Otro tutorial completo de Claude Code en español.

> Una guía práctica para usar Claude Code de forma efectiva en proyectos reales de software.

Piensa primero, escribe después

La mayoría de la gente asume que con Claude Code lo primero es escribir. Pero el paso más importante ocurre antes de tocar el teclado: **pensar**.

El modo plan (`Shift + Tab` dos veces) produce resultados consistentemente mejores que improvisar directamente. Son 5 minutos de inversión que ahorran horas de depuración.

Esto aplica a todo: antes de pedir una funcionalidad, piensa en la arquitectura. Antes de refactorizar, piensa en el estado final. Antes de depurar, piensa en lo que realmente sabes del problema.

Si no tienes experiencia suficiente para planificar solo, usa otro LLM como interlocutor previo. Describe lo que quieres construir, pide opciones de diseño y lleguen a una solución juntos. El diálogo debe ser bidireccional: ambos haciendo preguntas.

La arquitectura importa más que el código

Un prompt vago como "constrúyeme un sistema de autenticación" deja demasiado margen de interpretación. Compara con:

> "Construye autenticación email/contraseña usando el modelo de Usuario existente, almacena sesiones en Redis con expiración de 24h y añade middleware que proteja las rutas bajo `/api/protected`."

La diferencia entre ambos es la diferencia entre resultados predecibles y código que necesitará reescribirse.

CLAUDE.md: tu archivo más importante

CLAUDE.md es lo primero que Claude lee al iniciar una sesión. Cada instrucción ahí moldea cómo aborda tu proyecto. Es material de inducción que se ejecuta antes de cada conversación.

Principios para un buen CLAUDE.md

Mantenlo corto. Claude puede seguir de forma fiable entre 150-200 instrucciones simultáneas, y el prompt de sistema ya consume unas 50. Cada instrucción compite por atención. Si tu CLAUDE.md es una novela, Claude empezará a ignorar cosas al azar.

Hazlo específico para tu proyecto. No expliques qué es una carpeta de componentes. Claude ya lo sabe. Documenta las cosas que son propias de tu proyecto: comandos de bash relevantes, convenciones no estándar, particularidades de tu flujo de trabajo.

Dile el porqué, no solo el qué. "Usa el modo estricto de TypeScript" está bien. "Usa el modo estricto de TypeScript porque hemos tenido errores en producción por tipos `any` implícitos" es mejor. El porqué le da contexto para tomar decisiones que no anticipaste.

Actualízalo constantemente. Pulsa `#` mientras trabajas y Claude añadirá instrucciones automáticamente. Si te encuentras corrigiendo a Claude sobre lo mismo dos veces, eso debería estar en el archivo. Con el tiempo, CLAUDE.md se convierte en un documento vivo.

> **Referencia rápida:** Un mal CLAUDE.md parece documentación para un nuevo empleado. Un buen CLAUDE.md se parece a las notas que te dejarías a ti mismo si supieras que mañana vas a tener amnesia.

Gestión de la ventana de contexto

Opus 4.5 tiene 200,000 tokens de ventana de contexto. Pero la calidad del output empieza a desgastarse bastante antes del límite — según la experiencia de muchos usuarios, entre el 20-40%

de uso ya puede notarse degradación, aunque varía según el tipo de tarea y la interfaz utilizada. (Nota: esto es una observación empírica ampliamente reportada, no un dato oficial de Anthropic.)

Si has experimentado que Claude compacta el contexto (`/compact`) y después sigue dando resultados mediocres, es por esto: el modelo ya estaba degradado antes de la compactación, y comprimir no restaura la calidad.

Estrategias que funcionan

Acota tus conversaciones. Una conversación por funcionalidad o tarea. No mezcles construir el sistema de auth con refactorizar la capa de base de datos. Los contextos se contaminarán mutuamente.

Usa memoria externa. Para trabajo complejo, haz que Claude escriba planes y progreso en archivos reales (SCRATCHPAD, plan.md o similar). Estos persisten entre sesiones. Cuando retomes, Claude puede leer el archivo y continuar donde lo dejó. Si usas un sistema jerárquico de archivos, mantener estos documentos en la raíz los hace accesibles para todas las tareas.

El truco "copiar-pegar y reiniciar". Cuando el contexto se infla:

1. Copia lo importante de la terminal
2. Ejecuta `/compact` para obtener un resumen
3. Ejecuta `/clear` para limpiar el contexto completamente
4. Pega solo lo que importa

Contexto fresco con la información crítica preservada. Es significativamente mejor que seguir trabajando con un contexto degradado.

Saber cuándo limpiar. Si una conversación se ha descarrilado, `/clear` y empezar de cero es casi siempre mejor que intentar reconducir la confusión. Claude seguirá teniendo tu CLAUDE.md, así que el contexto del proyecto no se pierde.

> **Modelo mental clave:** Claude no tiene estado. Cada conversación comienza desde cero, excepto por lo que le das explícitamente. Planifica en consecuencia.

Prompts: la habilidad más infravalorada

El prompting no es un arte místico. Es comunicación. Y como toda comunicación, la claridad produce mejores resultados que la vaguedad.

Lo que realmente marca la diferencia

Sé específico sobre lo que quieres. La especificidad reduce el margen de interpretación. Incluye restricciones, tecnologías concretas, y el comportamiento esperado.

Dile qué NO hacer. Claude (especialmente los modelos más capaces) tiende a sobreingenierizar: archivos extra, abstracciones innecesarias, flexibilidad que no pediste. Si quieres algo minimalista, dilo explícitamente: "Mantén esto simple. No añadas abstracciones que no pedí. Un solo archivo si es posible."

Dale contexto sobre las restricciones. "Necesitamos que esto sea rápido porque se ejecuta en cada solicitud" cambia radicalmente cómo Claude aborda el problema. "Este es un prototipo que vamos a desechar" cambia qué tradeoffs tienen sentido. Claude no puede adivinar restricciones que no mencionas.

Revisa siempre el output. La IA acelera el desarrollo, no reemplaza el criterio del ingeniero. Claude comete errores. La capacidad de reconocer esos errores — código innecesariamente complejo, abstracciones prematuras, edge cases ignorados — es lo que separa el uso productivo del uso problemático.

La calidad del output depende del input

Cuando los resultados son malos con un modelo capaz, el primer lugar donde buscar mejoras es en el propio prompt: ¿fue suficientemente específico? ¿Proporcionaste las restricciones necesarias? ¿Comunicaste lo que realmente querías?

Mejorar en prompting significa mejorar en tres áreas simultáneamente: cómo redactas las solicitudes (específico > vago, restricciones > abierto, ejemplos > descripciones), cómo estructuras las tareas (dividir en pasos, acordar arquitectura antes de implementar, iterar sobre resultados), y cómo proporcionas contexto (qué necesita saber Claude, qué suposiciones haces que Claude no puede ver).

Cuándo usar cada modelo

Sonnet: Más rápido y económico. Ideal para tareas de ejecución donde el camino está claro: escribir boilerplate, refactorizar basándose en un plan definido, implementar funcionalidades con decisiones arquitectónicas ya tomadas.

Opus: Más lento y costoso. Mejor para razonamiento complejo, planificación y tareas que requieren pensar en profundidad sobre tradeoffs.

Flujo de trabajo recomendado: Usa Opus para planificar y tomar decisiones arquitectónicas, luego cambia a Sonnet (`Shift+Tab`) para la implementación. Tu CLAUDE.md asegura que ambos modelos operen bajo las mismas restricciones, haciendo el traspaso limpio.

MCP, herramientas y configuración

Claude Code tiene un ecosistema extenso: servidores MCP, hooks, comandos slash personalizados, configuraciones de `settings.json`, skills, plugins. No necesitas todos, pero vale la pena experimentar de forma deliberada para encontrar los que encajan en tu flujo de trabajo.

MCP (Model Context Protocol)

Permite que Claude se conecte a servicios externos: Slack, GitHub, bases de datos, APIs. Si te encuentras copiando información constantemente hacia Claude, probablemente exista un servidor MCP que automatice ese paso. Hay múltiples marketplaces disponibles, y si no existe el que necesitas, puedes crear tu propio servidor MCP — al final es solo una forma de obtener datos estructurados.

Hooks

Ejecutan código automáticamente antes o después de que Claude realice cambios. Algunos usos prácticos: ejecutar Prettier en cada archivo que Claude toca, comprobar tipos después de cada edición, ejecutar linters periódicamente. Los hooks detectan problemas inmediatamente en lugar de dejar que se acumulen como deuda técnica.

Comandos slash personalizados

Prompts que usas repetidamente, empaquetados como comandos. Crea una carpeta `.claude/commands`, añade archivos markdown con tus prompts, y ejecútalos con `/nombre-del-comando`. Ideal para tareas recurrentes: depurar, revisar, desplegar.

Cuando Claude se queda atascado

A veces Claude entra en bucle: intenta lo mismo, falla, lo vuelve a intentar. O implementa con total confianza algo incorrecto. El instinto es seguir presionando con más instrucciones y correcciones, pero generalmente el mejor movimiento es cambiar de enfoque.

Limpia la conversación. El contexto acumulado puede ser el problema. `/clear` te da un punto de partida limpio.

Simplifica la tarea. Si Claude tiene dificultades con algo complejo, divídelo en partes más pequeñas y haz que cada parte funcione antes de combinarlas. Si esto sucede con frecuencia, probablemente tu planificación previa es insuficiente.

Muestra en lugar de decir. Si Claude sigue entendiendo mal, escribe tú mismo un ejemplo mínimo: "Así debería ser el resultado. Ahora aplica este patrón al resto". Claude es muy efectivo siguiendo ejemplos concretos.

Reformula el problema. A veces la forma en que planteaste la tarea no encaja con cómo piensa Claude. Cambiar el enfoque — "implementa esto como una máquina de estados" frente a "maneja estas transiciones" — puede desbloquear el progreso.

> **La meta-habilidad:** Reconoce cuándo estás en un bucle lo antes posible. Si has explicado lo mismo tres veces sin éxito, explicar más no ayudará. Cambia algo.

Construye sistemas, no tareas puntuales

El flag `-p` (modo headless) ejecuta un prompt y devuelve el resultado sin interfaz interactiva. Esto permite automatización real: scripts, pipes con otras herramientas, integración en flujos de trabajo automatizados.

Casos de uso en producción: revisiones automáticas de PR, respuestas a tickets de soporte, actualización de documentación. Todo registrado, auditable y mejorando iterativamente.

El ciclo de mejora funciona así: Claude comete un error → revisas los logs → mejoras el CLAUDE.md o las herramientas → Claude mejora la próxima vez. Este ciclo se acumula. Después de meses de iteración, los sistemas construidos así son significativamente mejores que al inicio, usando los mismos modelos pero mejor configurados.

Resumen

1. **Piensa antes de escribir.** El modo plan produce resultados dramáticamente mejores que improvisar.

2. **CLAUDE.md es tu mayor punto de apalancamiento.** Corto, específico, con el porqué, actualizado constantemente.
3. **El contexto se degrada antes del límite.** Usa memoria externa, acota conversaciones, y no temas limpiar y reiniciar.
4. **La arquitectura determina el resultado.** No puedes saltarte la planificación.
5. **El output proviene del input.** Si los resultados son malos, mejora los prompts.
6. **Experimenta con herramientas.** MCP, hooks, comandos slash. Encuentra lo que encaja en tu flujo.
7. **Cuando te atasques, cambia de enfoque.** No entres en bucle. Limpia, simplifica, muestra, reformula.
8. **Construye sistemas, no interacciones puntuales.** Modo headless, automatización, mejora iterativa.

Revision #1
Created 2026-02-15 06:17:09 UTC by Abkrim
Updated 2026-02-15 06:18:17 UTC by Abkrim