Rédiger des prompts qui font construire votre logiciel par un LLM.

BY MARC GLOOR · PAGE 1 SUR 2 — NEUF PRINCIPES ESSENTIELS ET LE MODÈLE DE RÉFÉRENCE

La réussite d'un modèle à traduire une idée en code fonctionnel dépend presque entièrement de la clarté et de la structure de la demande. Les travaux récents en ingénierie des exigences et en génération de code convergent vers la même conclusion : un prompt d'une ligne produit un logiciel d'une ligne. Voici un ensemble distillé de pratiques — issues d'études évaluées par les pairs et de guides professionnels — pour transformer des exigences en prompts qu'un LLM peut réellement exécuter.

NEUF PRINCIPES ESSENTIELS

Des exigences au code, du terminal au navigateur — les pratiques qui font passer le résultat du prototype à la production.

1 Poser d'abord le rôle et le contexte

Commencez par un rôle — « Agissez comme un ingénieur backend senior dans une architecture orientée services » — et un paragraphe nommant le projet, ses contraintes et les standards à respecter. Un rôle importe implicitement des priorités (sécurité, scalabilité, style idiomatique) que le modèle devrait sinon deviner.

2 Utiliser un langage affirmatif et précis

Remplacez « devrait » et « peut » par « doit » et « est tenu de ». Les verbes flous appellent du code flou. Nommez les fichiers, classes, méthodes et patterns existants que le nouveau code doit suivre, plutôt que de les décrire abstraitement.

> ✗ « Écris une fonction pour traiter les données utilisateur. »
> ✓ « Ajoute *transformUser* dans *UserProcessor.js*, sur le modèle de *transformPaymentData*. »

3 Fixer l'environnement technique

Indiquez le langage et les versions exactes, le framework, les bibliothèques autorisées (et interdites), le runtime, la plateforme cible et les conventions de code. À défaut, le modèle retombera sur les patterns majoritaires de ses données d'entraînement, qui contredisent souvent votre base de code réelle.

4 Structurer les exigences en liste numérotée

Décomposez ce que le logiciel doit faire en éléments distincts, numérotés et testables. Les listes numérotées sont traitées plus fiablement que la prose et vous obligent à résoudre les ambiguïtés avant le modèle. Regroupez séparément les exigences fonctionnelles, non fonctionnelles et d'interface.

5 Définir entrées, sorties et cas limites

Précisez le format d'E/S, les pré- et post-conditions, les modes d'erreur et au moins deux ou trois exemples concrets — dont un cas limite. Les exemples « few-shot » l'emportent systématiquement sur les spécifications abstraites, surtout pour les tâches sensibles à la forme des données.

6 Adopter le prompting progressif

Pour tout ce qui dépasse une fonction isolée, ne demandez pas le code d'un coup. Guidez le modèle par étapes : affiner les exigences, dériver une conception, écrire les tests, écrire le code. La sortie d'une étape devient l'entrée de la suivante. Ce schéma reproduit le cycle en V et offre une surface de revue à chaque étape.

7 Intégrer la vérification dans le prompt

Demandez au modèle d'écrire les tests avant — ou en parallèle — de l'implémentation, et de relire lui-même le résultat au regard de vos critères d'acceptation, en listant les hypothèses retenues et les cas non traités. Les tests sont la forme exécutable de vos exigences ; sans eux, impossible de savoir si le code y répond.

8 Calibrer la rigueur au risque

Un script jetable tolère un prompt « paresseux » ; un flux de paiement non. Calibrez l'effort sur l'enjeu : les prototypes à faible risque autorisent un prompting libre et exploratoire ; les fonctionnalités critiques exigent un passage complet spec-puis-code, avec exigences, conception et tests conservés.

9 Préciser le paradigme d'interface — TUI ou GUI

Nommez l'interface explicitement et fixez ses conventions ; sinon le modèle choisira des valeurs par défaut qui correspondent rarement à votre plateforme.

> **TUI** bibliothèque (textual / curses / prompt_toolkit) ; schéma de touches (vi / emacs / nommé) ; largeur minimale ; --json pour le piping ; monochrome par défaut.
> **GUI** design system ; états (vide, chargement, erreur, succès) ; accessibilité (WCAG AA) ; contraste ; parité clavier ; microcopy dans le prompt.

MODÈLE DE RÉFÉRENCE · UN SQUELETTE DE PROMPT RÉUTILISABLE

La structure en huit blocs qui tient sur la plupart des projets.

```
ROLE          Vous êtes un ingénieur {langage} senior. Priorités : clarté, testabilité, sécurité.
CONTEXT       Projet : {nom, objet en une ligne}. Style de la base : {patterns, fichiers à imiter}.
STACK         Langage {X.Y}. Framework {X.Y}. Bibliothèques autorisées : {…}. Interdites : {…}.
UI / UX        Paradigme : {TUI | GUI}.
              TUI — bibliothèque {textual/curses/prompt_toolkit} ; touches {vi/emacs/nommées} ; largeur ≥ {N} colonnes ; --json pour le piping ; monochrome
par défaut.
              GUI — design system {…} ; états vide/chargement/erreur/succès ; accessibilité {WCAG AA} ; contraste ≥ 4,5:1 ; microcopy ici.
REQUIREMENTS  1. {obligatoire} 2. {obligatoire} 3. {interdit} … (numéroté, affirmatif, testable).
I/O & EDGES   Entrées : {types, formats}. Sorties : {types, formats}. Cas limites : {…}. Exemples : {…}.
PROCESS       Étape 1 — proposer une signature de fonction + docstring ; attendre ma validation.
              Étape 2 — écrire des tests qui échouent pour les exigences ci-dessus.
              Étape 3 — implémenter ; puis relire soi-même au regard des exigences et lister les hypothèses.
OUTPUT        Chemins complets des fichiers avant chaque bloc. Prêt pour la production, sans troncatures, sans espaces réservés.
```

À ÉVITER

Exigences d'une ligne. **Mélanger le quoi et le comment** dans une même phrase. **Laisser le modèle** inventer la stack. **Accepter le premier jet** comme version finale. **Pas de tests, pas d'acceptation.**

7×

Davantage de petites corrections d'importance émergent lorsqu'un prompt est découpé en **exigences → conception → tests → code** plutôt que tiré d'un coup. Le prompting progressif les attrape *avant* qu'elles n'atteignent la base de code.

« Nous déconseillons les méthodologies fondées sur des exigences d'une ligne pour le développement logiciel assisté par LLM. Nous soulignons au contraire l'importance de consacrer temps et énergie à formuler des exigences détaillées et de qualité. »

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*.

EXEMPLE · UNE CONSTRUCTION COMPLÈTE, DE BOUT EN BOUT

Construisez un éditeur de texte. Copiez le modèle. Remplissez les accolades.

Deux blocs ci-dessous. **Le premier** *est le squelette nu en huit blocs de la page un — collez-le dans le modèle de votre choix et remplacez chaque* `{accolade}` *par les particularités de votre projet.* **Le second** *est le même squelette, déjà rempli pour une cible petite mais réaliste : un éditeur de texte brut à fenêtre unique, fonctionnant dans le navigateur (une GUI). La variante TUI est décrite dans le bloc UI / UX du modèle nu. Servez-vous de l'un comme solution-type, ou copiez et adaptez.*

BLOCK 1 — Le modèle nu — à copier

REMPLACER CHAQUE {ACCOLADE}

ROLE	Vous êtes un ingénieur {langage} senior. Priorités : clarté, testabilité, {priorités}.
CONTEXT	Projet : {nom + objet en une ligne}. Style de la base : {patterns, fichiers à imiter}.
STACK	Langage {X.Y}. Framework {X.Y}. Bibliothèques autorisées : {…}. Interdites : {…}.
UI / UX	Paradigme : {TUI | GUI}. TUI — bibliothèque {textual/curses/prompt_toolkit} ; touches {vi/emacs/nommées} ; largeur ≥ {N} colonnes ; --json pour le piping ; monochrome par défaut. GUI — design system {…} ; états vide/chargement/erreur/succès ; accessibilité {WCAG AA} ; contraste ≥ 4,5:1 ; microcopy ici.
REQUIREMENTS	1. {obligatoire} 2. {obligatoire} 3. {interdit} … (numéroté, affirmatif, testable).
I/O & EDGES	Entrées : {types, formats}. Sorties : {types, formats}. Cas limites : {…}. Exemples : {…}.
PROCESS	Étape 1 — proposer la structure des fichiers + les types ; attendre validation. Étape 2 — écrire des tests qui échouent pour les exigences. Étape 3 — implémenter ; relire soi-même ; lister les hypothèses.
OUTPUT	Chemins complets des fichiers avant chaque bloc. Prêt pour la production, sans troncatures, sans espaces réservés.

BLOCK 2 — Exemple complet — un éditeur de texte dans le navigateur

COPIER ET ADAPTER

ROLE	Vous êtes un ingénieur TypeScript / React senior. Priorités : clarté, testabilité, accessibilité. Préférez un code simple et idiomatique aux abstractions astucieuses.
CONTEXT	Projet : **NotePad** — un éditeur de texte brut à fenêtre unique tournant dans le navigateur. Style : composants courts et focalisés, fonctions pures pour les opérations sur le texte, pas d'état global. Reprenez les patterns de `src/components/Toolbar.tsx`.
STACK	TypeScript 5.4. React 18.3. Vite 5. CSS Modules. Bibliothèques autorisées : `react` et `react-dom` uniquement. Interdites : Redux, MobX, jQuery, toute bibliothèque rich-text (Quill, Slate, CKEditor). L'éditeur DOIT être un `<textarea>` contrôlé, pas un `contenteditable`.
UI / UX	Paradigme : **GUI** (web app à fenêtre unique). Design : minimal — polices système, palette neutre, pas d'habillage hormis la barre d'outils. États : document vide → placeholder centré « Commencez à taper ou ouvrez un fichier » ; chargement → spinner dans la barre d'outils ; erreur → toast en haut, fermable ; succès (enregistré) → badge « Enregistré » de 2 s. Accessibilité (WCAG AA) : parité clavier complète, aucune action réservée à la souris ; `<textarea>` étiqueté ; boutons de la barre d'outils avec `aria-label` ; anneau de focus visible ; contraste ≥ 4,5:1 ; respecte `prefers-reduced-motion`. Microcopy : bouton « Enregistrer .txt » ; barre d'état « Mots : {n} · Caractères : {n} » ; avertissement de fermeture « Modifications non enregistrées. Abandonner ? »
REQUIREMENTS	1. Afficher le document dans un `<textarea>` occupant toute la fenêtre. 2. La barre d'outils DOIT fournir : Nouveau, Ouvrir (.txt), Enregistrer (téléchargement .txt), Nombre de mots. 3. Ouvrir DOIT lire via FileReader et remplacer le document. 4. Enregistrer DOIT déclencher un téléchargement en texte brut UTF-8. 5. Le nombre de mots DOIT se mettre à jour à chaque frappe et apparaître dans une barre d'état. 6. Le document DOIT être persisté dans localStorage à chaque modification et restauré au chargement. 7. Ctrl/Cmd+S DOIT déclencher Enregistrer ; Ctrl/Cmd+O DOIT ouvrir le sélecteur de fichier. 8. L'application DOIT avertir à la fermeture de l'onglet en cas de modifications non enregistrées (beforeunload).
I/O & EDGES	Entrées : frappes clavier ; fichiers .txt jusqu'à 5 Mo. Sorties : un fichier .txt téléchargé, UTF-8. Cas limites : document vide au premier lancement ; restauration d'un fichier de 4 Mo depuis localStorage ; collage de binaires aléatoires (ne doit pas planter) ; noms de fichiers avec espaces. Exemple : ouvrir `notes.txt` contenant « hello\nworld » → le textarea affiche deux lignes, nombre de mots = 2.
PROCESS	Étape 1 — Proposer la structure des fichiers (composants, hooks, utilitaires) et les types TypeScript pour AppState. Attendre ma validation. Étape 2 — Écrire des tests Vitest couvrant : fonction de comptage de mots, dérivation du nom de fichier à l'enregistrement, aller-retour localStorage, gestion des raccourcis clavier. Étape 3 — Implémenter ; puis relire soi-même au regard des huit exigences ci-dessus et lister toutes les hypothèses.
OUTPUT	Chemins complets des fichiers avant chaque bloc (par ex. `src/App.tsx`). Prêt pour la production, sans troncatures, sans commentaires « placeholder ». Pour les modifications de fichiers existants, renvoyer le fichier complet mis à jour.

Filet rouge = le modèle nu. Collez-le comme point de départ, puis remplacez chaque `{accolade}`. **Filet vert** = une instance entièrement remplie. Même squelette, rendu concret pour un petit produit.

MODE D'EMPLOI

1. Copiez le Bloc 1 dans la conversation. **2.** Remplacez chaque `{accolade}` par les particularités de votre projet — le Bloc 2 montre le niveau de détail qui fonctionne. **3.** Envoyez le prompt et attendez à l'**étape 1** que le modèle propose une structure de fichiers avant de valider les étapes 2 et 3. **4.** Traitez le résultat comme un brouillon : examinez la liste des hypothèses, exécutez les tests, puis itérez.

COMPLÉMENT DE LA PAGE 1 · Le squelette en huit blocs s'appuie sur les mêmes sources qu'en page 1 — principalement Arora et al. (2024) sur le prompting progressif, *Spec-Then-Code* de Mosofsky sur le calibrage rigueur-risque, et les recommandations empiriques de Mastropaolo et al. (2026). La cible « construire un éditeur de texte » est illustrative ; la même structure convient à un outil CLI, à un microservice ou à un pipeline de données.