeln kannst: Was stimmen alle Berater zu — darauf kannst du mit hoher Konfidenz bauen. Wo widersprechen sie sich — das ist deine echte Unsicherheit, nicht der Scheinkomfort einer einzelnen KI-Antwort. Plus: einen konkreten nächsten Schritt, immer.

1. 1. Installiere den LLM Council Skill unter `~/.claude/skills/llm-council/SKILL.md`.
2. 2. Stelle eine Entscheidungsfrage mit echten Tradeoffs — z.B. 'council this: Soll ich mein SaaS-Produkt freemium oder trial-only launchen?'
3. 3. Warte bis alle 5 Sub-Agenten ihre Analyse abgeschlossen haben.
4. 4. Lies das anonymisierte Peer-Review (Antworten A–E, randomisiert zugeordnet).
5. 5. Lies das Chairman-Fazit: Konsens-Punkte, Konflikt-Punkte, Blind Spots, Empfehlung, nächster Schritt.
6. 6. Entscheide auf Basis des Fazits — nicht auf Basis der Antwort die dir am besten gefällt.

PROMPT

```
council this: [Kontext in 3–4 Sätzen]. Die konkrete Entscheidungsfrage: [Frage]. Meine zwei Optionen: (A) [Option A], (B) [Option B]. Liefere das vollständige Chairman-Fazit mit Konsens, Konflikten, Blind Spots und einem konkreten nächsten Schritt.
```

PROMPT

```
pressure-test this: [deine Idee oder Entscheidung]. Suche aktiv nach dem fatalen Fehler. Was übersehe ich?
```

ÜBUNG

Nutze den Council für eine echte Entscheidung aus deinem aktuellen Projekt oder Business. Formuliere die Frage mit 'council this:' gefolgt von: (1) dem Kontext in 3–4 Sätzen, (2) der konkreten Entscheidungsfrage, (3) den beiden Optionen die du abwägst. Lies das Chairman-Fazit und formuliere aus dem 'konkreten nächsten Schritt' eine Aufgabe, die du innerhalb von 24 Stunden umsetzen kannst.

▶ Lösung

4. Bonus-Skills für ship-ready Code

4.1 UI UX Pro Max: Automatisches Design-System

Generische KI-generierte UIs sehen aus wie generische KI-generierte UIs: gleiche Abstände, gleiche Farben, gleiche Typografie. Der Grund ist einfach — ohne explizite Design-Intelligenz optimiert Claude für 'funktioniert', nicht für 'sieht professionell aus'. UI UX Pro Max ist ein Skill, der diese Lücke schließt: Er bringt 50+ Styles, 161 Farbpaletten, 57 Font-Paarungen und 99 UX-Guidelines mit und wendet sie automatisch an, wenn du an UI-Tasks arbeitest.

Der praktische Unterschied zum normalen Claude-Workflow: Ohne den Skill schreibt Claude ein Button-Component in der Standardfarbe des Frameworks. Mit UI UX Pro Max analysiert er zuerst das bestehende visuelle Stil-Inventar des Projekts, wählt eine passende Farbpalette und Font-Paarung, und generiert dann konsistentes UI das zum Gesamt-Design passt — alles automatisch, ohne dass du 'mach es schöner' schreiben musst.

Der Skill triggert sich bei UI-bezogenen Aufgaben selbst — du musst ihn nicht manuell aktivieren. Sobald deine Anfrage UI-Elemente betrifft (Komponenten, Layouts, Formulare, Landing-Pages), erkennt Claude Code den Kontext und zieht die Design-System-Regeln des Skills automatisch hinzu.

Wenn du UI UX Pro Max auf ein bestehendes Projekt anwendest das noch kein Design-System hat: Lass Claude zuerst eine `design-system.md` generieren. Diese Datei beschreibt das visuelle Fundament — Primärfarben, Typografie-Hierarchie, Abstands-Skala, Komponent-Stile. Danach wird sie zur Referenz für alle folgenden UI-Generierungen im Projekt.

1. 1. Installiere UI UX Pro Max unter `~/.claude/skills/ui-ux-pro-max/SKILL.md`.
2. 2. Starte eine neue Session im Projektverzeichnis.
3. 3. Lass Claude zuerst das aktuelle visuelle Inventory des Projekts analysieren: 'Analysiere das bestehende Design-System des Projekts und liste Farben, Typografie und Komponenten-Stile.'
4. 4. Wenn noch kein Design-System existiert: 'Generiere eine design-system.md für dieses Projekt basierend auf UI UX Pro Max. Wähle Palette und Fonts die zu [Projekt-Kontext] passen.'
5. 5. Beauftrage einen UI-Task und beobachte ob Claude das Design-System automatisch anwendet: 'Erstelle eine Pricing-Page-Komponente.'
6. 6. Vergleiche die Ausgabe mit einer Ausgabe ohne UI UX Pro Max: Wirkt sie konsistenter und designbewusster?

PROMPT

```
Analysiere das bestehende visuelle Design-Inventar dieses Projekts: Farben, Typografie, Komponenten-Stile, Abstands-Skala. Erstelle danach eine design-system.md die das Fundament für alle UI-Generierungen in diesem Projekt definiert.
```

PROMPT

```
Erstelle [Komponenten-Beschreibung]. Wende dabei das Design-System aus
design-system.md konsistent an. Falls noch keine design-system.md existiert,
generiere sie zuerst.
```

ÜBUNG

Installiere UI UX Pro Max. Lass Claude eine `design-system.md` für dein Projekt oder ein fiktives SaaS-Produkt generieren. Beauftrage danach zwei UI-Komponenten: eine primäre Call-to-Action Sektion und eine Feature-Card. Prüfe ob beide Komponenten das Design-System konsistent anwenden — gleiche Primärfarbe, gleiche Font-Hierarchie, gleiche Abstands-Skala.

▶ Lösung

4.2 Context7 MCP: Aktuelle Framework-Doku in Echtzeit

Claude Codes Training hat ein Cutoff-Datum. Für stabile Kern-APIs (JavaScript-Grundlagen, HTTP-Protokoll) spielt das keine Rolle. Für sich schnell ändernde Frameworks — React Server Components, Next.js App Router, Supabase Row Level Security, Tailwind v4 — ist veraltetes Trainingswissen ein reales Risiko: Claude schlägt APIs vor, die in der aktuellen Version nicht mehr existieren, oder übersieht Breaking Changes.

Context7 ist ein MCP-Server, der Claude Code Echtzeit-Zugriff auf aktuelle Framework-Dokumentationen gibt. Statt aus dem Training zu antworten, ruft Claude die aktuelle Dokumentation ab und nutzt sie als Grundlage. Das Ergebnis: Kein 'veraltetes Wissen' mehr, kein manuelles 'Prüf bitte ob das noch aktuell ist'.

Die Einrichtung erfordert einen MCP-Server-Eintrag in der Claude Code Konfiguration — nicht über eine SKILL.md, sondern über die Konfigurationsdatei für externe Server. Der Unterschied zu Skills: Context7 ist kein Verhaltens-Skill, sondern ein Daten-Connector, der zur Laufzeit externe Informationen lädt.

Praktischer Test nach der Einrichtung: Frage Claude nach einem API-Feature das sich in den letzten 12 Monaten geändert hat — z.B. der Next.js `metadata`-Export für SEO (App Router, 2023 eingeführt) oder Tailwind CSS v4 Alpha-Syntax. Wenn Claude die aktuelle Version nennt und nicht die veraltete, funktioniert Context7.

1. 1. Installiere Context7 als MCP-Server: `npx -y @upstash/context7-mcp@latest` (oder die aktuell aktuelle Installationsmethode — prüfe die offizielle Context7-Dokumentation für die neueste Version).
2. 2. Füge den MCP-Server in die Claude Code Konfiguration ein (`~/.claude/config.json` oder die projektspezifische Konfigurationsdatei).
3. 3. Starte Claude Code neu, damit der neue MCP-Server geladen wird.
4. 4. Teste: Frage nach dem aktuellen Stand einer Framework-Funktion — z.B. 'Was ist die korrekte Syntax für Metadata in Next.js App Router?' Claude sollte die aktuelle Dokumentation zitieren, nicht veraltete API-Signaturen.
5. 5. Prüfe in Claudes Antwort ob er explizit auf die abgerufene Dokumentation hinweist — das bestätigt dass Context7 aktiv ist und nicht das Training genutzt wird.

PROMPT

```
Nutze Context7 um die aktuelle Dokumentation für [Framework + Feature]
abzurufen. Zeige mir die aktuelle Syntax und weise auf Breaking Changes hin,
falls vorhanden.
```

PROMPT

```
Schreibe folgenden Code unter Verwendung der aktuellen [Framework]-API –
nutze Context7 für die aktuelle Dokumentation: [Code-Anforderung]
```

ÜBUNG

Richte Context7 ein und führe danach drei Testrunden durch. Frage Claude nach: (1) der aktuellen Syntax für Supabase Row Level Security Policies, (2) dem aktuellen API für React `use()` Hook, (3) Tailwind CSS v4 Konfiguration. Prüfe für jede Antwort: Referenziert Claude die aktuelle Dokumentation? Gibt es Hinweise auf die Dokumentationsversion oder das Datum?

▶ Lösung

4.3 /security-review als Pflicht vor jedem Deploy

Sicherheitslücken entstehen in der Praxis selten durch ignorierte Best Practices — sie entstehen durch Zeitdruck, Ablenkung und 'das macht Claude schon richtig'. Ein Auth-Endpunkt ohne Rate-Limiting, ein API-Key in einer Umgebungsvariable die per `console.log` ausgegeben wird, ein SQL-ähnlicher Filter ohne Input-Validierung — diese Fehler passieren in echten Projekten täglich.

Das `/security-review`-Kommando in Claude Code ist kein separates Install-Paket, sondern ein eingebautes Capability-Kommando das den geänderten Code auf bekannte Sicherheitsprobleme scannt. Es filtert False Positives — nicht jede fehlende Sanitisierung ist tatsächlich ein Problem, und Claude ist in der Lage, das im Projektkontext einzuschätzen.

Die wichtigste Fähigkeit, die du beim Security-Review brauchst: den Unterschied zwischen echtem Fund und False Positive erkennen. Echte Funde haben immer: einen konkreten Angriffs-Vektor ('ein Angreifer kann X tun, wenn Y passiert'), einen messbaren Schaden ('Daten von anderen Nutzern lesen', 'Rate-Limit umgehen') und einen spezifischen Ort im Code. False Positives sind vage ('diese Funktion könnte theoretisch...') ohne konkreten Angriffs-Vektor.

Workflow-Integration: `/security-review` gehört in denselben Schritt wie `/ultrareview` — nach der Feature-Implementierung, vor dem Deploy. Nicht nach jedem Commit, aber mindestens einmal pro Feature-Branch bevor sie in main gemergt wird.

1. 1. Stelle sicher, dass du Code-Änderungen in einem Feature-Branch gemacht hast.
2. 2. Tippe `/security-review` im Claude Code Chat.
3. 3. Spezifiziere optional den Scope: '/security-review der Auth-Endpunkte in src/api/auth/'.
4. 4. Lies den Output: Jeder Fund sollte einen Schweregrad haben (Critical/High/Medium/Low), einen konkreten Angriffs-Vektor und den genauen Ort im Code.
5. 5. Klassifiziere jeden Fund als echt oder False Positive — prüfe ob ein konkreter Angriffs-Vektor beschrieben wird.
6. 6. Für echte Funde: 'Zeig mir den konkreten Fix für [Fund]' und implementiere ihn direkt.
7. 7. Für False Positives: Notiere warum es kein Problem ist, damit du beim nächsten Review schneller entscheiden kannst.

PROMPT

```
/security-review — Analysiere alle Änderungen seit dem letzten Commit.
Kategorisiere jeden Fund nach Schweregrad (Critical/High/Medium/Low), nenne
den konkreten Angriffs-Vektor und den genauen Ort im Code.
```

PROMPT

```
Zeig mir den konkreten Fix für den folgenden Security-Fund und erkläre warum der Fix die Lücke schließt: [Fund-Beschreibung aus dem Review]
```

ÜBUNG

Füge absichtlich zwei klassische Sicherheitsprobleme in eine Test-Datei ein: (1) einen hardcodierten API-Key als String-Literal, (2) eine Funktion die User-Input direkt in eine URL concateniert ohne Encoding. Führe dann `/security-review` aus und prüfe: Findet Claude beide Probleme? Beschreibt er konkrete Angriffs-Vektoren? Lass Claude danach beide Probleme fixen und prüfe ob die Fixes korrekt sind.

▶ Lösung

5. Praxis-Workflow: Alles zusammensetzen

5.1 Alle 8 Skills mit einem einzigen Prompt installieren

Jeden Skill einzeln zu installieren — Dokumentation suchen, aktuellen Install-Command finden, ausführen, verifizieren — dauert pro Skill 5–15 Minuten. Für 8 Skills summiert sich das auf über eine Stunde Overhead, bevor du überhaupt mit dem eigentlichen Setup angefangen hast.

Der Massen-Install-Prompt delegiert diesen Prozess komplett an Claude Code: Du nennst die 8 Skills in einem einzigen Prompt, Claude recherchiert für jeden Skill selbstständig den aktuellen offiziellen Install-Command (via Context7 oder Websuche), führt ihn aus und verifiziert die Installation. Das dauert typischerweise 3–5 Minuten für alle 8 Skills.

Der Prozess ist nicht unfehlbar: Manchmal schlägt eine Einzel-Installation fehl — falscher Pfad, veralteter Command, Netzwerkfehler. Nach dem Massen-Install prüfst du daher systematisch welche Skills korrekt installiert wurden und installierst fehlgeschlagene manuell nach. Das Verhältnis ist in der Praxis günstig: Meist sind 6–7 von 8 beim ersten Durchlauf erfolgreich.

Nach der Installation gehört eine Verifikationsrunde dazu: Für jeden installierten Skill einmal das Trigger-Kommando testen. Wenn Claude korrekt reagiert — Installation erfolgreich. Wenn nicht — SKILL.md-Datei prüfen: Existiert sie am richtigen Pfad? Ist der Dateiname korrekt (`SKILL.md`, nicht `skill.md`)? War die Sitzung danach neu gestartet?

1. 1. Stelle sicher, dass du Claude Code mit Internet-Zugriff nutzt (damit Claude die aktuellen Install-Commands recherchieren kann).
2. 2. Schicke den Massen-Install-Prompt (siehe Prompts-Abschnitt) — liste alle 8 Skills darin: Superpowers, Skill Creator, Caveman, LLM Council, UI UX Pro Max, Security Review, Frontend Design, Context7 MCP.
3. 3. Warte bis Claude alle Installationen abgeschlossen hat und einen Status-Report liefert.
4. 4. Prüfe den Bericht: Welche Skills wurden erfolgreich installiert, welche sind fehlgeschlagen?
5. 5. Verifiziere manuell: `ls ~/.claude/skills/` — alle installierten Skill-Ordner sollten sichtbar sein.
6. 6. Starte eine neue Claude Code Session.
7. 7. Teste jeden Skill mit seinem Trigger-Kommando. Notiere welche nicht funktionieren.
8. 8. Installiere fehlgeschlagene Skills manuell nach — SKILL.md-Inhalt direkt mit dem install-Prompt anlegen.

PROMPT

```
Installiere folgende 8 Claude Code Skills und den Context7 MCP-Server.
Recherchiere für jeden Skill den aktuellen offiziellen Install-Command,
führe ihn aus und verifiziere die Installation. Liefere am Ende einen
Status-Report pro Skill:

1. Superpowers (agentic Framework mit /brainstorm, TDD, Git-Worktrees)
2. Skill Creator (offizieller Anthropic-Skill zum Bauen eigener Skills)
3. Caveman (Token-Reduktion um 65–75%)
4. LLM Council (5 Sub-Agenten für Entscheidungen)
5. UI UX Pro Max (Design-System mit 161 Farbpaletten, 57 Font-Paarungen)
6. Security Review (eingebaut in Claude Code – kein separates Install, nur
Dokumentation eintragen)
7. Frontend Design (offizieller Anthropic-Skill für Designphilosophie)
8. Context7 MCP (Echtzeit-Framework-Dokumentation)

Installationspfad für globale Skills: ~/.claude/skills/<skill-name>/SKILL.md
```

PROMPT

```
Prüfe welche Claude Code Skills korrekt installiert sind: Lese alle Dateien
unter ~/.claude/skills/ und teste jedes Trigger-Kommando. Liste Status pro
Skill.
```

ÜBUNG

Führe den Massen-Install-Prompt aus. Erstelle danach eine Tabelle mit 3 Spalten: Skill-Name, Installationsstatus (✓ / ✗), Trigger-Test-Ergebnis (funktioniert / nicht). Für jeden Skill der fehlgeschlagen ist: Identifiziere den Fehlergrund (falscher Pfad, falscher Dateiname, Session nicht neu gestartet) und behebe ihn.

▶ Lösung

5.2 Feature-Workflow: Planen → Bauen → Reviewen → Shippen

Alle installierten Skills und Konfigurationen entfalten ihren vollen Wert erst, wenn sie als zusammenhängender Workflow eingesetzt werden — nicht als isolierte Einzelwerkzeuge. Der vollständige Feature-Workflow sieht so aus: Brainstorming und Planung mit Superpowers, Implementierung mit TDD, Design mit UI UX Pro Max, Review mit `/ultrareview`, Sicherheits-Check mit `/security-review`, und bei kritischen Entscheidungen zwischendurch `council this:`.

Der Council hat im Workflow einen spezifischen Platz: nach dem Brainstorming, wenn du zwischen zwei gleichwertigen Architektur-Optionen wählen musst. Nicht nach jedem Commit, nicht bei Implementierungsdetails. Das Signal für 'jetzt Council': Du hast zwei Optionen und keine offensichtliche Überlegenheit einer der beiden. Dann ist die strukturierte Mehr-Perspektiven-Analyse sinnvoll. Bei klaren technischen Fragen ('Welche Library für JWT?') reicht eine normale Claude-Antwort.

Das `/ultrareview` vor dem Ship ist kein optionaler Luxus, sondern der Moment, wo Claude mit dem stärksten Modell auf Effort Max den gesamten Code der Feature-Branch unter die Lupe nimmt — Korrektheit, Edge Cases, Performance, Lesbarkeit. Kombiniert mit `/security-review` ist das die Qualitäts-Gate, die verhindert, dass Fehler in Production landen.

Langfristig: Die Qualität jeder Ausgabe hängt direkt von der Qualität der CLAUDE.md ab. Nach den ersten zwei bis drei Feature-Zyklen mit diesem Workflow wirst du Muster erkennen — bestimmte Typen von Fehlern, die Claude wiederholt macht, bestimmte Anforderungen die er misversteht. Diese Muster kommen direkt in die CLAUDE.md als neue Regeln. Die CLAUDE.md ist kein einmaliges Dokument, sondern ein lebendes Briefing das mit jedem Feature besser wird.

1. 1. Feature-Start: `/brainstorm [Feature-Beschreibung]`. Beantworte Claudes Rückfragen vollständig.
2. 2. Plan-Review: Lies die generierte PLAN.md. Gibt es Architektur-Konflikte? Falls ja: `council this:` mit den zwei Optionen.
3. 3. Implementierung: 'Execute den Plan mit TDD.' Verfolge die Commits: Sind sie atomar? Gibt es Tests?
4. 4. UI-Tasks (falls vorhanden): Claude wendet UI UX Pro Max automatisch an — prüfe Konsistenz mit dem Design-System.
5. 5. Feature-Complete: `/ultrareview` ausführen. Alle gefundenen HIGH und CRITICAL Punkte fixen.
6. 6. Security-Gate: `/security-review` ausführen. Echte Funde sofort fixen, False Positives dokumentieren.
7. 7. Deploy: Deploy-Kommando aus der CLAUDE.md ausführen.

8. 8. Post-Deploy: Was hat der Workflow gut gemacht, was wurde trotzdem problematisch? → CLAUDE.md aktualisieren.

PROMPT

```
/brainstorm [Feature-Beschreibung in einem Satz]. Stell mir alle Rückfragen die du brauchst. Wenn du alle Infos hast, schreib die PLAN.md und zeig sie mir zum Review bevor du anfängst.
```

PROMPT

```
/ultrareview — Analysiere alle Änderungen in diesem Feature-Branch auf: Korrektheit der Logik, fehlende Edge-Case-Behandlung, Performance-Probleme, Lesbarkeit. Nutze das stärkste Modell mit maximalem Effort. Kategorisiere jeden Fund nach CRITICAL/HIGH/MEDIUM/LOW.
```

PROMPT

```
council this: Ich stehe vor folgender Architektur-Entscheidung in [Feature-Name]: [Kontext]. Option A: [Beschreibung]. Option B: [Beschreibung]. Mein Kontext: [Stack, Teamgröße, erwartete Last]. Was ist die bessere Wahl und warum?
```

ÜBUNG

Führe den vollständigen Feature-Workflow für ein echtes Mini-Feature durch — z.B. 'Kontaktformular mit Email-Versand' oder 'Dark-Mode-Toggle'. Nutze alle Schritte: /brainstorm → PLAN.md → TDD-Execute → /ultrareview → /security-review → Deploy. Dokumentiere nach dem Deploy in deiner CLAUDE.md: Welche eine Regel hättest du schon vorher in der CLAUDE.md haben wollen, die Claude dann automatisch beachtet hätte?

▶ Lösung