Prompts, mit denen ein LLM Ihre Software baut.

BY MARC GLOOR · SEITE 1 VON 2 — NEUN GRUNDREGELN & DIE REFERENZVORLAGE

Ob ein Modell aus einer Idee funktionierenden Code macht, hängt fast vollständig von Klarheit und Struktur der Aufforderung ab. Neuere Arbeiten zu Requirements Engineering und Code-Generierung führen zur gleichen Erkenntnis: Einzeilige Prompts ergeben einzeilige Software. Es folgt eine verdichtete Sammlung von Praktiken — aus begutachteten Studien und praxiserprobten Leitfäden — um Anforderungen in Prompts zu überführen, die ein LLM tatsächlich umsetzt.

NEUN GRUNDREGELN

Von Anforderung zu Code, vom Terminal bis zum Browser — die Praktiken, die Ergebnisse von Prototyp- auf Produktionsniveau heben.

1 Rolle und Kontext zuerst setzen

Beginnen Sie mit einer Rolle — „Agiere als erfahrener Backend-Ingenieur in einer service-orientierten Architektur." — und einem Absatz, der das Projekt, seine Rahmenbedingungen und die einzuhaltenden Standards benennt. Eine Rolle bringt implizit Prioritäten (Sicherheit, Skalierbarkeit, idiomatischen Stil) ein, die das Modell sonst raten müsste.

2 Bestimmte, spezifische Sprache verwenden

Ersetzen Sie „sollte" und „kann" durch „muss" und „ist erforderlich". Vage Verben provozieren vagen Code. Benennen Sie Dateien, Klassen, Methoden und bestehende Muster, denen der neue Code folgen muss, statt sie abstrakt zu beschreiben.

> ✗ „Schreibe eine Funktion zur Verarbeitung von Benutzerdaten."
> ✓ „Füge *transformUser* in *UserProcessor.js* hinzu, analog zu *transformPaymentData*."

3 Technische Umgebung festlegen

Geben Sie Sprache und exakte Versionen an, das Framework, erlaubte (und verbotene) Bibliotheken, die Laufzeit, die Zielplattform und die Code-Konventionen. Andernfalls greift das Modell auf Mehrheitsmuster aus den Trainingsdaten zurück, die oft im Widerspruch zu Ihrer tatsächlichen Codebasis stehen.

4 Anforderungen als nummerierte Liste

Zerlegen Sie, was die Software leisten muss, in einzelne, nummerierte, testbare Punkte. Nummerierte Listen werden zuverlässiger verarbeitet als Fließtext und zwingen Sie, Mehrdeutigkeiten zu klären, bevor das Modell es tut. Trennen Sie funktionale, nicht-funktionale und Schnittstellenanforderungen.

5 Eingaben, Ausgaben und Sonderfälle definieren

Benennen Sie das E/A-Format, Vor- und Nachbedingungen, Fehlerfälle und mindestens zwei oder drei konkrete Beispiele — darunter einen Grenzfall. Few-Shot-Beispiele schlagen abstrakte Spezifikationen zuverlässig, besonders bei Aufgaben mit sensiblen Datenformen.

6 Schrittweises Prompting anwenden

Für alles, was über eine einzelne Funktion hinausgeht: Keinen Code in einem Wurf anfordern. Führen Sie das Modell schrittweise: Anforderungen schärfen, Entwurf ableiten, Tests schreiben, Code schreiben. Die Ausgabe jeder Stufe wird zur Eingabe der nächsten. Das gleicht dem Wasserfall und bietet auf jeder Stufe eine Prüffläche.

7 Verifikation in den Prompt einbauen

Lassen Sie das Modell Tests vor — oder parallel zur — Implementierung schreiben und das Ergebnis gegen Ihre Akzeptanzkriterien selbst prüfen, inklusive Auflistung der getroffenen Annahmen und offener Sonderfälle. Tests sind die ausführbare Form Ihrer Anforderungen; ohne sie lässt sich nicht feststellen, ob der Code sie erfüllt.

8 Sorgfalt am Risiko ausrichten

Ein Wegwerf-Skript verträgt einen „faulen" Prompt; ein Zahlungsfluss nicht. Kalibrieren Sie den Aufwand am Einsatz: Prototypen mit geringem Risiko erlauben lockeres, explorativ es Prompting; kritische Features verlangen einen vollständigen Spec-vor-Code-Durchlauf mit dokumentierten Anforderungen, Entwurf und Tests.

9 Oberflächenparadigma festlegen — TUI oder GUI

Benennen Sie die Oberfläche ausdrücklich und legen Sie ihre Konventionen fest; sonst wählt das Modell Standards, die selten zu Ihrer Plattform passen.

> **TUI** Bibliothek (textual / curses / prompt_toolkit); Tastenschema (vi / emacs / benannt); Mindestbreite; --json für Pipes; monochrom zuerst.
> **GUI** Design-System; Zustände (leer, Laden, Fehler, Erfolg); Barrierefreiheit (WCAG AA); Kontrast; volle Tastaturunterstützung; Microcopy im Prompt.

REFERENZVORLAGE · WIEDERVERWENDBARES PROMPT-GERÜST

Die acht-Block-Struktur, die in den meisten Projekten trägt.

```
ROLE          Du bist ein erfahrener {Sprache}-Ingenieur. Priorisiere Klarheit, Testbarkeit, Sicherheit.
CONTEXT       Projekt: {Name, Einzeiler}. Codebasis-Stil: {Muster, Dateien als Vorbild}.
STACK         Sprache {X.Y}. Framework {X.Y}. Erlaubte Bibliotheken: {…}. Verboten: {…}.
UI / UX        Paradigma: {TUI | GUI}.
              TUI — Bibliothek {textual/curses/prompt_toolkit}; Tasten {vi/emacs/benannt}; Breite ≥ {N} Spalten; --json für Pipes; monochrom zuerst.
              GUI — Design-System {…}; Zustände leer/Laden/Fehler/Erfolg; Barrierefreiheit {WCAG AA}; Kontrast ≥ 4,5:1; Microcopy hier.
REQUIREMENTS  1. {Pflicht} 2. {Pflicht} 3. {verboten} … (nummeriert, bestimmt, testbar).
I/O & EDGES   Eingaben: {Typen, Formate}. Ausgaben: {Typen, Formate}. Sonderfälle: {…}. Beispiele: {…}.
PROCESS       Schritt 1 — Funktionssignatur + Docstring vorschlagen; auf meine Freigabe warten.
              Schritt 2 — fehlschlagende Tests für die obigen Anforderungen schreiben.
              Schritt 3 — implementieren; dann selbst gegen die Anforderungen prüfen und Annahmen auflisten.
OUTPUT        Vollständige Dateipfade vor jedem Block. Produktionsreif, ohne Auslassungen, ohne Platzhalter.
```

VERMEIDEN **Einzeilige Anforderungen.** **Was und Wie vermischen** in einem Satz. **Den Stack vom Modell** erfinden lassen. **Den ersten Entwurf** als endgültig nehmen. **Keine Tests, keine Abnahme.**

7×

Mehr kleine, aber folgenreiche Korrekturen tauchen auf, wenn ein Prompt aufgeteilt wird in **Anforderungen → Entwurf → Tests → Code** statt in einem Wurf. Schrittweises Prompting fängt sie ab, *bevor* sie die Codebasis erreichen.

„Wir raten von Methoden ab, die für LLM-basierte Softwareentwicklung auf einzeilige Anforderungen setzen. Stattdessen betonen wir die Bedeutung, Zeit und Mühe in qualitativ hochwertige, detaillierte Anforderungen zu investieren."

ARORA ET AL. · REQUIREMENTS ARE ALL YOU NEED (2024)

QUELLEN · White et al., *ChatGPT Prompt Patterns for Software Engineering* (arXiv 2303.07839) · Arora et al., *Requirements are All You Need: From Requirements to Code with LLMs* (arXiv 2406.10101) · *Prompt Engineering for Requirements Engineering: A Literature Review & Roadmap* (arXiv 2507.07682) · Ullrich, Koch & Vogelsang, *From Requirements to Code: Developer Practices in LLM-Assisted SE* (arXiv 2507.07548) · *Guidelines to Prompt LLMs for Code Generation* (arXiv 2601.13118) · Mosofsky, *Spec-Then-Code* (GitHub) · Palantir, *Prompt-Engineering Best Practices*.

BEISPIEL · EIN VOLLSTÄNDIGER BAU, VON ANFANG BIS ENDE

Einen Texteditor bauen. Vorlage kopieren. Klammern füllen.

Zwei Blöcke unten. **Der erste** *ist das nackte Acht-Block-Gerüst von Seite eins — fügen Sie es in das Modell Ihrer Wahl ein und ersetzen Sie jede* `{Klammer}` *durch die Spezifika Ihres Projekts.* **Der zweite** *ist dasselbe Gerüst, bereits ausgefüllt für ein kleines, aber realistisches Bauziel: einen einfenster-, browserbasierten Klartext-Editor (eine GUI). Die TUI-Variante ist im UI / UX-Block der nackten Vorlage beschrieben. Verwenden Sie eine als Musterlösung oder kopieren und anpassen Sie sie.*

BLOCK 1 Die nackte Vorlage — diese kopieren

JEDE {KLAMMER} ERSETZEN

ROLE	Du bist ein erfahrener {Sprache}-Ingenieur. Priorisiere Klarheit, Testbarkeit, {Prioritäten}.
CONTEXT	Projekt: {Name + Einzeiler}. Codebasis-Stil: {Muster, Dateien als Vorbild}.
STACK	Sprache {X.Y}. Framework {X.Y}. Erlaubte Bibliotheken: {…}. Verboten: {…}.
UI / UX	Paradigma: {TUI | GUI}. TUI — Bibliothek {textual/curses/prompt_toolkit}; Tasten {vi/emacs/benannt}; Breite ≥ {N} Spalten; --json für Pipes; monochrom zuerst. GUI — Design-System {…}; Zustände leer/Laden/Fehler/Erfolg; Barrierefreiheit {WCAG AA}; Kontrast ≥ 4.5:1; Microcopy hier.
REQUIREMENTS	1. {Pflicht} 2. {Pflicht} 3. {verboten} … (nummeriert, bestimmt, testbar).
I/O & EDGES	Eingaben: {Typen, Formate}. Ausgaben: {Typen, Formate}. Sonderfälle: {…}. Beispiele: {…}.
PROCESS	Schritt 1 — Dateibaum + Typen vorschlagen; auf Freigabe warten. Schritt 2 — fehlschlagende Tests für die Anforderungen schreiben. Schritt 3 — implementieren; selbst prüfen; Annahmen auflisten.
OUTPUT	Vollständige Dateipfade vor jedem Block. Produktionsreif, ohne Auslassungen, ohne Platzhalter.

BLOCK 2 Beispiel — ein browserbasierter Texteditor

KOPIEREN & ANPASSEN

ROLE	Du bist ein erfahrener TypeScript-/React-Ingenieur. Priorisiere Klarheit, Testbarkeit, Barrierefreiheit. Bevorzuge einfachen, idiomatischen Code vor klugen Abstraktionen.
CONTEXT	Projekt: **NotePad** — ein einfenster-Klartext-Editor im Browser. Stil: kleine, fokussierte Komponenten, reine Funktionen für Textoperationen, kein globaler Zustand. Folge den Mustern aus `src/components/Toolbar.tsx`.
STACK	TypeScript 5.4. React 18.3. Vite 5. CSS Modules. Erlaubte Bibliotheken: nur `react`, `react-dom`. Verboten: Redux, MobX, jQuery, jede Rich-Text-Bibliothek (Quill, Slate, CKEditor). Der Editor MUSS ein kontrolliertes `<textarea>` sein, kein `contenteditable`.
UI / UX	Paradigma: **GUI** (einfenster Web-App). Gestaltung: minimal — System-Schriften, neutrale Palette, keine Verzierung außer der Werkzeugleiste. Zustände: leeres Dokument → mittiger Platzhalter „Tippen Sie oder öffnen Sie eine Datei"; Laden → Spinner in der Werkzeugleiste; Fehler → oberes Toast, schließbar; Erfolg (gespeichert) → 2 s lange „Gespeichert"-Plakette. Barrierefreiheit (WCAG AA): volle Tastaturbedienung, keine ausschließlich mit Maus erreichbaren Aktionen; `<textarea>` beschriftet; Werkzeugleisten-Buttons mit `aria-label`; sichtbarer Fokusring; Kontrast ≥ 4,5:1; respektiert `prefers-reduced-motion`. Microcopy: Speichern-Button „.txt speichern"; Statusleiste „Wörter: {n} · Zeichen: {n}"; Schließwarnung „Es gibt ungespeicherte Änderungen. Verwerfen?"
REQUIREMENTS	1. Das Dokument in einem fensterfüllenden `<textarea>` anzeigen. 2. Die Werkzeugleiste MUSS bereitstellen: Neu, Öffnen (.txt), Speichern (lädt .txt herunter), Wortzahl. 3. Öffnen MUSS über FileReader lesen und das Dokument ersetzen. 4. Speichern MUSS einen Download als UTF-8-Klartext auslösen. 5. Die Wortzahl MUSS bei jedem Tastendruck aktualisiert und in einer Statusleiste angezeigt werden. 6. Das Dokument MUSS bei jeder Änderung in localStorage persistieren und beim Laden wiederhergestellt werden. 7. Strg/Cmd+S MUSS Speichern auslösen; Strg/Cmd+O MUSS den Dateidialog öffnen. 8. Beim Schließen des Tabs MUSS bei ungespeicherten Änderungen gewarnt werden (beforeunload).
I/O & EDGES	Eingaben: Tastendrücke; .txt-Dateien bis 5 MB. Ausgaben: eine heruntergeladene .txt-Datei, UTF-8. Sonderfälle: leeres Dokument beim ersten Start; Wiederherstellen einer 4-MB-Datei aus localStorage; Einfügen binären Mülls (darf nicht abstürzen); Dateinamen mit Leerzeichen. Beispiel: `notes.txt` mit „hello\nworld" öffnen → das textarea zeigt zwei Zeilen, Wortzahl = 2.
PROCESS	Schritt 1 — Schlage den Dateibaum (Components, Hooks, Utils) und die TypeScript-Typen für AppState vor. Warte auf meine Freigabe. Schritt 2 — Schreibe Vitest-Tests für: Wortzahl-Funktion, Dateinamen-Ableitung beim Speichern, localStorage-Roundtrip, Tastenkürzel. Schritt 3 — Implementiere; prüfe dann selbst gegen die acht Anforderungen oben und liste alle Annahmen auf.
OUTPUT	Vollständige Dateipfade vor jedem Block (z.B. `src/App.tsx`). Produktionsreif, keine Auslassungen, keine Platzhalter-Kommentare. Bei Änderungen an bestehenden Dateien die ganze aktualisierte Datei zurückgeben.

Rote Linie = die nackte Vorlage. Als Ausgangspunkt einfügen und jede {Klammer} ersetzen.

Grüne Linie = eine vollständig ausgefüllte Instanz. Dasselbe Gerüst, konkretisiert für ein kleines Produkt.

ANWENDUNG **1.** Block 1 in den Chat kopieren. **2.** Jede `{Klammer}` mit den Spezifika Ihres Projekts füllen — Block 2 zeigt das Detailniveau, das funktioniert. **3.** Den Prompt abschicken und bei **Schritt 1** auf den vom Modell vorgeschlagenen Dateibaum warten, bevor Sie die Schritte 2 & 3 freigeben. **4.** Das Ergebnis als Entwurf behandeln: Annahmenliste prüfen, Tests laufen lassen, dann iterieren.

BEGLEITUNG ZU SEITE 1 · Das Acht-Block-Gerüst stützt sich auf die dort zitierten Quellen — insbesondere Arora et al. (2024) zum schrittweisen Prompting, Mosofsky's *Spec-Then-Code* zur Risiko-Sorgfalt-Kalibrierung und die empirischen Leitlinien von Mastropaolo et al. (2026). Das Ziel „Texteditor bauen" ist illustrativ; dieselbe Struktur trägt auch ein CLI-Werkzeug, einen Microservice oder eine Datenpipeline.