200 PROMPTS QUE TODO PROGRAMADOR DEBE CONOCER

(Y OTROS CONSEJOS PARA CREAR EL PROMPT PERFECTO)

BIG school

Como desarrollador (especialmente si estás iniciando tu carrera), aprender a formular prompts efectivos, es decir, solicitudes o preguntas claras, puede marcar la diferencia entre obtener una solución útil o una respuesta confusa. Un prompt bien planteado es como una buena pregunta que le harías a un mentor o a una IA: si la pregunta es vaga, la respuesta también lo será, pero si proporcionas contexto y detalles específicos, obtendrás resultados de calidad.

ANATOMÍA DEL PROMPT PERFECTO

Un buen prompt es similar a un documento bien redactado o a una "user story" clara: necesita incluir ciertos elementos para no dejar lugar a la ambigüedad. A continuación, desglosamos la anatomía de un prompt profesional, con secciones recomendadas que te ayudarán a obtener respuestas precisas y útiles de una IA o de cualquier interlocutor técnico:

1. Rol o Persona

Qué es: Indica quién debe "ser" la IA o el interlocutor al responder. Por ejemplo, especifica "Actúa como un desarrollador senior en backend especializado en seguridad".

Por qué funciona: Le das un contexto de rol y expertise al modelo, para guiar el tono y nivel de la respuesta. No es lo mismo pedir ayuda a "un experto en ciberseguridad" que a "un profesor paciente de programación". Definir el rol ajusta el conocimiento y estilo de la respuesta.

2. Objetivo o Tarea

Qué es: Describe claramente qué necesitas. Es el objetivo específico: la pregunta a responder o la funcionalidad a implementar. Ejemplo: "Crea una función que valide emails" en lugar de "Háblame sobre validación".

Por qué funciona: Enfoca al ayudante (humano o IA) en la tarea exacta. Cuanto más específico sea el objetivo, más precisa será la respuesta. Definir claramente lo que esperas evita interpretaciones erróneas.

3. Contexto

Qué es: Proporciona el trasfondo o la información de fondo relevante. Por ejemplo: "Estoy desarrollando una app de e-commerce en Node.js con MongoDB, y necesito validar formularios del lado del servidor".

Por qué funciona: Eliminas suposiciones proporcionando detalles del entorno, restricciones tecnológicas o propósito del proyecto. Cuanto más contexto (sin excederse en irrelevancia), mejor se adapta la respuesta a tu caso.

4. Instrucciones o Pasos (si aplica)

Qué es: Si la tarea es compleja, puedes desglosarla en pasos o indicar un enfoque estructurado. Ejemplo: "Paso 1: leer entrada; Paso 2: validar formato; Paso 3: devolver resultado…".

Por qué funciona: Obliga a una estructura lógica en la solución. Es útil cuando quieres que la respuesta siga un orden determinado o cuando buscas un resultado paso a paso (ej. para resolución de problemas). Si la solución falla, puedes identificar en qué paso hubo un error.

5. Formato de la Respuesta

Qué es: Especifica cómo deseas recibir la respuesta. ¿En texto plano, en formato de lista, con ejemplos de código, solo código sin explicaciones? Ejemplo: "Responde solo con el código Python dentro de un bloque de código, sin texto adicional".

Por qué funciona: Garantiza que el resultado sea fácil de usar para ti. Esto ahorra tiempo limpiando la respuesta y es esencial si planeas copiar código o integrarlo directamente.

6. Restricciones y Advertencias

Qué es: Establece límites o requisitos que no deben violarse. Pueden ser restricciones técnicas (por ejemplo: "No uses librerías externas"), de estilo ("Sigue las guías de estilo de Airbnb") o advertencias para evitar suposiciones incorrectas.

Por qué funciona: Acota las posibles soluciones y asegura que la respuesta cumpla ciertos estándares o reglas. Además, añadir advertencias puede prevenir errores; por ejemplo, pedir "no inventar datos si falta información" mejora la precisión.

7. Cláusula de Clarificación (Opcional)

Qué es: Una última instrucción que invite a preguntar aclaraciones si falta información. Por ejemplo: "Si necesitas más datos para dar una respuesta precisa, pregúntame antes de asumir".

Por qué funciona: Transforma la interacción en un diálogo colaborativo. En lugar de arriesgarse a "alucinar" o asumir cosas, la IA pedirá detalles adicionales. Esto mejora la fiabilidad de la solución y te ahorra tiempo de depuración por malentendidos.

¿Por qué tanta estructura? Porque un prompt bien estructurado es como un plano claro para construir una solución. Si aplicas estas secciones, estarás diciéndole exactamente qué quieres al "dev junior súper rápido" que es tu IA, tal como lo harías al asignar tareas en la vida real. En la práctica, no siempre necesitarás todos los apartados, pero tenerlos en mente te ayudará a no olvidar nada importante. Piensa en tu prompt como un ticket completo: contexto, objetivo, criterios de aceptación… Mientras más claro y detallado (relevante) seas, mejor será el resultado.

Esta estructura tan completa funciona especialmente bien en modelos razonadores, los cuales utilizan mayor tiempo y capacidad de cómputo para elaborar las respuestas. Por supuesto, no es necesario que escribas todos los prompts siguiendo un esquema tan profundo, pero valora hacerlo si así lo requiere tu problema.

A continuación encontrarás el listado de plantillas (divididas en prompts RÁPIDOS, BÁSICOS y AVANZADOS) que servirán para que cada persona pueda entender de una manera detallada y efectiva cómo crear sus propios prompts basándose en estos ejemplos reales. Ten en cuenta que para obtener los mejores resultados debes seguir la estructura detallada en la sección de anatomía de un prompt perfecto.

PROMPTS RÁPIDOS

Esta sección presenta 100 plantillas de prompts diseñadas para abordar tareas comunes a lo largo del ciclo de vida del desarrollo de software. No es una lista para memorizar, sino un conjunto de patrones para adaptar. La habilidad clave reside en reconocer qué patrón de prompt es el más adecuado para un problema de programación específico.

El uso de estas plantillas estructuradas va más allá de la simple obtención de una respuesta. Un desarrollador junior a menudo no sabe qué preguntas hacer o qué aspectos críticos considerar, como la seguridad, el rendimiento o los casos límite. Al incorporar explícitamente estas consideraciones en las plantillas, el prompt no solo guía a la IA, sino que también entrena al desarrollador para pensar como un ingeniero senior, fomentando un aprendizaje acelerado e implícito de las mejores prácticas.

Prompts para estudiar y aprender fundamentos

Aquí tienes recopilados 20 prompts diseñados para reforzar conceptos teóricos y fundamentos de la programación. Son preguntas que puedes hacer para entender mejor un tema o práctica, y que todo junior debería ser capaz de responder o, al menos, investigar. Usar este tipo de prompts con una IA o en tu estudio personal te ayudará a llenar vacíos de conocimiento y afianzar la base teórica, desde estructuras de datos hasta principios de diseño.

Recuerda: la programación es un continuo aprendizaje, y pedir a ChatGPT, Gemini, Claude, GitHub Copilot, Claude Code (o el modelo/herramienta que sea) que explique conceptos o código es una gran manera de entender nuevos temas.

1. Explica con tus propias palabras qué es y cómo funciona la memoria dinámica en un programa (¿qué son el heap y el stack?).

2. ¿Cuál es la diferencia entre un arreglo (array) y una lista enlazada, y en qué casos usarías cada uno?

3. Define qué es la recursividad y proporciona un ejemplo sencillo de cómo se utiliza.

4. ¿Cómo funciona el algoritmo de búsqueda binaria y cuál es su complejidad temporal Big-O?

5. Explica el principio SOLID de Responsabilidad Única y por qué es importante en el diseño de software.

6. ¿Qué son las condiciones de carrera (race conditions) y cómo puedes prevenirlas en un programa concurrente?

7. Explica la diferencia entre procesos e hilos (threads) en un sistema operativo.

8. ¿Qué es un sistema de control de versiones (ej. Git) y por qué es esencial en el desarrollo de software?

9. Enumera las etapas principales del ciclo de vida de desarrollo de software (SDLC) y describe brevemente qué se hace en cada una.

10. ¿Cuál es la diferencia entre una prueba unitaria y una prueba de integración, y cuál es el propósito de cada una?

Ejemplos más genéricos:

11. Actúa como un profesor de programación para principiantes. Explica [concepto, ej. variables en Python] paso a paso, con analogías del mundo real y un ejemplo simple de código. Usa viñetas para los pasos.

12. Soy un programador junior. Dame un tutorial corto sobre [tema, ej. bucles for en JavaScript], incluyendo sintaxis, errores comunes y 3 ejercicios prácticos con soluciones. Formato: Tabla para ejercicios.

13. Explica la diferencia entre [concepto_1] y [concepto_2, ej. listas vs tuplas en Python] como si se lo dijeras a un niño de 10 años. Luego, da ejemplos de código para cada uno.

14. Crea un mapa mental en texto para [tema, ej. programación orientada a objetos]. Incluye definiciones clave, relaciones y ejemplos en [lenguaje].

15. Lista 5 recursos gratuitos (libros, videos, sitios web) para aprender [tecnología, ej. React]. Para cada uno, describe por qué es bueno para juniors y qué cubre. Usa una tabla.

16. Explica cómo funciona [algoritmo, ej. quicksort] paso a paso con un ejemplo numérico. Incluye código en [lenguaje] y complejidad temporal.

17. Dame un glosario de 10 términos clave en [área, ej. bases de datos SQL] con definiciones simples y un ejemplo de uso. Formato: Tabla.

18. Simula una sesión de estudio: Pregúntame 5 preguntas de opción múltiple sobre [tema, ej. funciones en C++], y explica las respuestas correctas después.

19. Resume un artículo o documentación sobre [tema, ej. async/await en JavaScript]. Proporciona puntos clave, pros/contras y un ejemplo. (Primero busca el artículo si es necesario).

20. Crea un plan de estudio de una semana para aprender [lenguaje/framework, ej. FastAPI] desde cero. Incluye metas diarias y recursos.

Prompts para desarrollo e implementación de código

Estos son otros 20 prompts orientados a la escritura de código y la implementación de soluciones. Este tipo de acciones son desafíos prácticos y tareas típicas que un programador enfrenta, planteados de forma agnóstica a cualquier lenguaje (puedes resolverlos en tu lenguaje preferido). Estos prompts te servirán para practicar la lógica de programación, el diseño de algoritmos y la traducción de requerimientos en código funcional. Si utilizas una IA, dar suficiente contexto y detalles (como aprendimos en la anatomía del prompt) es crucial para obtener código útil. Incluso sin una IA, formular bien el problema es el primer paso para resolverlo.

21. Escribe un pseudocódigo para calcular el factorial de un número dado de forma iterativa.

22. Implementa una función que determine si una cadena de texto es un palíndromo (ignorando mayúsculas, espacios y acentos).

23. Diseña un algoritmo para ordenar una lista de números utilizando el método de ordenamiento burbuja (Bubble Sort).

24. Crea una función que reciba una lista de números y devuelva la suma de todos los números pares en la lista.

25. Describe cómo conectarías una base de datos a una aplicación sin usar frameworks, mencionando los pasos principales (crear conexión, ejecutar consultas, cerrar conexión, etc.).

26. Escribe un fragmento de código que abra un archivo de texto y lea su contenido línea por línea (en el lenguaje de tu preferencia).

27. ¿Cómo manejarías las excepciones o errores en un programa para evitar que este se cierre inesperadamente? (Explica una estrategia general de manejo de errores).

28. Implementa una función que realice una búsqueda de un número en una lista (array) y retorne su índice si lo encuentra o -1 si no está presente.

29. Crea una estructura de datos pila (stack) básica usando un array, con funciones para apilar (push) y desapilar (pop) elementos.

30. Traduce el siguiente fragmento de código de Python a JavaScript manteniendo la misma funcionalidad (ejemplo: un script simple de "Hola Mundo").

Ejemplos más genéricos:

31. Actúa como un desarrollador senior. Genera código en [lenguaje] para [tarea, ej. una función que calcule factorial]. Incluye comentarios y maneja errores. Formato: Bloque de código.

32. Crea una clase en [lenguaje, ej. Java] para [objeto, ej. un coche] con atributos, métodos y herencia. Explica cada parte paso a paso.

33. Escribe un script completo en [lenguaje] para [proyecto pequeño, ej. un juego de adivinanza]. Usa mejores prácticas e incluye pruebas con ejemplos.

34. Genera una API REST simple en [framework, ej. Express.js] con endpoints para CRUD en una lista de tareas. Incluye código y explicaciones.

35. Crea un componente React para [elemento, ej. un formulario de login]. Incluye estado, props y estilos básicos. Explica el flujo.

36. Implementa un patrón de diseño [patrón, ej. singleton] en [lenguaje]. Da un ejemplo real y por qué usarlo.

37. Escribe código para leer/escribir archivos en [lenguaje, ej. Python] manejando excepciones. Incluye un ejemplo con datos JSON.

38. Genera una consulta SQL para [tarea, ej. unir dos tablas]. Explica la sintaxis y posibles optimizaciones.

39. Crea un loop que procese [datos, ej. un array de números] para calcular estadísticas básicas. Usa bibliotecas si aplica.

40. Desarrolla un frontend simple con HTML/CSS/JS para [página, ej. un contador]. Hazlo responsive y accesible.

Prompts para depuración de errores (Debugging)

La depuración de errores es una parte inevitable (¡y formativa!) de la vida del programador. Aquí listamos 15 prompts enfocados en identificar y solucionar bugs y problemas en el código. Estos escenarios te harán pensar en cómo abordarías distintas situaciones de error, desde fallos comunes hasta comportamientos inesperados.

Recuerda que encontrar bugs puede consumir mucho tiempo, pero saber preguntar "¿por qué falla esto?" de forma efectiva, ya sea a un compañero, a la documentación o a una IA, puede agilizar el proceso. Plantear claramente el síntoma y el contexto del error es clave para obtener ayuda útil en debugging.

41. Tienes un programa que se queda en un bucle infinito. ¿Qué pasos seguirías para encontrar la causa del bucle y resolverlo?

42. El código arroja un error NullPointerException al ejecutarse. ¿Cómo identificarías dónde ocurre el null y cómo lo solucionarías?

43. ¿Cómo usarías un depurador (debugger) para inspeccionar el estado de un programa paso a paso mientras se ejecuta?

44. Tu aplicación web devuelve un error 500 (Internal Server Error) sin detalles adicionales. ¿Cómo procederías para encontrar la causa raíz del problema?

45. Tienes una función que debería ejecutarse en segundos, pero tarda minutos en completarse. ¿Qué técnicas usarías para identificar el cuello de botella y mejorar el rendimiento?

46. Al ejecutar un script, este no produce ninguna salida ni error. ¿Cómo investigarías el problema para determinar qué está ocurriendo?

47. Describe un método para encontrar posibles fugas de memoria (memory leaks) en una aplicación que corre durante horas.

48. Un programa lanza una excepción de "índice fuera de rango". ¿Qué podría estar causando este error y cómo lo solucionarías?

49. Tu programa compila y corre, pero el resultado que obtienes es incorrecto. ¿Cómo aislarías la parte del código donde puede estar el error lógico?

50. Menciona al menos tres técnicas o buenas prácticas generales que ayudarían a depurar errores en el código de manera más eficiente.

Ejemplos más genéricos:

51. Tengo este error: [descripción del error, ej. "TypeError: undefined"]. Aquí está mi código: [código]. Explica por qué ocurre y cómo solucionarlo paso a paso.

52. Depura este código en [lenguaje]: [código]. Identifica bugs potenciales y sugiere correcciones con explicaciones.

53. Explica técnicas de depuración en [lenguaje, ej. con debugger en VS Code]. Dame pasos para un ejemplo simple.

54. Mi programa se cuelga en [escenario, ej. loops infinitos]. Analiza este código: [código] y propone soluciones.

55. Compara errores comunes en [lenguaje1] vs [lenguaje2, ej. Python vs Java]. Lista 5 por cada uno con fixes. Formato: Tabla.

Prompts para pruebas y control de calidad (Testing)

Esta sección está enfocada en 14 prompts relacionados con testing y control de calidad del código. Aquí encontrarás ideas para generar casos de prueba, validar funcionalidades y asegurar que el código cumple con los estándares de calidad. Estos prompts te ayudarán a pensar como un tester, identificando casos límite, errores comunes y asegurando la robustez del software. Además, aprenderás a usar herramientas y frameworks para automatizar pruebas y mejorar la cobertura de código.

56. Escribe casos de prueba unitarios para una función hipotética esPrimo(n) que verifica si un número es primo.

57. ¿Cómo probarías la funcionalidad de registro de usuarios en una aplicación web? Menciona los casos de prueba principales (ej. registro exitoso, datos inválidos, usuario ya existente, etc.).

58. Genera datos de prueba (dummy data) para validar una aplicación de gestión de estudiantes (ej. lista de nombres, notas, cursos, incluyendo algunos valores atípicos).

59. Explica en qué consiste el Desarrollo Guiado por Pruebas (TDD) y cómo se aplicaría en un proyecto pequeño.

60. La aplicación funciona bien con entradas habituales, pero falla cuando se introduce un carácter especial (por ej. ñ o @). ¿Cómo crearías una prueba para ese caso y corregirías el problema?

61. ¿Qué es una prueba de regresión y por qué es importante ejecutarla después de realizar cambios o corregir un bug en el código?

62. Describe la diferencia entre pruebas automatizadas y pruebas manuales, y da un ejemplo de cuándo conviene usar cada una.

63. Tu equipo quiere asegurar un alto nivel de calidad. ¿Qué métricas o herramientas usarías para medir la calidad del código (ej. cobertura de tests, análisis estático, linting)?

64. Diseña casos de prueba para una función calculadora que realiza divisiones, asegurando incluir la división por cero como caso de borde.

Ejemplos más genéricos:

65. Escribe tests unitarios en [framework, ej. Jest] para esta función: [función]. Cubre casos edge y normales.

66. Explica qué es testing [tipo, ej. integración] y cómo implementarlo en [lenguaje]. Da un ejemplo.

67. Revisa este código para calidad: [código]. Sugiere mejoras en legibilidad, eficiencia y estándares (ej. PEP8).

68. Genera mocks para testing en [lenguaje, ej. con unittest en Python]. Usa un ejemplo con APIs externas.

69. Lista 5 mejores prácticas para escribir código testable. Explica cada una con un ejemplo corto.

Prompts para optimización, rendimiento y documentación

Aquí tienes agrupados 10 prompts diseñados para mejorar el rendimiento del código, optimizar recursos y generar documentación clara y útil. Los prompts te guiarán en cómo identificar cuellos de botella, reducir la complejidad temporal y espacial, y escribir documentación que facilite la comprensión y mantenimiento del código. Es ideal para programadores que buscan llevar su código al siguiente nivel en términos de eficiencia y claridad.

70. Optimiza este código en [lenguaje]: [código]. Reduce complejidad temporal y explica cambios.

71. Explica profiling en [lenguaje, ej. con cProfile en Python]. Dame pasos para analizar un script.

72. Sugiere formas de optimizar [tarea, ej. consultas SQL lentas]. Incluye ejemplos antes/después.

73. Compara algoritmos para [problema, ej. sorting] en términos de rendimiento. Recomienda uno para [escenario]. Formato: Tabla.

74. Optimiza memoria en [lenguaje, ej. garbage collection en Java]. Da tips y un ejemplo.

75. Genera documentación para esta función: [función]. Usa formato docstring y explica parámetros/retorno.

76. Crea un README.md para un proyecto [descripción, ej. app de notas]. Incluye instalación, uso y contribuciones.

77. Revisa este pull request: [descripción_cambios]. Sugiere feedback constructivo en viñetas.

78. Explica cómo usar [herramienta, ej. JSDoc] para documentar código. Da un ejemplo.

79. Lista estándares de codificación para [lenguaje, ej. Google Style Guide para C++]. Resume 10 reglas clave.

Prompts para crecimiento, diseño de software y mejores prácticas

Últimos 21 prompts sobre crecimiento personal, diseño de sistemas, arquitectura y buenas prácticas de desarrollo. Estos van un paso más allá del código puntual: se enfocan en cómo planificar y estructurar un proyecto o aplicar principios de alto nivel para escribir código de calidad. Las preguntas incluyen escenarios de diseño a gran escala, uso de patrones de diseño, elección de tecnologías, mantenimiento de código limpio y documentación.

Familiarizarse con estas ideas desde etapas tempranas te dará perspectiva de buen ingeniero y te ayudará a trabajar en equipo de manera más eficiente.

80. ¿Cómo incorporarías pruebas de seguridad (ej. probar inyecciones SQL, XSS) en el proceso de testing de una aplicación web?

81. Diseña la arquitectura de un sistema de chat en tiempo real capaz de soportar miles de usuarios concurrentes (piensa en componentes: cliente, servidor, base de datos, etc.).

82. ¿Qué patrón de diseño aplicarías para permitir extender las funcionalidades de un módulo sin modificar su código existente (Principio Abierto/Cerrado), y cómo lo implementarías en un caso práctico?

83. Recomienda una pila de tecnologías – frontend, backend y base de datos – para construir una aplicación web de e-commerce básica, explicando brevemente por qué elegirías cada componente.

84. Menciona algunas mejores prácticas para escribir código limpio y mantenible (nombres de variables claros, funciones cortas, DRY, etc.).

85. Tu proyecto necesita documentación. ¿Qué información clave incluirías en un buen archivo README de un repositorio de código?

86. Te toca revisar código escrito por otro desarrollador. ¿En qué aspectos te fijarías para asegurar la calidad (legibilidad, eficiencia, manejo de errores, estilo) y qué le sugerirías mejorar?

87. ¿Cómo abordarías la escalabilidad de una aplicación web si anticipas un crecimiento de usuarios del 1000% el próximo año? (Considera bases de datos, balanceo de carga, caché, etc.).

88. Describe por qué es importante la modularidad en el desarrollo de software y cómo la aplicarías para organizar el código de un proyecto grande.

89. ¿Qué harías para garantizar que tu equipo sigue una guía de estilos de código consistente? (ej. uso de linters, formateadores automáticos, revisiones de código).

90. Si necesitas integrar tu aplicación con un servicio externo (API de terceros), ¿qué consideraciones de diseño y seguridad tendrías en cuenta para hacerlo de forma segura y mantenible?

91. Prepara preguntas de entrevista para [rol, ej. junior developer en Python]. Incluye 5 técnicas y 5 de código con soluciones.

92. Configura un entorno de desarrollo para [stack, ej. MERN]. Lista pasos, herramientas y comandos.

93. Dame tips para debugging en producción vs desarrollo. Incluye herramientas como logs.

94. Explica Agile/Scrum para programadores juniors. Describe roles, ceremonias y cómo aplicarlo en proyectos personales.

95. Sugiere un workflow Git para principiantes: branching, commits, merges. Incluye comandos y diagramas en texto.

96. Lista 10 hábitos diarios para mejorar como programador junior. Explica cada uno brevemente.

97. Recomienda extensiones VS Code para [lenguaje, ej. Python]. Describe qué hace cada una y por qué usarla. Formato: Tabla.

98. Explica seguridad básica en código: [tema, ej. SQL injection]. Da prevención y ejemplos.

99. Crea un plan para refactorizar código legacy: [descripción]. Pasos secuenciales y riesgos.

100. Dame consejos para equilibrar aprendizaje y programación en una carrera junior. Incluye metas semanales realistas.

PROMPTS BÁSICOS

En esta sección se recopilan 50 prompts categorizados, orientados a resolver tareas comunes y habituales desde un enfoque simple.

Comprensión y aprendizaje

Estos 10 prompts están diseñados para acelerar la curva de aprendizaje y desmitificar conceptos complejos.

1. Explicar un concepto técnico

Actúa como un profesor de ciencias de la computación especializado en simplificar temas complejos. Explica el concepto de [concepto] a un desarrollador junior. Comienza con una analogía del mundo real para ilustrar la idea principal. A continuación, proporciona un fragmento de pseudocódigo claro que demuestre su aplicación práctica. Finalmente, enumera en una lista de viñetas 3 ventajas clave y 2 desventajas o contrapartidas de su uso.

2. Comparar dos tecnologías o enfoques

Actúa como un analista tecnológico objetivo. Crea una tabla en formato Markdown que compare [tecnología_1] y [tecnología_2] en base a los siguientes criterios: [criterios] y "casos de uso ideales". Para cada criterio, proporciona una explicación concisa.

3. Traducir un fragmento de código

Actúa como un programador políglota. Traduce el siguiente [código]. Asegúrate de que la traducción sea idiomática y siga las convenciones de estilo del lenguaje de destino. Añade comentarios en el código traducido para explicar cualquier diferencia semántica o sintáctica importante.

4. Resumir documentación técnica

Actúa como un editor técnico. Lee el siguiente texto de documentación y resúmelo en 5 puntos clave (bullet points). El resumen debe centrarse en los aspectos más importantes para un desarrollador que necesita empezar a usar esta tecnología rápidamente.

5. Explicar un fragmento de código complejo

Actúa como un programador senior que realiza una revisión de código. Explica el siguiente fragmento de código línea por línea o bloque por bloque. Describe el propósito general de la función, la lógica de cada parte y cómo interactúan entre sí. Señala cualquier patrón de diseño o algoritmo utilizado.

6. Explicar un mensaje de error

Actúa como un depurador de código experto. Dado el siguiente mensaje de error: [mensaje], explica cuál es la causa raíz más probable de este error. Luego, proporciona una lista de 3 posibles soluciones, ordenadas de la más probable a la menos probable.

7. Identificar el propósito de un script

Analiza el siguiente script y describe su propósito principal en una sola frase. Luego, detalla las acciones que realiza en una lista de pasos secuenciales.

8. Encontrar recursos de aprendizaje

Actúa como un curador de contenido educativo. Recomienda los 3 mejores recursos (artículos, tutoriales en vídeo o cursos) para un desarrollador junior que quiere aprender sobre [tema]. Para cada recurso, indica por qué lo recomiendas.

9. Generar preguntas de entrevista

Actúa como un entrevistador técnico. Genera 5 preguntas de entrevista de nivel junior sobre el tema. Incluye tanto preguntas conceptuales como un pequeño problema de código (en pseudocódigo).

10. Explicar la complejidad algorítmica

Analiza la siguiente función en pseudocódigo y determina su complejidad temporal (Big O notation) en el peor de los casos. Explica tu razonamiento paso a paso, identificando las operaciones dominantes.

Generación de código

Estos 15 prompts ayudan a escribir código de manera más rápida y consistente, incorporando buenas prácticas desde el inicio.

11. Generar una función con requisitos específicos

Actúa como un programador senior que sigue las mejores prácticas de código limpio. Genera una función en [lenguaje] que [descripción]. La función debe cumplir los siguientes requisitos:

1. Nombre de la función: [nombre], parámetros: [parámetros].
2. Debe incluir un manejo de errores robusto para casos límite como [casos_límite].
3. La implementación debe ser eficiente, con una complejidad óptima.
4. Incluye comentarios de documentación completos (ej. docstrings, JSDoc) explicando el propósito de la función, sus parámetros y el valor de retorno.

12. Crear una clase o módulo

Diseña una clase llamada [nombre] en [lenguaje] para gestionar [recurso]. Debe incluir:

1. Un constructor que inicialice [propiedades].
2. Métodos públicos para [funcionalidades].
3. Métodos privados si son necesarios para la lógica interna.
4. Sigue los principios de encapsulación.

13. Generar una expresión regular (Regex)

Actúa como un experto en expresiones regulares. Crea una regex para validar [patrón]. La regex debe ser compatible con [motor/lenguaje]. Proporciona una explicación detallada de cada parte de la regex.

14. Escribir una consulta de base de datos

Actúa como un administrador de bases de datos (DBA). Escribe una consulta SQL para [objetivo]. Las tablas involucradas son (columnas: [COL1, COL2]) y (columnas: [COL3, COL4]). La consulta debe ser optimizada para el rendimiento.

15. Crear un archivo de configuración

Genera un archivo de configuración para un proyecto estándar de [tecnología]. El archivo debe estar bien comentado para explicar las opciones más importantes.

16. Generar datos de prueba (Mocks/Fixtures)

Genera un array de 5 objetos en formato JSON para simular datos de [dominio]. Cada objeto debe tener las siguientes claves:
[CLAVE_1] [tipo], [CLAVE_2] [tipo], [CLAVE_3] [tipo].
Los datos deben parecer realistas.

17. Implementar un algoritmo clásico

Implementa el algoritmo de [algoritmo] en [lenguaje]. La función debe estar bien documentada y incluir un ejemplo de uso.

18. Generar código a partir de comentarios

[lenguaje]

1. Declara una función que acepte una lista de números.
2. Filtra la lista para mantener solo los números pares.
3. Eleva cada número par al cuadrado.
4. Devuelve la suma de los números resultantes.

19. Completar una función

Completa la siguiente función en [lenguaje]:
// Función para convertir una cadena a formato título.
function convertirATitulo(texto) { ... }

20. Crear un endpoint de API REST básico

Genera el código boilerplate en [tecnología] para un endpoint de API REST que responda a una petición GET en la ruta "/api/items". Debe devolver un array de objetos JSON con un código de estado 200.

21. Generar una estructura de datos

Define la estructura de datos (ej. struct, interface, class) en [lenguaje] para representar una [entidad] con los siguientes atributos: [lista_de_atributos].

22. Escribir una función de utilidad común

Crea una función de utilidad reutilizable en [lenguaje] para [propósito].

23. Generar un fragmento de HTML/CSS

Genera el código HTML y CSS para crear un [componente]. El diseño debe ser moderno y accesible.

24. Crear un script de migración de base de datos

Escribe un script de migración de base de datos (en pseudocódigo o para un ORM específico) que añada una nueva columna llamada [nombre] de tipo [tipo] a la tabla [tabla].

25. Generar una plantilla de correo electrónico

Crea una plantilla HTML para un correo electrónico transaccional de [contexto]. Debe ser responsive e incluir placeholders para [variables].

Calidad y mantenimiento del código

Estos 15 prompts se centran en mejorar la calidad, legibilidad y robustez del código existente.

26. Realizar una revisión de código (Code Review)

Actúa como un revisor de código senior, meticuloso y constructivo. Revisa el siguiente fragmento de código y proporciona feedback. Tu análisis debe cubrir los siguientes aspectos:

1. Calidad y adherencia a las mejores prácticas (ej. SOLID, DRY).
2. Potenciales bugs, errores lógicos o casos límite no controlados.
3. Oportunidades de optimización de rendimiento.
4. Legibilidad, claridad y mantenibilidad del código.
5. Posibles vulnerabilidades de seguridad (ej. inyección SQL, XSS).
   Para cada punto, explica el problema y sugiere una versión corregida del código.

27. Refactorizar código para mejorar legibilidad

Refactoriza el siguiente código para mejorar su legibilidad y mantenibilidad sin cambiar su funcionalidad. Aplica principios de código limpio, como usar nombres de variables descriptivos, extraer métodos y simplificar condicionales complejos.

28. Encontrar bugs en una función

Analiza la siguiente función en busca de posibles bugs o errores lógicos. Describe cualquier problema que encuentres y explica por qué es un bug. Proporciona una versión corregida del código.

29. Optimizar una función lenta

El siguiente fragmento de código es lento. Identifica los cuellos de botella de rendimiento y sugiere optimizaciones. Explica por qué tus sugerencias mejorarían el rendimiento y proporciona el código optimizado.

30. Generar tests unitarios

Actúa como un ingeniero de QA. Genera un conjunto de tests unitarios para la siguiente función usando el framework de testing. Los tests deben cubrir:

1. El "happy path" o caso de uso normal.
2. Casos límite (ej. arrays vacíos, valores cero, strings vacíos).
3. Entradas inválidas y manejo de errores.

31. Generar tests de integración

Describe un plan de tests de integración para dos módulos que interactúan: [descripción_acciones].
Escribe 2 o 3 casos de prueba en pseudocódigo que verifiquen su correcta interacción.

32. Añadir documentación a código existente

Genera comentarios de documentación en formato para la siguiente función. La documentación debe explicar qué hace la función, describir cada uno de sus parámetros y especificar qué devuelve.

33. Mejorar el manejo de errores

Revisa el siguiente código y mejora su manejo de errores. Reemplaza los bloques try-catch genéricos con un manejo de excepciones más específico y proporciona mensajes de error más informativos.

34. Identificar vulnerabilidades de seguridad

Actúa como un experto en ciberseguridad. Analiza el siguiente fragmento de código en busca de vulnerabilidades de seguridad comunes (ej. Inyección de SQL, Cross-Site Scripting (XSS), Path Traversal). Para cada vulnerabilidad encontrada, explica el riesgo y cómo mitigarlo.

35. Convertir código a un estilo funcional

Reescribe la siguiente función, que utiliza un estilo imperativo con bucles y mutación de estado, para que siga un paradigma de programación funcional. Utiliza funciones de orden superior como map, filter y reduce si es aplicable.

36. Aplicar un patrón de diseño

Refactoriza el siguiente código para aplicar el patrón de diseño. Explica por qué este patrón es adecuado aquí y cómo mejora la estructura del código.

37. Verificar guías de estilo

Revisa el siguiente código y señala cualquier desviación de la guía de estilo.

38. Simplificar condicionales complejos

La siguiente estructura de if-else anidada es demasiado compleja. Simplifícala usando técnicas como guard clauses, polimorfismo o tablas de búsqueda (lookup tables).

39. Identificar código duplicado (Principio DRY)

Analiza los siguientes dos fragmentos de código. Identifica la lógica duplicada y extráela a una función reutilizable.

40. Añadir logging significativo

Añade sentencias de logging al siguiente código. Los logs deben ser informativos y registrar eventos clave, advertencias y errores para facilitar la depuración en producción.

Planificación y arquitectura

Estos 5 prompts ayudan a estructurar el pensamiento y a tomar decisiones de diseño antes de escribir la primera línea de código.

41. Proponer un patrón de diseño adecuado

Actúa como un arquitecto de software experimentado. Estoy enfrentando el siguiente problema de diseño: [problema].
Sugiere 2 o 3 patrones de diseño que podrían ser una solución adecuada. Para cada patrón sugerido, proporciona:

1. Una breve explicación de qué es.
2. Por qué es una buena opción para este problema específico.
3. Una lista de sus pros y contras en este contexto.

42. Diseñar la estructura de datos para un objeto

Diseña la estructura de datos en formato JSON para representar [objeto]. Debe incluir campos anidados, arrays y diferentes tipos de datos. Considera todos los estados posibles del objeto.

43. Diseñar la arquitectura de un microservicio

Esboza una arquitectura de alto nivel para un microservicio cuya única responsabilidad es. El esquema debe incluir:

1. Los endpoints de la API que expondrá (con métodos HTTP y rutas).
2. Las dependencias que tendrá (ej. base de datos, otros servicios).
3. La tecnología sugerida para la pila (stack).

44. Planificar los pasos para implementar una feature

Actúa como un gestor de proyectos técnicos. Desglosa la implementación de la siguiente funcionalidad: [funcionalidad] en una lista de tareas técnicas más pequeñas y manejables. Ordena las tareas por dependencia.

45. Elegir una estructura de proyecto

Sugiere una estructura de directorios estándar y escalable para un nuevo proyecto de [descripción_proyecto]. Explica el propósito de cada directorio principal.

Herramientas y línea de comandos

Estos 5 prompts desmitifican herramientas poderosas del ecosistema de desarrollo.

46. Generar un comando de Git complejo

Actúa como un experto en Git y control de versiones. Necesito un único comando de Git para lograr el siguiente objetivo: [descripción]. Proporciona el comando y luego explica cada parte del mismo para que pueda entender cómo funciona.

47. Escribir un script de shell (Bash/PowerShell)

Escribe un script en [Bash o PowerShell] que automatice la siguiente tarea: [descripción].

48. Generar un comando awk o sed

Proporciona un comando de una línea usando [awk o sed] para procesar un archivo de texto [acción].

49. Crear una pipeline de CI/CD simple

Genera un archivo de configuración básico para [plataforma_CI] que se active en cada push a la rama "main". El pipeline debe realizar los siguientes pasos:

1. Instalar dependencias.
2. Ejecutar los tests.
3. Construir el proyecto.

50. Explicar un comando de CLI

Desglosa y explica el siguiente comando de la línea de comandos: [comando]. Describe qué hace el comando principal y qué significa cada una de sus flags o argumentos.

PROMPTS AVANZADOS

En esta última sección encontrarás recopilados 50 prompts categorizados, orientados a resolver tareas comunes y habituales desde un enfoque más avanzado y detallado.

Debugging

1. Debugging sistemático

Eres un experto en depuración. Para este código con error, proporciona:

1. Calificación del código.
2. Lista de problemas identificados.
3. Pasos específicos para depurar sin cambiar funcionalidad.
4. Explicación de por qué ocurre el error.

Código: [código]
Error: [error]

2. Debugging con contexto específico

Actúa como desarrollador senior con 20+ años de experiencia. Analiza:

- Lenguaje: [lenguaje]
- Framework: [framework]
- Error: "[error]" en [archivo] línea [n]
- Código: [código]

Proporciona: diagnóstico del problema raíz, solución paso a paso, prevención futura, mejores prácticas relacionadas.

3. Debugging reactivo

Detecta y corrige errores off-by-one o condiciones límite en [archivo].
Analiza el siguiente error y sugiere correcciones: "[error]" en [archivo] línea [n].

4. Debugging de rendimiento

Identifica cuellos de botella de rendimiento en este código:
[código]
Proporciona análisis de complejidad computacional, uso de memoria y optimizaciones específicas.

5. Debugging de dependencias

Analiza conflictos de dependencias en este proyecto [tecnología]:
[ej. package.json/requirements.txt/etc]
Sugiere resolución de versiones compatibles y mejores prácticas.

6. Debugging de lógica de negocio

Este código debería [comportamiento_esperado] pero está haciendo [comportamiento_actual].
Código: [código]
Encuentra la discrepancia lógica y propón corrección.

7. Debugging de concurrencia

Revisa este código en busca de problemas de concurrencia o condiciones de carrera:
[código]
Sugiere soluciones thread-safe apropiadas.

8. Debugging con logs estratégicos

Añade declaraciones de log estratégicas para rastrear el flujo en este código:
[código]
Incluye niveles apropiados de logging y información contextual útil.

Refactoring y optimización

9. Refactoring sistemático multi-step

Refactoriza código siguiendo un proceso de múltiples pasos:

1. Reescribir a estándares [ej. ES6] modernos.
2. Identificar problemas lógicos o de seguridad.
3. Revisar recomendaciones y errores.

4. Reescribir función basándose en revisión final.
5. Crear dos tests: uno que pase y otro que falle.

Código: [código]

10. Optimización de rendimiento

Actúa como arquitecto de software especializado en optimización.
Código: [código]
Optimiza para: reducir complejidad computacional, mejorar uso de memoria, eliminar cuellos de botella, mantener legibilidad.
Proporciona: análisis actual, código optimizado, explicación de mejoras, métricas esperadas.

11. Refactoring modular

Refactoriza [módulo/archivo] para mejorar legibilidad, modularidad y añadir comentarios explicativos.
Divide funciones grandes en funciones más pequeñas de responsabilidad única.

12. Modernización de código

Migra APIs deprecadas en [archivo] a las nuevas APIs recomendadas.
Convierte funciones basadas en callbacks a async/await.

13. Eliminación de código muerto

Identifica y elimina código muerto o no utilizado en [carpeta/archivo].
Proporciona análisis de impacto antes de eliminación.

14. Refactoring semántico

Renombra variables y funciones en [archivo] para mejorar claridad semántica.
Sigue convenciones de nomenclatura establecidas para [lenguaje].

15. Refactoring de patrones

Convierte este código al patrón [patrón_específico]:
[código]
Explica beneficios del patrón aplicado.

16. Refactoring de estructura de datos

Optimiza la estructura de datos en este código para mejor rendimiento:
[código]
Considera uso de memoria, velocidad de acceso y operaciones frecuentes.

Testing y QA

17. Generación de tests

Actúa como ingeniero de testing senior con experiencia en [framework_testing].
Función: [función]
Genera suite completa: tests unitarios básicos, casos límite, manejo de errores, integración, rendimiento.
Para cada test explica: qué prueba, por qué es importante, configuración necesaria.

18. Testing de cobertura

Basado en requisitos: [lista_requisitos] y casos actuales: [casos_existentes],
¿Qué gaps existen en cobertura?
Sugiere tests adicionales: casos de error, integraciones, regresión, performance, seguridad.
Prioriza por riesgo e impacto.

19. Estrategia de automatización de test

Genera tests unitarios para [función] en [archivo] usando [framework_testing].
Incluye setup/teardown apropiado y mocks necesarios.

20. Tests de integración

Crea tests de integración para [feature/módulo].
Incluye validación de contratos entre servicios.

21. Tests end-to-end

Escribe tests E2E para [user_journey/flujo].
Incluye happy path y escenarios de error principales.

22. Mocking estratégico

Mockea llamadas a servicios externos para aislamiento de tests en [archivo].
Proporciona ejemplos de respuestas realistas.

23. Tests de casos límite

Añade tests de casos límite y edge cases para [función].
Incluye inputs inválidos, valores nulos, límites de memoria.

24. Property-based testing

Genera tests basados en propiedades [fuzz tests] para inputs de [función].
Define propiedades invariantes que deben mantenerse.

Documentación

25. Documentación completa de módulos

Añade documentación para [módulo], incluyendo:

- Propósito y responsabilidades.
- Ejemplos de uso.
- Diagramas relevantes.
- Documentación para otros desarrolladores.
- Consideraciones de actualización futura.

Código: [código]

26. Documentación para no-técnicos

Explica este código para una persona no técnica usando Markdown formateado.
Organízalo por secciones: ¿qué hace?, componentes principales, flujo paso a paso, resultados esperados.

Código: [código]

27. Auto-generación de documentación API

Autogenera la documentación de la API para [archivo/módulo].
Incluye endpoints, métodos, parámetros, respuestas, códigos de error.

28. Docstrings inline

Añade docstrings inline a todos los métodos públicos en [clase] siguiendo el estilo de Google.
Incluye parámetros, retorno, raises, ejemplos.

29. README

Genera README.md con instrucciones de instalación, uso y contribución para este proyecto.
Incluye badges, screenshots, roadmap.

30. Documentación de arquitectura

Escribe resumen de arquitectura de alto nivel para este repo.
Incluye diagramas C4, decisiones arquitectónicas, trade-offs.

Aprendizaje y explicación

31. Explicación línea por línea

Contexto: Desarrollador [backend/frontend] necesita entender funciones.
Tecnologías: [tecnologías]
Explica línea por línea: qué hace, por qué se usa ese patrón, alternativas, mejores prácticas.

Código: [código]

32. Proyectos de práctica

Necesito practicar [concepto/tecnología].
Dame proyecto nivel [principiante/intermedio/avanzado] con:

- Objetivos específicos.
- Lista de características.
- Tecnologías sugeridas.
- Estructura de archivos.
- Hitos de progreso.
- Recursos adicionales.

33. Conceptos con ejemplos

Explica [concepto_programación] como instructor para principiantes.
Incluye: explicación breve, ejemplos simples, casos de uso reales, errores comunes, ejercicios progresivos.
Usar [lenguaje] para ejemplos.

34. Analogías técnicas

Explica [concepto_técnico_complejo] usando analogías del mundo real.
Haz paralelos entre el concepto técnico y situaciones cotidianas comprensibles.

35. Comparación de enfoques

Compara diferentes enfoques para [problema_técnico]:
Para cada enfoque incluye: pros, contras, casos de uso ideales, complejidad de implementación.

36. Tutoriales paso a paso

Crear tutorial paso a paso para implementar [funcionalidad].
Cada paso debe ser verificable independientemente con outputs esperados.

Revisión de código

37. Revisión empresarial

Actúa como desarrollador senior con experiencia empresarial.
Analiza para: Principios SOLID, legibilidad, mantenibilidad, documentación, nomenclatura, patrones de diseño, APIs REST, optimización de datos, seguridad OWASP, validación de inputs, headers de seguridad, optimización de consultas, caching, microservicios.

Código: [código]

38. Revisión específica por aspecto

Como desarrollador senior, revisa enfocándote en:

- Adherencia a estándares del equipo.
- Problemas potenciales de rendimiento.
- Oportunidades de refactoring.

- [aspecto_específico]
  Proporciona retroalimentación constructiva y sugerencias específicas.

Código: [código]

39. Revisión de seguridad

Escanea proyecto en busca de vulnerabilidades comunes.
Verifica: inyección SQL, XSS, CSRF, algoritmos criptográficos desactualizados, validación de inputs, autorización de rutas.

40. Revisión de mejores prácticas

Evalúa adherencia a mejores prácticas de [tecnología]:
Código: [código]
Proporciona score y recomendaciones específicas por categoría.

41. Revisión de arquitectura

Analiza decisiones arquitectónicas en este código:
¿Se siguen patrones apropiados? ¿Hay violaciones de principios SOLID? ¿La separación de concerns es clara?

42. Revisión de rendimiento

Audita este código para problemas de rendimiento:
Identifica: operaciones O(n²), memory leaks, consultas N+1, blocking operations innecesarias.

43. Revisión de mantenibilidad

Evalúa mantenibilidad de este código:
Considera: complejidad ciclomática, acoplamiento, cohesión, naming, documentación.

44. Revisión de testing

Revisa la calidad de estos tests:
¿Cubren casos importantes? ¿Son mantenibles? ¿Siguen AAA pattern? ¿Son determinísticos?

Generación y arquitectura

45. Generación con contexto específico

Contexto: Software para [dominio]
Tecnologías: [tecnologías]
Descripción: [funcionalidad_requerida]
Crear función con: validación de inputs, manejo de errores robusto, comentarios de lógica de negocio, tests básicos.

46. Especificaciones técnicas

Eres ingeniero de software de clase mundial. Redacta especificación técnica para: [descripción_sistema]
Incluye: resumen ejecutivo, arquitectura, componentes, APIs, datos, seguridad, implementación, riesgos.

47. Patrones arquitectónicos

Diseñamos sistema distribuido con objetivos:

- [objetivo_1: ej. Escalabilidad horizontal]
- [objetivo_2: ej. Procesamiento asíncrono]
- [objetivo_3: ej. Alta disponibilidad]
  Sugiere patrones que cumplan objetivos. Lista riesgos y anti-patrones. Recomienda alternativas más simples si funcionan.

48. Diagramas de arquitectura

Escribe código Mermaid para diagrama de arquitectura de: [solución]
Incluye: componentes principales, flujos de datos, tecnologías, integraciones externas.
Usa formato C4 si es apropiado.

49. Boilerplate inteligente

Crear boilerplate de [tecnología] que:

- Tome una variable de tipo [tipo]
- Valide [condiciones]
- Obtenga [datos] desde [fuente]

- Retorne [formato]
  Incluya configuración básica y estructura de carpetas recomendada.

50. Análisis de trade-offs

Para este diseño arquitectónico: [descripción]
Proporciona: análisis de fortalezas, identificación de riesgos, trade-offs, alternativas, plan de evolución.
Considera: rendimiento, escalabilidad, mantenibilidad, coste operacional.

10 CONSEJOS PARA CREAR PROMPTS EFECTIVOS

Dominar la ingeniería de prompts es un proceso continuo de aprendizaje y refinamiento. Los siguientes diez principios reflejan las mejores prácticas y fomentan una mentalidad que transforma al desarrollador de un simple "usuario" de la IA a un "ingeniero" de la comunicación con ella. Este cambio de paradigma implica un enfoque proactivo y sistemático: diseñar interacciones, probar hipótesis con diferentes prompts, analizar los resultados y optimizar el proceso de forma iterativa.

1. Define tu objetivo primero

Antes de escribir una sola palabra del prompt, es fundamental articular con precisión el resultado deseado. Formular el objetivo en una frase clara y concisa obliga a la mente a pasar de la ambigüedad a la especificidad. La claridad mental es un prerrequisito indispensable para la claridad del prompt.

> Ten en cuenta que cada modelo es especialista en tareas concretas y debes utilizarlo siguiendo las recomendaciones oficiales.

2. La especificidad vence a la ambigüedad

El error más común y perjudicial es la vaguedad. Un prompt debe incluir todos los detalles relevantes que puedan influir en la respuesta. En lugar de instrucciones genéricas como "optimiza este código", se deben dar directivas específicas como "refactoriza esta función para reducir su complejidad ciclomática y eliminar variables globales".

3. Divide y vencerás

Intentar resolver un problema complejo con un único prompt monolítico suele llevar a resultados mediocres o incompletos. Al igual que en el desarrollo de software, los problemas grandes deben descomponerse en subtareas más pequeñas y manejables. Abordar cada subtarea con un prompt enfocado y luego combinar los resultados produce una solución final mucho más robusta y precisa.

4. Itera y refina (el prompt es una conversación)

El primer prompt rara vez es el definitivo. La interacción con un LLM debe ser tratada como un diálogo, no como una transacción única. La respuesta inicial del modelo proporciona información valiosa que puede ser utilizada para refinar y mejorar las preguntas de seguimiento. Este proceso iterativo de ajuste es clave para converger hacia el resultado óptimo.

5. Usa instrucciones positivas

Los modelos de lenguaje responden mejor cuando se les guía hacia lo que deben hacer, en lugar de simplemente listar lo que deben evitar. Una instrucción positiva es más directa y menos propensa a interpretaciones erróneas. Por ejemplo, "Escribe código conciso y legible utilizando nombres de variables explícitos" es significativamente más efectivo que "No escribas código complicado ni uses nombres de variables cortos".

6. Pide que "piense paso a paso"

Para tareas que requieren razonamiento lógico, como la depuración de un algoritmo o la planificación de una arquitectura, añadir la simple frase "Pensemos paso a paso" al final del prompt puede mejorar drásticamente la calidad de la respuesta. Esta instrucción activa una técnica conocida como Chain-of-Thought (CoT), que incita al modelo a externalizar su proceso de razonamiento, lo que a menudo conduce a conclusiones más correctas y coherentes.

7. Verifica siempre, no confíes ciegamente

El código y la información generados por la IA deben ser tratados siempre como una sugerencia de un colega extremadamente talentoso pero ocasionalmente falible. Los LLMs pueden "alucinar", es decir, generar información que parece plausible pero es factualmente incorrecta o introduce errores sutiles en el código. La responsabilidad final de entender,

probar y validar la corrección y seguridad de cualquier resultado recae siempre en el desarrollador.

8. Crea tu biblioteca personal de prompts

Cuando un prompt produce un resultado excepcionalmente bueno, debe ser guardado y categorizado. Con el tiempo, esta práctica permite construir una biblioteca personal de plantillas de alta calidad. Esta caja de herramientas personalizada se convierte en un activo invaluable que acelera drásticamente el trabajo diario y garantiza la consistencia en los resultados.

9. Entiende las limitaciones del modelo

Es crucial ser consciente de las limitaciones inherentes de los LLMs. No tienen acceso a información en tiempo real (a menos que estén conectados a herramientas externas), su conocimiento está limitado a la fecha de corte de sus datos de entrenamiento, y pueden reflejar los sesgos presentes en dichos datos. No "entienden" el código en el sentido humano; reconocen y manipulan patrones. Utilizar la IA como una herramienta para aumentar la inteligencia humana, no para reemplazar el juicio crítico, es la clave para un uso responsable y efectivo.

10. Experimenta constantemente

La ingeniería de prompts es, en gran medida, una disciplina empírica. La mejor manera de desarrollar una intuición sobre lo que funciona es a través de la experimentación constante. Probar diferentes formulaciones, asignar distintas personas, variar los formatos de salida y observar cómo pequeñas modificaciones pueden alterar significativamente la respuesta es fundamental. La práctica deliberada y la curiosidad son los verdaderos motores que convierten a un principiante en un experto.