31 Dinge, die deine KI-App sofort schneller machen

Mein Performance-Prompt für Claude Code: erst messen, dann 31 Maßnahmen von Skeleton-Ladescreens bis Datenbank-Index, dann nachmessen.



Berkan Ceyhan · Stand November 2026

AUF EINEN BLICK

- **31 Maßnahmen** für Oberfläche, Browser, Daten und Server
- **Erst messen**, dann ändern, dann nachmessen
- **Jede Änderung** ein eigener Commit
- **Claude fragt**, bevor etwas Geld kostet

Kopier den Prompt in Claude Code

Öffne dein Projekt in Claude Code und schick den Prompt. Claude misst zuerst, setzt die Punkte um, die bei dir etwas bringen, und zeigt dir am Ende eine Tabelle mit Vorher und Nachher.

App schneller machen

```
Claude, ich will nicht, dass meine gevibecodete App langsam ist und Nutzer verliert. Geh
mein Projekt durch und mach es schneller. Arbeite in drei Phasen: messen, umsetzen,
nachmessen.

PHASE 0: PROJEKT VERSTEHEN
Lies die Projektdateien und schreib mir in fünf Zeilen auf: Framework, Sprache, Datenbank,
Hoster, wo Bilder und Schriften liegen. Frag mich nur, wenn du etwas nicht herausfinden
kannst.

PHASE 1: MESSEN (bevor du etwas änderst)
- Miss die Startseite und die zwei wichtigsten Seiten mit Lighthouse im mobilen Modus.
Notier LCP, INP, CLS, Gesamtgröße und Zahl der Anfragen.
- Starte den Bundle-Analyzer deines Build-Tools und notier die fünf größten Pakete.
- Such die langsamsten API-Routen und Datenbankabfragen. Lass sie loggen oder nutze
EXPLAIN. Notier die Top 5 mit Dauer.
Gib mir diese Zahlen als Tabelle. Sie sind der Vorher-Stand.

PHASE 2: UMSETZEN
Geh die 31 Punkte unten einzeln durch. Prüf bei jedem, ob er zu meinem Projekt passt. Setz
ihn um, wenn er einen messbaren Effekt hat. Passt er nicht, überspring ihn und schreib in
einem Satz, warum. Arbeite nach Wirkung: zuerst die Punkte, die deine Messung als größte
```

```
Bremse zeigt. Mach eine Änderung nach der anderen, lass nach jeder die Tests und den Build laufen und halt die Funktionen so, wie sie sind.

Frag mich vorher, bevor du etwas einführst, das Geld kostet oder Betrieb bedeutet: Redis, ein CDN-Vertrag, ein Load Balancer, eine Job-Queue als eigener Dienst, ein Monitoring-Dienst. Beschreib mir dann Nutzen, Kosten und die einfachste Variante.

A) WAS DER NUTZER SIEHT
1. Skeleton-Ladescreens: Ersetz Spinner und leere Flächen durch Platzhalter in der Form des späteren Inhalts, für jede Liste, Karte und Detailseite, die Daten nachlädt. Beim Einblenden darf nichts springen.
2. Optimistische Updates: Bei Aktionen wie Liken, Abhaken oder Speichern aktualisier die Oberfläche sofort und schick die Anfrage danach. Schlägt sie fehl, roll zurück und zeig eine verständliche Meldung.
3. KI-Antworten streamen: Zeig Antworten von Sprachmodellen Token für Token (Server-Sent Events oder Stream), statt auf die ganze Antwort zu warten. Ein Abbrechen-Knopf gehört dazu.
4. Debounce bei Suche und Autocomplete: Feuer die Anfrage frühestens 250 bis 300 ms nach dem letzten Tastendruck und erst ab 2 Zeichen. Brich die vorherige Anfrage ab, wenn eine neue startet.
5. Lange Listen virtualisieren: Ab etwa 100 Einträgen rendert nur, was im Sichtbereich liegt (zum Beispiel mit TanStack Virtual oder react-window).
6. Unnötige Re-Renders vermeiden: Finde Komponenten, die ohne Grund neu rendern (React Profiler). Zieh State dichter an die Stelle, die ihn braucht, teil große Contexts auf und setz memo, useMemo und useCallback nur dort ein, wo die Messung es zeigt.

B) WAS DER BROWSER LÄDT
7. Bilder als WebP oder AVIF: Konvertier alle Fotos und Grafiken, liefere passende Größen (srcset und sizes) und setz feste Breite und Höhe. Logos und Icons als SVG. Melde mir jede Bilddatei über 200 KB.
8. Bilder unterhalb des sichtbaren Bereichs später laden: loading="lazy" für alles unterhalb der ersten Bildschirmhöhe. Das größte Bild im Sichtbereich lädt sofort (fetchpriority="high"), nie lazy.
9. JavaScript pro Seite aufteilen: Nutz Code-Splitting und dynamische Imports für schwere Teile (Editor, Charts, Karten, Dialoge). Eine Seite lädt nur den Code, den sie braucht.
10. Ungenutzte Abhängigkeiten entfernen: Such mit depcheck oder knip nach Paketen, die niemand importiert. Ersetz riesige Pakete durch kleinere, wenn du nur eine Funktion brauchst. Zeig mir die Bundle-Größe vorher und nachher.
11. Drittanbieter-Skripte erst nach der Seite laden: Analytics, Chat-Widgets, Pixel und Tag Manager laden mit defer oder async, nach der ersten Interaktion oder nach Einwilligung. Prüf, ob jedes Skript noch gebraucht wird.
12. Schriften selbst hosten und begrenzen: woff2 vom eigenen Server statt von Drittanbietern, nur die Schnitte und Zeichensätze, die die Seite nutzt, font-display: swap, die wichtigste Schrift per preload.
13. Render Seiten auf dem Server oder beim Build: Seiten, die für alle gleich sind (Startseite, Preise, Blog), werden beim Build vorgerendert. Dynamische Seiten rendern auf dem Server. Eine Landingpage, die erst im Browser aus JavaScript entsteht, baust du um.

C) WIE DATEN UNTERWEGS SIND
14. Doppelte API-Aufrufe verhindern: Finde Stellen, an denen dieselbe Anfrage mehrfach feuert (useEffect ohne saubere Abhängigkeiten, mehrere Komponenten holen dieselben Daten). Nutz eine Datenschicht mit Cache und Deduplizierung (TanStack Query, SWR oder das, was dein Framework mitbringt).
15. Unabhängige Anfragen parallel ausführen: Wo zwei Anfragen nicht voneinander abhängen, starte sie gleichzeitig (Promise.all) statt nacheinander.
```

16. API-Ergebnisse paginieren: Kein Endpunkt liefert alles auf einmal. Nutz Cursor-Pagination mit 20 bis 50 Einträgen pro Seite und „Mehr laden“ oder unendliches Scrollen.
17. Timeouts für externe API-Aufrufe: Setz bei jedem Aufruf an Dritte ein Timeout (5 bis 10 Sekunden) und zeig bei Ablauf eine Meldung statt eines endlosen Spinners. Wiederhol nur Aufrufe, die man gefahrlos wiederholen kann, mit wachsendem Abstand.
18. Kompression einschalten: Brotli oder Gzip für HTML, CSS, JavaScript und JSON. Prüf mit curl -I und dem Header Accept-Encoding, ob sie ankommt.
19. Statische Dateien über ein CDN ausliefern: Bilder, Schriften, JS und CSS kommen vom CDN, nicht vom App-Server.
20. Browser-Caching: Dateien mit Hash im Namen bekommen Cache-Control: public, max-age=31536000, immutable. HTML bekommt kurze Zeiten oder Revalidierung. Prüf die Header mit dem Browser.

D) WAS AUF DEM SERVER PASSIERT
21. N+1-Abfragen beheben: Such Schleifen, die pro Eintrag eine Abfrage absetzen. Ersetz sie durch einen Join, Eager Loading oder einen Dataloader. Log die Zahl der Abfragen pro Request vorher und nachher.
22. Datenbank-Indizes für häufige Abfragen: Leg Indizes für Spalten an, nach denen du filterst, joinst oder sortierst. Prüf mit EXPLAIN ANALYZE, dass sie genutzt werden, und leg keine Indizes auf jede Spalte.
23. Nur die benötigten Spalten abfragen: Kein SELECT *. Hol in Listen nur, was die Liste zeigt, und lass große Textfelder weg.
24. Connection Pooling: Nutz einen Pool für die Datenbankverbindungen (zum Beispiel den Pooler deines Anbieters oder PgBouncer), besonders bei Serverless-Funktionen. Stell die Pool-Größe passend zum Datenbanklimit ein.
25. Häufig genutzte Daten cachen: Cache Daten, die oft gelesen und selten geändert werden (Redis oder ein Cache deines Hosters). Gib jedem Eintrag eine Lebensdauer und räum ihn bei Änderungen auf. Nutzerspezifische Daten nur mit Nutzer im Schlüssel.
26. Langsame Aufgaben in eine Warteschlange: E-Mails, PDFs, Bildverarbeitung und lange KI-Aufrufe laufen im Hintergrund. Die Anfrage antwortet sofort mit einem Status, und die App fragt das Ergebnis ab oder bekommt Bescheid.

E) WO DER SERVER STEHT
27. Server nah an der Datenbank: App und Datenbank stehen in derselben Region. Prüf die Regionen bei deinem Hoster und schlag eine Änderung vor, wenn sie auseinanderliegen.
28. Last auf mehrere Instanzen verteilen: Nur wenn die Messung zeigt, dass eine Instanz nicht reicht. Mach die App zustandslos (Sitzungen nicht im Arbeitsspeicher), leg Health-Checks an und beschreib die einfachste Variante bei meinem Hoster.

F) MESSEN UND TESTEN
29. Langsame Backend-Anfragen verfolgen: Bau OpenTelemetry ein (Traces für Datenbank und externe Aufrufe) und zeig mir die langsamsten Routen mit p95.
30. Echte Ladezeiten messen: Richte Performance-Monitoring mit echten Nutzerdaten ein (Web Vitals: LCP, INP, CLS, zum Beispiel über deinen Hoster oder ein Monitoring-Tool). Lab-Werte allein reichen nicht.
31. Auf langsamen Handys und Netzen testen: Teste mit gedrosseltem Netz (Slow 4G) und gedrosselter CPU (4x langsamer) in den Chrome-Entwicklertools und, wenn möglich, auf einem echten Mittelklasse-Handy.

PHASE 3: NACHMESSEN
Miss genau wie in Phase 1 und gib mir eine Tabelle: Messwert, vorher, nachher. Danach drei Listen:
1. Umgesetzt (mit Datei und einem Satz zur Wirkung)
2. Übersprungen (mit Grund)
3. Offen, weil es Geld oder Betrieb kostet (mit Nutzen und einfachster Variante)

```
REGELN
- Ändere keine Funktionen und kein Aussehen, außer es ist für die Geschwindigkeit nötig.
Sag mir dann, was.
- Erfinde keine Messwerte. Konntest du etwas nicht messen, schreib das hin.
- Commit nach jedem Punkt einzeln, mit einem klaren Satz, damit ich jede Änderung
zurücknehmen kann.
- Leg keine Schlüssel, Passwörter oder Tokens in den Code.
```

Du willst wissen, wo KI in deinem Betrieb Zeit spart? Buche einen Call mit mir und wir finden es gemeinsam raus.

Call buchen

Nicht alles passt zu jeder App

MEIN TIPP

Redis, Warteschlange und Load Balancer sind Betrieb

Sie kosten Geld und müssen laufen. Der Prompt lässt Claude dich vorher fragen. Bei einer kleinen App bringen Bilder, Indizes und weniger JavaScript meist mehr als ein neuer Dienst.



Automatisierungs-Call für Agenturen, Coaches und Dienstleister

In 30 Minuten schauen wir gemeinsam, wo KI-Agenten und automatisierte Prozesse in deinem Betrieb zuerst Zeit und Geld sparen. Kostenlos.

FÜR WEN IST DAS?

- ✓ Agenturen, Coaches und Dienstleister
- ✓ ab 10.000 € Monatsumsatz
- ✓ ein klarer Flaschenhals im Tagesgeschäft
- ✓ du willst KI im Betrieb einsetzen

Call buchen

api.leadconnectorhq.com/widget/booking/mXsszCow3Sxqm4QEnOxK

Scannen & Termin wählen