Writing prompts that make an LLM build your software.

BY MARC GLOOR · PAGE 1 OF 2 — NINE ESSENTIALS & THE REFERENCE TEMPLATE

The model's success in translating an idea into working code depends almost entirely on the clarity and structure of the request. Recent work in requirements engineering and code generation converges on the same conclusion: one-line prompts produce one-line software. What follows is a distilled set of practices — drawn from peer-reviewed studies and practitioner guides — for turning requirements into prompts an LLM can actually execute.

NINE ESSENTIALS

From requirements to code, terminal to browser — the practices that move output from prototype-grade to production-grade.

1 Set role and context first

Open with a persona — "Act as a senior backend engineer in a service-oriented architecture" — and a paragraph naming the project, its constraints, and the standards it must follow. A persona implicitly imports priorities (security, scalability, idiomatic style) the model would otherwise have to guess.

2 Use assertive, specific language

Replace "should" and "may" with "must" and "is required to". Vague verbs invite vague code. Name files, classes, methods, and existing patterns the new code must follow rather than describing them abstractly.

> ✗ "Write a function to process user data."
> ✓ "Add *transformUser* to *UserProcessor.js*, mirroring *transformPaymentData*."

3 Pin the technical environment

State the language and exact versions, the framework, the libraries that are allowed (and forbidden), the runtime, the target platform, and the coding conventions. Without this, the model defaults to majority training-data patterns, which often contradict the codebase you actually have.

4 Structure requirements as a numbered list

Decompose what the software must do into discrete, numbered, testable items. Numbered lists are processed more reliably than prose and force you to confront ambiguities before the model does. Group functional, non-functional, and interface requirements separately.

5 Define inputs, outputs, and edge cases

Specify the I/O format, pre- and post-conditions, error modes, and at least two or three concrete examples — including a corner case. Few-shot examples consistently outperform abstract specifications, especially for data-shape-sensitive tasks.

6 Use progressive prompting

For anything beyond a single function, do not request code in one shot. Walk the model through a stepwise chain: refine requirements, derive a design, write tests, write code. Each stage's output becomes the next stage's input. This mirrors waterfall and gives you a review surface at every step.

7 Build verification into the prompt

Ask the model to write tests before — or alongside — the implementation, and to self-review the result against your acceptance criteria, listing assumptions and unhandled edges. Tests are the executable form of your requirements; without them, you cannot tell whether the code meets them.

8 Match rigour to risk

A throwaway script tolerates a "lazy" prompt; a payment flow does not. Calibrate effort to stakes: low-risk prototypes can use loose, exploratory prompting; high-stakes features warrant a full spec-then-code pass with persisted requirements, design, and tests.

9 Specify the interface paradigm — TUI or GUI

Name the interface explicitly and pin its conventions; the model will otherwise pick defaults that rarely match your platform.

> **TUI** library (textual / curses / prompt_toolkit); key scheme (vi / emacs / named); minimum width; --json for piping; monochrome-first.
> **GUI** design system; states (empty, loading, error, success); a11y level (WCAG AA); contrast; keyboard parity; microcopy in the prompt.

REFERENCE TEMPLATE · A REUSABLE PROMPT SKELETON

The eight-block structure that survives most projects.

```
ROLE          You are a senior {language} engineer. Prioritise clarity, testability, security.
CONTEXT       Project: {name, one-line purpose}. Codebase style: {patterns, files to mirror}.
STACK         Language {X.Y}. Framework {X.Y}. Allowed libs: {…}. Forbidden: {…}.
UI / UX        Paradigm: {TUI | GUI}.
              TUI — lib {textual/curses/prompt_toolkit}; keys {vi/emacs/named}; width ≥ {N} cols; --json for piping; monochrome-first.
              GUI — design system {…}; states empty/loading/error/success; a11y {WCAG AA}; contrast ≥ 4.5:1; microcopy here.
REQUIREMENTS  1. {must-have} 2. {must-have} 3. {must-not} … (numbered, assertive, testable).
I/O & EDGES   Inputs: {types, formats}. Outputs: {types, formats}. Edge cases: {…}. Examples: {…}.
PROCESS       Step 1 — propose a function signature + docstring; wait for my approval.
              Step 2 — write failing tests for the requirements above.
              Step 3 — implement; then self-review against the requirements and list assumptions.
OUTPUT        Full file paths before each block. Production-ready, no truncation, no placeholders.
```

AVOID

One-line requirements. **Mixing what with how** in a single sentence. **Letting the model invent** the stack. **Accepting the first draft** as final. **No tests, no acceptance.**

7×

More small but consequential corrections surface when a prompt is split into **requirements → design → tests → code** rather than fired in one shot. Progressive prompting catches them *before* they reach the codebase.

> "*We advocate against the reliance on one-line requirement methodologies for LLM-based software development. Instead, we emphasise the importance of dedicating time and effort to articulate high-quality, detailed requirements.*"
>
> ARORA ET AL. · REQUIREMENTS ARE ALL YOU NEED (2024)

SOURCES · 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*.

EXAMPLE · A WORKED BUILD, END TO END

Build a text editor. Copy the template. Fill the braces.

Two blocks below. **The first** *is the bare eight-block skeleton from page one — paste it into your model of choice and replace each* `{brace}` *with your project's specifics.* **The second** *is the same skeleton already filled in for a small but realistic build target: a single-window, browser-based plain-text editor (a GUI). The TUI alternative is described in the UI / UX block of the bare template. Use either as a worked solution, or copy and adapt.*

BLOCK 1 The bare template — copy this

REPLACE EVERY {BRACE}

ROLE	You are a senior {language} engineer. Prioritise clarity, testability, {priorities}.
CONTEXT	Project: {name + one-line purpose}. Codebase style: {patterns, files to mirror}.
STACK	Language {X.Y}. Framework {X.Y}. Allowed libs: {…}. Forbidden: {…}.
UI / UX	Paradigm: {TUI | GUI}. TUI — lib {textual/curses/prompt_toolkit}; keys {vi/emacs/named}; width ≥ {N} cols; --json for piping; monochrome-first. GUI — design system {…}; states empty/loading/error/success; a11y {WCAG AA}; contrast ≥ 4.5:1; microcopy here.
REQUIREMENTS	1. {must-have} 2. {must-have} 3. {must-not} … (numbered, assertive, testable).
I/O & EDGES	Inputs: {types, formats}. Outputs: {types, formats}. Edge cases: {…}. Examples: {…}.
PROCESS	Step 1 — propose file structure + types; wait for approval. Step 2 — write failing tests for the requirements. Step 3 — implement; self-review; list assumptions.
OUTPUT	Full file paths before each block. Production-ready, no truncation, no placeholders.

BLOCK 2 Worked example — a browser-based text editor

COPY & ADAPT

ROLE	You are a senior TypeScript / React engineer. Prioritise clarity, testability, accessibility. Prefer simple, idiomatic code over clever abstractions.
CONTEXT	Project: **NotePad** — a single-window plain-text editor running in the browser. Style: small focused components, pure functions for text operations, no global state. Mirror the patterns in `src/components/Toolbar.tsx`.
STACK	TypeScript 5.4. React 18.3. Vite 5. CSS Modules. Allowed libs: `react`, `react-dom` only. Forbidden: Redux, MobX, jQuery, any rich-text library (Quill, Slate, CKEditor). The editor MUST be a controlled `<textarea>`, not `contenteditable`.
UI / UX	Paradigm: **GUI** (single-window web app). Design: minimal — system fonts, neutral palette, no chrome but the toolbar. States: empty doc → centred placeholder "Start typing or open a file"; loading → spinner in toolbar; error → top toast, dismissable; success (saved) → 2 s "Saved" badge. A11y (WCAG AA): full keyboard parity, no mouse-only actions; `<textarea>` labelled; toolbar buttons with `aria-label`; visible focus ring; contrast ≥ 4.5:1; respects `prefers-reduced-motion`. Microcopy: Save button "Save .txt"; status bar "Words: {n} · Chars: {n}"; close-warning "You have unsaved changes. Discard?"
REQUIREMENTS	1. Display the document in a full-window `<textarea>`. 2. Toolbar MUST provide: New, Open (.txt), Save (downloads .txt), Word Count. 3. Open MUST read via FileReader and replace the document. 4. Save MUST trigger a download of the document as UTF-8 plain text. 5. Word count MUST update on every keystroke and show in a status bar. 6. The document MUST persist to localStorage on every change and restore on load. 7. Ctrl/Cmd+S MUST trigger Save; Ctrl/Cmd+O MUST open the file picker. 8. The app MUST warn on tab close if there are unsaved changes (beforeunload).
I/O & EDGES	Inputs: keystrokes; .txt files up to 5 MB. Outputs: a downloaded .txt file, UTF-8. Edges: empty document on first run; restoring a 4 MB file from localStorage; pasting binary garbage (must not crash); filenames with spaces. Example: open `notes.txt` containing "hello\nworld" → textarea shows two lines, word count = 2.
PROCESS	Step 1 — Propose the file structure (components, hooks, utils) and the TypeScript types for AppState. Wait for my approval. Step 2 — Write Vitest tests covering: word-count function, save filename derivation, localStorage round-trip, keyboard-shortcut handling. Step 3 — Implement; then self-review against the eight requirements above and list any assumptions.
OUTPUT	Full file paths before each block (e.g. `src/App.tsx`). Production-ready, no truncation, no placeholder comments. For changes to existing files, return the full updated file.

Red rule = the bare template. Paste it as your starting point, then replace every {brace}.

Green rule = a fully filled-in instance. Same skeleton, made concrete for one small product.

HOW TO USE **1.** Copy Block 1 into the chat. **2.** Replace each `{brace}` with your project's specifics — Block 2 shows the level of detail that works. **3.** Send the prompt and wait at **Step 1** for the model's proposed file structure before approving Steps 2 & 3. **4.** Treat the result as a draft: review the assumptions list, run the tests, then iterate.

COMPANION TO PAGE 1 · The eight-block skeleton draws on the same sources cited there — chiefly Arora et al. (2024) on progressive prompting, Mosofsky's *Spec-Then-Code* on rigour-to-risk calibration, and the empirical guidelines of Mastropaolo et al. (2026). The "build a text editor" target is illustrative; the same structure works for a CLI tool, a microservice or a data pipeline.