Escribir prompts que hagan que un LLM construya tu software.

BY **MARC GLOOR** · PÁGINA 1 DE 2 — NUEVE REGLAS ESENCIALES Y LA PLANTILLA DE REFERENCIA

Que un modelo logre traducir una idea en código funcional depende casi por completo de la claridad y la estructura de la petición. Los trabajos recientes en ingeniería de requisitos y generación de código convergen en la misma conclusión: los prompts de una línea producen software de una línea. A continuación, un conjunto destilado de prácticas — extraído de estudios revisados por pares y guías profesionales — para convertir requisitos en prompts que un LLM realmente pueda ejecutar.

NUEVE REGLAS ESENCIALES

De los requisitos al código, del terminal al navegador: las prácticas que llevan el resultado del prototipo a la producción.

1 Fijar primero el rol y el contexto

Empieza con un rol — «Actúa como ingeniero backend senior en una arquitectura orientada a servicios» — y un párrafo que nombre el proyecto, sus restricciones y los estándares que debe respetar. Un rol importa de forma implícita prioridades (seguridad, escalabilidad, estilo idiomático) que, de otro modo, el modelo tendría que adivinar.

2 Usar un lenguaje asertivo y específico

Sustituye «debería» y «podría» por «debe» y «está obligado a». Los verbos vagos invitan a código vago. Nombra los archivos, clases, métodos y patrones existentes que el nuevo código debe seguir, en lugar de describirlos en abstracto.

> ✗ «Escribe una función para procesar datos de usuario.»
> ✓ «Añade *transformUser* en *UserProcessor.js*, en la línea de *transformPaymentData*.»

3 Fijar el entorno técnico

Indica el lenguaje con las versiones exactas, el framework, las librerías permitidas (y prohibidas), el runtime, la plataforma de destino y las convenciones de código. Sin eso, el modelo recurre a los patrones mayoritarios de sus datos de entrenamiento, que a menudo contradicen tu código real.

4 Estructurar los requisitos como lista numerada

Descompón lo que el software debe hacer en puntos separados, numerados y verificables. Las listas numeradas se procesan con más fiabilidad que el texto corrido y te obligan a resolver ambigüedades antes que el modelo. Agrupa por separado los requisitos funcionales, no funcionales y de interfaz.

5 Definir entradas, salidas y casos límite

Especifica el formato de E/S, las pre- y poscondiciones, los modos de error y al menos dos o tres ejemplos concretos — incluyendo un caso límite. Los ejemplos few-shot superan de forma sistemática a las especificaciones abstractas, sobre todo en tareas sensibles a la forma de los datos.

6 Usar prompting progresivo

Para cualquier cosa que pase de una sola función, no pidas el código de un golpe. Lleva al modelo por una cadena por etapas: afinar requisitos, derivar un diseño, escribir tests, escribir código. La salida de cada etapa se convierte en la entrada de la siguiente. Imita al ciclo en cascada y te ofrece una superficie de revisión en cada paso.

7 Integrar la verificación en el prompt

Pídele al modelo que escriba los tests antes — o en paralelo — a la implementación, y que revise él mismo el resultado frente a tus criterios de aceptación, enumerando los supuestos y los casos límite no atendidos. Los tests son la forma ejecutable de tus requisitos; sin ellos no puedes saber si el código los cumple.

8 Calibrar el rigor al riesgo

Un script desechable tolera un prompt «perezoso»; un flujo de pago, no. Ajusta el esfuerzo al riesgo: los prototipos de bajo riesgo admiten un prompting suelto y exploratorio; las funciones críticas justifican una pasada completa de especificar antes de codificar, con requisitos, diseño y tests guardados.

9 Indicar el paradigma de interfaz: TUI o GUI

Nombra la interfaz de forma explícita y fija sus convenciones; en caso contrario, el modelo escogerá valores por defecto que rara vez encajan con tu plataforma.

> **TUI** librería (textual / curses / prompt_toolkit); esquema de teclas (vi / emacs / con nombre); ancho mínimo; --json para piping; monocromo por defecto.
> **GUI** sistema de diseño; estados (vacío, cargando, error, éxito); accesibilidad (WCAG AA); contraste; paridad de teclado; microcopy en el prompt.

PLANTILLA DE REFERENCIA · UN ESQUELETO DE PROMPT REUTILIZABLE

La estructura de ocho bloques que aguanta en la mayoría de los proyectos.

```
ROLE          Eres ingeniero {lenguaje} senior. Prioriza claridad, testabilidad, seguridad.
CONTEXT       Proyecto: {nombre, propósito en una línea}. Estilo del código: {patrones, archivos a imitar}.
STACK         Lenguaje {X.Y}. Framework {X.Y}. Librerías permitidas: {…}. Prohibidas: {…}.
UI / UX        Paradigma: {TUI | GUI}.
              TUI — librería {textual/curses/prompt_toolkit}; teclas {vi/emacs/con nombre}; ancho ≥ {N} columnas; --json para piping; monocromo por defecto.
              GUI — sistema de diseño {…}; estados vacío/cargando/error/éxito; accesibilidad {WCAG AA}; contraste ≥ 4,5:1; microcopy aquí.
REQUIREMENTS  1. {obligatorio} 2. {obligatorio} 3. {prohibido} … (numerado, asertivo, verificable).
I/O & EDGES   Entradas: {tipos, formatos}. Salidas: {tipos, formatos}. Casos límite: {…}. Ejemplos: {…}.
PROCESS       Paso 1 — propón una firma de función + docstring; espera mi aprobación.
              Paso 2 — escribe tests que fallen para los requisitos anteriores.
              Paso 3 — implementa; luego autorrevisa frente a los requisitos y enumera los supuestos.
OUTPUT        Rutas completas de archivo antes de cada bloque. Listo para producción, sin truncamientos, sin marcadores de posición.
```

EVITAR

Requisitos de una línea. **Mezclar el qué con el cómo** en la misma frase. **Dejar que el modelo** se invente el stack. **Aceptar el primer borrador** como definitivo. **Sin tests, no hay aceptación.**

7×

Aparecen más correcciones pequeñas pero relevantes cuando un prompt se divide en **requisitos → diseño → tests → código** en lugar de lanzarse de una sola vez. El prompting progresivo las atrapa *antes* de que lleguen al código.

«Desaconsejamos depender de metodologías de requisitos de una línea en el desarrollo de software basado en LLM. En su lugar, subrayamos la importancia de dedicar tiempo y esfuerzo a articular requisitos detallados y de alta calidad.»

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

FUENTES · 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*.

EJEMPLO · UNA BUILD COMPLETA, DE PRINCIPIO A FIN

Construye un editor de texto. Copia la plantilla. Rellena las llaves.

Dos bloques abajo. **El primero** *es el esqueleto desnudo de ocho bloques de la página uno: pégalo en el modelo que prefieras y sustituye cada* `{llave}` *por los detalles de tu proyecto.* **El segundo** *es el mismo esqueleto, ya rellenado para un objetivo pequeño pero realista: un editor de texto plano de una sola ventana en el navegador (una GUI). La alternativa TUI se describe en el bloque UI / UX de la plantilla desnuda. Úsalos como solución modelo, o cópialos y adáptalos.*

BLOCK 1 — La plantilla desnuda — copia esto

SUSTITUYE CADA {LLAVE}

ROLE	Eres ingeniero {lenguaje} senior. Prioriza claridad, testabilidad, {prioridades}.
CONTEXT	Proyecto: {nombre + propósito en una línea}. Estilo del código: {patrones, archivos a imitar}.
STACK	Lenguaje {X.Y}. Framework {X.Y}. Librerías permitidas: {…}. Prohibidas: {…}.
UI / UX	Paradigma: {TUI | GUI}. TUI — librería {textual/curses/prompt_toolkit}; teclas {vi/emacs/con nombre}; ancho ≥ {N} columnas; --json para piping; monocromo por defecto. GUI — sistema de diseño {…}; estados vacío/cargando/error/éxito; accesibilidad {WCAG AA}; contraste ≥ 4.5:1; microcopy aquí.
REQUIREMENTS	1. {obligatorio} 2. {obligatorio} 3. {prohibido} … (numerado, asertivo, verificable).
I/O & EDGES	Entradas: {tipos, formatos}. Salidas: {tipos, formatos}. Casos límite: {…}. Ejemplos: {…}.
PROCESS	Paso 1 — propón la estructura de archivos + tipos; espera aprobación. Paso 2 — escribe tests que fallen para los requisitos. Paso 3 — implementa; autorrevisa; enumera supuestos.
OUTPUT	Rutas completas de archivo antes de cada bloque. Listo para producción, sin truncamientos, sin marcadores de posición.

BLOCK 2 — Ejemplo resuelto — un editor de texto en el navegador

COPIAR Y ADAPTAR

ROLE	Eres ingeniero TypeScript / React senior. Prioriza claridad, testabilidad y accesibilidad. Prefiere código sencillo e idiomático antes que abstracciones ingeniosas.
CONTEXT	Proyecto: **NotePad**: un editor de texto plano de una sola ventana que corre en el navegador. Estilo: componentes pequeños y enfocados, funciones puras para las operaciones de texto, sin estado global. Sigue los patrones de `src/components/Toolbar.tsx`.
STACK	TypeScript 5.4. React 18.3. Vite 5. CSS Modules. Librerías permitidas: solo `react` y `react-dom`. Prohibidas: Redux, MobX, jQuery y cualquier librería rich-text (Quill, Slate, CKEditor). El editor DEBE ser un `<textarea>` controlado, no `contenteditable`.
UI / UX	Paradigma: **GUI** (web app de una sola ventana). Diseño: minimalista: tipografías del sistema, paleta neutra, sin adornos salvo la barra de herramientas. Estados: documento vacío → marcador de posición centrado «Empieza a escribir o abre un archivo»; cargando → spinner en la barra; error → toast superior, descartable; éxito (guardado) → distintivo «Guardado» durante 2 s. Accesibilidad (WCAG AA): paridad completa de teclado, sin acciones exclusivas de ratón; `<textarea>` etiquetado; botones de la barra con `aria-label`; anillo de foco visible; contraste ≥ 4,5:1; respeta `prefers-reduced-motion`. Microcopy: botón Guardar «Guardar .txt»; barra de estado «Palabras: {n} · Caracteres: {n}»; aviso al cerrar «Hay cambios sin guardar. ¿Descartarlos?»
REQUIREMENTS	1. Mostrar el documento en un `<textarea>` que ocupe toda la ventana. 2. La barra de herramientas DEBE ofrecer: Nuevo, Abrir (.txt), Guardar (descarga .txt), Conteo de palabras. 3. Abrir DEBE leer mediante FileReader y reemplazar el documento. 4. Guardar DEBE activar la descarga del documento como texto plano UTF-8. 5. El conteo de palabras DEBE actualizarse en cada pulsación y mostrarse en una barra de estado. 6. El documento DEBE persistirse en localStorage en cada cambio y restaurarse al cargar. 7. Ctrl/Cmd+S DEBE activar Guardar; Ctrl/Cmd+O DEBE abrir el selector de archivos. 8. La app DEBE avisar al cerrar la pestaña si hay cambios sin guardar (beforeunload).
I/O & EDGES	Entradas: pulsaciones de teclado; archivos .txt de hasta 5 MB. Salidas: un archivo .txt descargado, UTF-8. Casos límite: documento vacío en el primer arranque; restaurar un archivo de 4 MB desde localStorage; pegar basura binaria (no debe romper la app); nombres de archivo con espacios. Ejemplo: abrir `notes.txt` con «hello\nworld» → el textarea muestra dos líneas, conteo de palabras = 2.
PROCESS	Paso 1 — Propón la estructura de archivos (componentes, hooks, utilidades) y los tipos TypeScript para AppState. Espera mi aprobación. Paso 2 — Escribe tests con Vitest que cubran: función de conteo de palabras, derivación del nombre de archivo al guardar, ida y vuelta a localStorage, gestión de atajos de teclado. Paso 3 — Implementa; luego autorrevisa frente a los ocho requisitos anteriores y enumera todos los supuestos.
OUTPUT	Rutas completas de archivo antes de cada bloque (p. ej. `src/App.tsx`). Listo para producción, sin truncamientos, sin comentarios de marcador. Para cambios en archivos existentes, devuelve el archivo completo actualizado.

Línea roja = la plantilla desnuda. Pégala como punto de partida y sustituye cada `{llave}`.

Línea verde = una instancia completamente rellenada. El mismo esqueleto, concretado para un producto pequeño.

CÓMO USARLA

1. Copia el Bloque 1 en el chat. **2.** Sustituye cada `{llave}` por los detalles del proyecto — el Bloque 2 muestra el nivel de detalle que funciona. **3.** Envía el prompt y espera en el **Paso 1** a que el modelo proponga la estructura de archivos antes de aprobar los Pasos 2 y 3. **4.** Trata el resultado como un borrador: revisa la lista de supuestos, ejecuta los tests y luego itera.

COMPLEMENTO DE LA PÁGINA 1 · El esqueleto de ocho bloques se apoya en las mismas fuentes citadas allí — sobre todo Arora et al. (2024) sobre prompting progresivo, *Spec-Then-Code* de Mosofsky sobre la calibración rigor-riesgo y las directrices empíricas de Mastropaolo et al. (2026). El objetivo «construir un editor de texto» es ilustrativo; la misma estructura sirve para una herramienta CLI, un microservicio o un pipeline de datos.