Chuleta de Claude Code: comandos, atajos y skills

Todo lo que se escribe en Claude Code, en una sola página: los comandos de sesión (`/clear`, `/compact`, `/init`…), los atajos de teclado y los tres modos que se ciclan con `shift+tab`, qué hace cada fichero de `.claude/`, y cómo se escribe una skill que se dispare sola. Verificado contra Claude Code 2.1.222.

⬇️ Descargar en PDF (gratis, sin registro)

Comandos de sesión

Se escriben dentro de una sesión de Claude Code, empezando por `/`.

Qué escribes	Qué hace	Cuándo usarlo
`/init`	Lee tu proyecto y redacta un `CLAUDE.md` inicial.	El primer día en un repo que ya existe. Luego recorta sin piedad: borrar es más fácil que escribir.
`/clear`	Vacía el contexto y empieza de cero.	Al cambiar de tarea. Arrastrar el contexto de lo anterior es lo que hace que Claude se despiste.
`/compact`	Resume la conversación para liberar contexto sin perder el hilo.	Sesión larga en la que sigues con lo mismo y notas que se está quedando sin sitio.
`/context`	Enseña en qué se está gastando el contexto.	Cuando quieres saber qué te está comiendo la ventana antes de decidir entre compactar o limpiar.
`/resume`	Retoma una conversación anterior.	Al volver a un trabajo de ayer sin querer explicarlo todo otra vez.
`/rename`	Pone nombre a la conversación actual.	Justo al empezar algo largo: es lo que hará que lo encuentres luego en `/resume`.
`/memory`	Ver y gestionar lo que Claude recuerda entre sesiones.	Cuando repite un error que ya corregiste, o para revisar qué se ha guardado.
`/config`	Ajustes de la sesión, incluido el modo de permisos por defecto.	Para dejar Plan Mode como modo de entrada y no tener que activarlo cada vez.
`/permissions`	Pre-aprueba o deniega herramientas y comandos.	Cuando te cansas de confirmar el mismo `git status` veinte veces al día.
`/model`	Cambia el modelo de la sesión.	Modelo potente para diseñar y arrancar; uno más rápido para retoques e iteración.
`/agents`	Gestiona los subagentes del proyecto.	Para crear o revisar el equipo: investigador, revisor, arquitecto.
`/hooks`	Configura los disparadores automáticos.	Al automatizar comprobaciones que no deben gastar tokens (formato, palabras prohibidas, tests).
`/mcp`	Gestiona los servidores MCP conectados.	Para comprobar si Notion, GitHub o Airtable están realmente conectados antes de pedir nada.

Qué escribes	Qué hace	Cuándo usarlo
`/plugin`	Instala y administra plugins.	Para traer flujos de trabajo enteros ya empaquetados en vez de reinventarlos.
`/cost` · `/usage`	Lo que llevas gastado y tu consumo.	Antes de lanzar una tarea larga y desatendida.
`/doctor`	Diagnostica la instalación.	Primera parada cuando algo va raro y no sabes si es tu proyecto o tu instalación.

Atajos de teclado y modos

Qué pulsas	Qué hace	Cuándo usarlo
`shift+tab`	Cicla entre los tres modos: manual → auto-aceptar ediciones → Plan Mode.	El atajo más rentable de todos. Dos pulsaciones desde manual te dejan en Plan Mode.
`Ctrl+C`	Para lo que esté haciendo.	En cuanto veas que ha cogido el camino equivocado. No esperes a que termine.
Doble `Esc`	Rebobina la conversación (y el código) a un punto anterior.	Cuando las últimas tres respuestas han empeorado las cosas: vuelve atrás en vez de seguir parcheando.
`Ctrl+D`	Cierra la sesión.	Al terminar. Lo hablado se recupera después con `/resume`.
`@`	Referencia un fichero o carpeta por su ruta.	Siempre que hables de un fichero concreto: es mucho más fiable que describirlo con palabras.
`!`	Modo shell: ejecuta el comando directamente.	Para un `git status` o un `ls` rápido sin gastar una respuesta del modelo.
`Ctrl+V`	Pega una imagen del portapapeles.	Para pasarle una captura de la web que quieres imitar o del error que ves. En Mac es `control+v`, **no** `cmd+v`.
`Enter` mientras trabaja	Encola otro mensaje sin interrumpir.	Cuando se te ocurre lo siguiente y no quieres esperar a que acabe.
`Shift+Enter`	Salto de línea en un mensaje multilínea.	Requiere ejecutar `/terminal-setup` una vez. En Terminal.app de Mac es `Option+Enter`.

Ficheros del proyecto

Fichero o carpeta	Qué hace	Cuándo usarlo
`CLAUDE.md`	Las reglas del proyecto. Es lo primero que Claude lee en cada sesión.	Siempre. Corto, específico y solo con lo que Claude no puede deducir solo.
`CLAUDE.local.md`	Tus manías personales, fuera de git.	Para separar lo tuyo de lo del equipo sin llenar el fichero compartido.
`~/.claude/CLAUDE.md`	Reglas globales, válidas en todos tus proyectos.	Tu forma de trabajar (gestor de paquetes, estilo de código) que no cambia de repo en repo.
`.claude/skills/`	Skills del proyecto: se suben a git y las ve quien clona el repo.	Procedimientos atados a ESE proyecto: su stack, sus convenciones, su despliegue.
`~/.claude/skills/`	Skills globales, disponibles en cualquier proyecto de tu máquina.	Lo tuyo y personal: tu checklist de revisión, tu formato de commits.
`.claude/agents/`	Subagentes especializados con su contexto aislado.	Cuando quieres separar investigar / escribir / revisar para que no se mezclen los contextos.
`.claude/settings.json`	Permisos, hooks y variables de entorno del proyecto.	Para automatizar comprobaciones y dejar de aprobar a mano lo de siempre.

Anatomía de una skill

Pieza	Qué hace	Cuándo usarlo
`.claude/skills/<nombre>/SKILL.md`	Un fichero Markdown en su carpeta. Con eso ya existe la skill.	Es todo el andamiaje que hace falta. No hay más ceremonia.
`name:`	El nombre con el que la invocas: `/nombre`.	Corto y en minúsculas. Es la vía fiable de disparar la skill.
`description:`	Qué hace **y en qué situación aplica**. Es lo único que Claude lee para decidir si usarla.	El campo que decide si la skill se dispara sola. Escribe el "cuándo", no solo el "qué".
Cuerpo del `SKILL.md`	El procedimiento paso a paso, en texto plano.	Claude solo lo lee *después* de decidir usar la skill. Aquí va el detalle.
`allowed-tools:`	Acota qué puede tocar la skill, p. ej. `Read, Bash(git log:*)`.	En skills que ejecutan cosas: limita el radio de acción antes de necesitarlo.
`scripts/`	Ficheros propios de la skill (scripts, plantillas, ejemplos).	Cuando la tarea no se resuelve solo con texto y hay que ejecutar algo de verdad.
`git add .claude/skills/…`	Comparte la skill con el equipo.	Tu compañero hace `git pull` y ya la tiene. Sin zips ni copiar carpetas.

Errores típicos

- **Una `description` vaga.** "Ayuda con el código" no dispara nada: Claude decide leyendo solo `name` y `description`, y con eso no sabe cuándo aplica. Describe la situación concreta, no la categoría.
- **Escribir y rezar.** Pedir algo complejo sin Plan Mode y esperar que salga. Dos pulsaciones de `shift+tab` y revisar el plan cuesta un minuto; deshacer el código equivocado cuesta una tarde.
- **Convertir el `CLAUDE.md` en un vertedero.** Apunta, no vuelques: deja una línea que diga dónde está el detalle (`/docs/brand-voice.md`) en vez de pegar el detalle entero. El fichero principal tiene que seguir siendo rápido de leer.
- **No cerrar nunca la sesión.** Encadenar tareas distintas en la misma conversación provoca *context rot*: Claude arrastra decisiones viejas que ya no aplican. `/clear` al cambiar de tarea, `/compact` si sigues con lo mismo.
- **Guardar la skill en el sitio equivocado.** En `~/.claude/skills/` tu compañero no la verá nunca; en `.claude/skills/` de un repo ajeno, tú tampoco. Del proyecto lo que dependa del proyecto; global lo que sea tuyo.

Preguntas frecuentes

¿Por qué mi skill no se dispara sola?

Porque Claude decide si usar una skill leyendo solo el `name` y la `description`; nunca ha visto el cuerpo todavía. Si la description es vaga ("ayuda con el código"), no tiene forma de saber que existe para tu caso. Escribe qué hace y en qué situación aplica, y se disparará sola.

¿Dónde guardo una skill: en el proyecto o global?

Si depende del contexto de ese proyecto (su stack, sus convenciones), va en `.claude/skills/` y se sube a git con el resto del código. Si es algo tuyo y personal que quieres en todos los proyectos, va en `~/.claude/skills/`.

¿Qué diferencia hay entre una skill y un subagente?

Una skill es conocimiento y procedimiento que Claude carga en la sesión actual cuando es relevante. Un subagente es una sesión aparte, con su propio contexto aislado y sus propias herramientas, a la que delegas un trabajo completo. Skill para *cómo* hacer algo; subagente para que *otro* lo haga sin ensuciar tu contexto.

¿Qué es el Plan Mode y cuándo conviene usarlo?

Es el modo en que Claude diseña y te enseña un plan antes de tocar una sola línea. Se activa pulsando `shift+tab` dos veces. Úsalo siempre que la tarea toque varios ficheros o no tengas clara la arquitectura: corregir un plan cuesta un minuto, corregir el código ya escrito cuesta una tarde.

¿Necesito saber programar para usar Claude Code?

Para construir páginas, dashboards o automatizaciones sencillas, no: se pide en lenguaje natural y Claude escribe y prueba los ficheros. Lo que sí necesitas es saber revisar lo que hace y trabajar por iteraciones, pidiendo una primera versión y puliéndola.

Vídeos relacionados

- Todos los Niveles de Claude Code Explicados en 50 Minutos — del Plan Mode a los pipelines autónomos.
- Canal de Juan Gabriel Gomila — más tutoriales de IA y programación en español.

¿Quieres pasar de pedirle cosas sueltas a un chat a diseñar sistemas que trabajan solos? Eso es exactamente lo que enseñamos, paso a paso y con proyectos reales, en el curso de Frogames.

De Vibe Coder a Ingeniero Agéntico →