⬇ Scarica questo libro in PDF

Apertura

E se lo strumento che ti manca potessi costruirtelo da solo?

Programmare con l'AI — anche se non hai mai scritto una riga di codice.

Quante volte hai pensato "mi servirebbe un programmino che fa questa cosa", per poi lasciar perdere perché non sai programmare? Un foglio che si aggiorna da solo, un piccolo strumento per il tuo lavoro, un'idea che hai in testa e che resta lì. Non ti chiedo se hai mai voluto costruire qualcosa. Ti chiedo quante idee hai abbandonato solo perché "non sono capace".

Se questa domanda ti ha punto, questa pagina è per te.

Te lo dico subito, ed è la notizia che cambia tutto: oggi non devi saper programmare per costruire. Devi saper spiegare cosa vuoi. L'AI scrive il codice; tu lo guidi a parole. La barriera che ti ha fermato per anni — quella sintassi incomprensibile, quei simboli — non è più tra te e la cosa che vuoi creare. Chi lo capisce inizia a costruire; chi non lo capisce continua ad aspettare che lo faccia qualcun altro.

Lo so perché ci sono passato. Non venivo dal codice, e per un sacco di tempo ho creduto che "costruire strumenti" fosse roba per altri. Ho toccato il fondo di quella convinzione il giorno in cui ho rinunciato all'ennesima idea per paura della parte tecnica. Poi ho provato a fare l'unica cosa sensata: descrivere a parole quello che volevo e lasciare che l'AI scrivesse. Funzionava. La prima volta che un mio strumento è partito davvero, qualcosa è cambiato per sempre.

Costruire con l'AI non significa diventare programmatore. Significa diventare quello che dice cosa va fatto e controlla che funzioni. La parte difficile non è più il codice: è avere chiara l'idea. E le idee, quelle, ce le hai già.

In queste pagine non trovi un corso di informatica. Trovi un percorso pratico: dall'idea allo strumento che funziona, spiegato per chi parte da zero. Come chiedere il codice, come provarlo, come sistemarlo quando qualcosa non va — sempre a parole, sempre col tuo controllo. Capitoli corti, un piccolo progetto vero che cresce mentre leggi.

E no, non devi imparare un linguaggio di programmazione. Devi solo imparare a guidare chi lo conosce — e quel "chi" è l'AI. Alla fine di queste pagine avrai costruito qualcosa con le tue mani. O meglio: con le tue parole.

Ma leggere non basta: si costruisce costruendo. Il libro ti dà il percorso; per partire davvero serve mettere le mani su un progetto vero. Per questo ho aperto un corso gratuito su Skool: ti guido a costruire i primi strumenti passo passo, anche senza programmare, con una community di builder alle prime armi come te. È gratis. Entra e costruisci la tua prima cosa con me.

→ https://www.skool.com/l-ai-e-la-nuova-elettricita-8966/about

---

Dedica

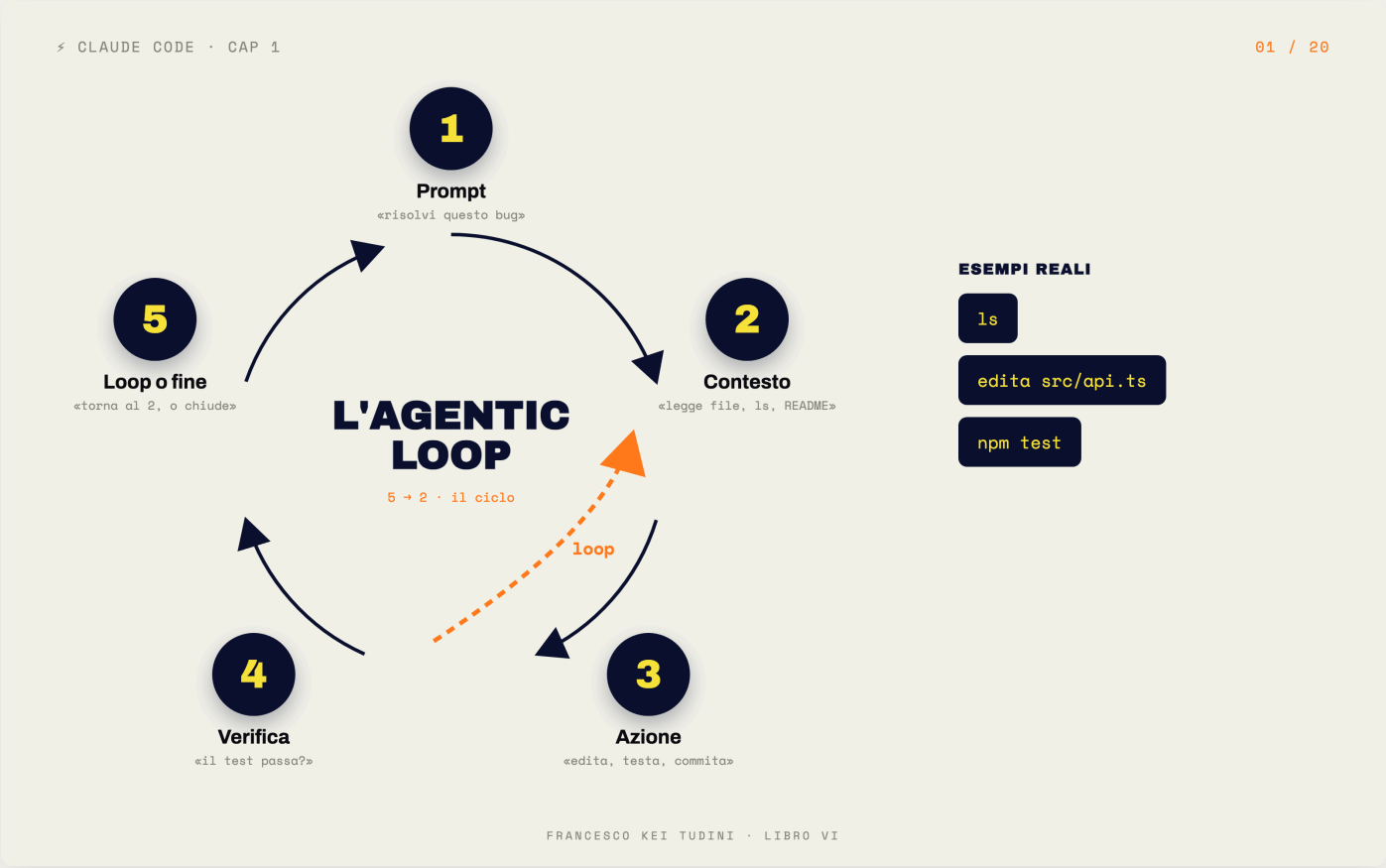
A mia figlia Minerva.

AI, in giapponese, vuol dire amore.

Indice

- Chiusura libro
- Back-cover

Capitolo 1 — Code non è una chat. È un compagno di lavoro nel tuo computer



«Sì, ma in fondo è sempre Claude, no?»

No. È la prima cosa da sistemare. **Claude Code non è un chatbot.** Non è la chat di claude.ai con qualche pulsante in più. Non è nemmeno Cowork in versione tecnica. È un agente che vive dentro la tua macchina, legge i tuoi file, esegue comandi al terminale, modifica codice e verifica da solo se quello che ha fatto funziona.

Sembra un dettaglio. Non lo è. Cambia tutto: cosa puoi chiedere, come glielo chiedi, dove può sbagliare, cosa firmi tu alla fine.

Pensa a Cowork — la scrivania aumentata su file e sistemi generici. Code è il fratello tecnico: stesso principio (l'AI lavora con te in un loop, in autonomia, con permessi), terreno diverso (un codebase, un terminale, dei commit Git). Se hai capito Cowork, hai già il 70% di Code. Il 30% che manca è dove ti porto in questo capitolo.

Cos'è un agente, davvero

Definizione operativa, una volta sola: **un agente è un programma che, in autonomia, alterna ragionamento e azione per raggiungere un obiettivo, usando strumenti.** Punto.

Una chat è statica. Le scrivi, ti risponde. Se vuoi che modifichi un file, copi, incolli, riscrivi, ricopi. Un agente no. Un agente prende il file da solo, lo apre, ci lavora, prova, sbaglia, corregge. Tu controlli al checkpoint. Lui fa il movimento.

Sotto il cofano c'è un meccanismo che si chiama **agentic loop** e ha cinque passi.

1. **Tu dai il prompt.** «Risolvi questo bug.» «Aggiungi questa funzionalità.» «Spiegami cosa fa questa cartella.»
2. **Claude raccoglie il contesto.** Legge file, lancia un `ls`, fa una ricerca dentro al codice, apre il README.
3. **Claude agisce.** Modifica un file. Lancia un test. Installa una dipendenza. Commita.
4. **Claude verifica.** Il test passa? L'app si avvia? Il diff è coerente con la richiesta?
5. **Loop.** Se serve, torna al passo 2 con il nuovo stato di mondo. Se ha finito, chiude.

Cinque passi. Sembrano semplici. Sono il motivo per cui Claude Code ti fa risolvere in venti minuti cose che a un team di tre persone richiedono mezza giornata — e anche il motivo per cui ti può rompere il progetto in dieci secondi, se non sai dove sono i freni.

Le tre cose da sapere prima di toccare un tasto

Tre. Non quattro. Non dieci. **Tre.**

Uno: la finestra di contesto (context window) è finita. È la memoria di lavoro di Claude in quella sessione. Ci entra dentro la tua conversazione, ogni file che legge, ogni output di comando, il contenuto di CLAUDE.md, i risultati dei tool. Sembra enorme — ed

è enorme. Ma in un progetto vero, dopo un paio d'ore, è piena. Quando si riempie, Claude **compatta**: riassume in automatico per fare spazio. Comodo, ma puoi perdere dettagli. Il comando `/context` ti mostra quanto è piena; `/compact` la riduce su richiesta; `/clear` la azzera per ripartire pulito. Tre comandi. Da imparare il primo giorno.

Due: Claude chiede il permesso. Per default. Ogni modifica a un file, ogni comando di shell, te lo mostra prima e aspetta il tuo OK. Lo vedi come un blocco colorato con un diff: verde quello che aggiunge, rosso quello che toglie. Tu premi `y` (accetta), `n` (rifiuta), `e` (modifica), `a` (accetta tutto da qui in poi). C'è anche **Plan Mode**: in quella modalità Claude usa solo strumenti in lettura, non tocca niente, e ti prepara un piano. Quando il piano ti convince, dai l'OK e parte. Plan Mode è dove correggi *prima* che il danno succeda. È la differenza tra un consulente che pianifica e uno che spara.

Tre: Claude sbaglia. Non spesso, non in modo catastrofico, ma sbaglia. Legge male un file di configurazione, rinomina una variabile in trenta posti tranne uno, ti propone un commit con un messaggio che non c'entra. Per questo c'è l'**human-in-the-loop**. Sei tu il filtro. Se accetti tutto al buio (modalità YOLO, che vedrai nel Capitolo 2), il filtro sparisce e la responsabilità no.

Tieni a mente: **memoria finita, permesso richiesto, errore possibile**. Questi sono i tre freni del veicolo. Senza non guidi.

Dove gira Claude Code: terminale, IDE, desktop, web

QUATTRO MODI, STESSO STRUMENTO

	TERMINALE	IDE	DESKTOP	WEB
Funzionalità nuove per prime	✔ si	◐ parz.	◐ parz.	✗ no
Diff visivi	◐ parz.	✔ si	◐ parz.	◐ parz.
Lavoro offline / locale	✔ si	✔ si	✔ si	✗ solo GitHub
Restrizioni	—	—	—	✗ solo repo GitHub
VOTO	il riferimento	diff visivi	lavori lunghi	plan B

Quattro modi, stesso strumento. Lo dico subito così non ti perdi.

Terminale. È la modalità di riferimento. Funziona su macOS, Linux, WSL (Windows Subsystem for Linux), Windows nativo. È quella dove le funzionalità nuove arrivano per prime. Apri il terminale, vai nella cartella del progetto, scrivi `claude`, parte. Tutto quello che è in quella cartella e nelle sottocartelle, Claude lo vede.

IDE. L'estensione esiste per **VS Code** (Marketplace ufficiale, badge blu Anthropic) e per **JetBrains** (IntelliJ, PyCharm, WebStorm, RubyMine — tutta la famiglia). Sotto cambia poco: è lo stesso Claude Code, esposto in un pannello laterale invece che in una finestra del terminale. Vantaggio: vedi i diff direttamente nell'editor, copi-incolli con un click, hai sempre la sintassi colorata.

App desktop. Claude Desktop ha un toggle in alto che dice «Code». Premi quel toggle e sei nello stesso ambiente, con un look più simile alla chat. Comodo se vuoi tenere Code in background mentre fai altro col Mac.

Web. Vai su `claude.ai/code`. Stessa interfaccia. Unico vincolo: lavori solo su repository GitHub. Niente cartelle locali. È la modalità giusta se vuoi lavorare da un

computer non tuo o da un tablet.

Quale scegliere? La regola è semplice. Se hai un Mac o un PC con un terminale, **inizia dal terminale**. È lì che le funzionalità escono prime e lì che impari il ritmo vero del tool. Quando hai capito i flussi, sposta una parte del lavoro nell'IDE per i diff visivi. L'app desktop è perfetta per i lanci lunghi in background («fammi il refactor mentre faccio altre tre cose»). Il web è plan B.

Cosa cambia rispetto alla chat

Te lo dico in tre numeri pratici, presi dal lavoro reale.

Da copia-incolla a zero copia-incolla. Con la chat, per modificare un file lavori in cinque passi: leggi nella chat, copi, incolli nel tuo file, salvi, torni nella chat. Con Code, scrivi una frase. Claude apre il file, lo modifica, te lo mostra, tu approvi. Cinque azioni diventano una.

Da una cosa per volta a molte cose insieme. Nella chat porti un pezzo di codice. In Code Claude vede tutta la cartella. Se modifichi una funzione, Claude trova *tutti* i posti dove quella funzione è chiamata e li aggiorna. Un refactor che a mano richiede due ore, fatta bene, si fa in cinque minuti.

Dal "spiegami" al "fallo". La chat ti spiega. Code esegue. Ti fa girare i test. Ti installa la libreria. Ti committa. Ti apre la pull request. La differenza tra capire una cosa e averla fatta.

E sì, può fare anche cose che la chat non fa proprio: lanciare uno script di backup, rinominare cento file con un pattern, leggere i log e proporre una patch. Tutto restando nel loop col tuo OK a ogni passo.

Errori da non fare (al primo contatto)

Trattarlo come una chat. Scrivere prompt da chatbot («ciao, mi spieghi React?») quando hai un agente con strumenti veri vuol dire sprecare il 90% del potenziale. Il prompt giusto in Code è operativo: «leggi `src/api/` e dimmi come è strutturato»,

«trova il file che gestisce i pagamenti», «aggiungi un test per la funzione `parseInvoice`».

Aprire la cartella sbagliata. Claude vede tutto ciò che è nella cartella in cui lo lanci, e nelle sottocartelle. Se lanci `claude` nella tua home directory, gli stai dando in mano l'intero disco. Da Documenti a Scaricati a Foto. Lancialo nella cartella del progetto. Nient'altro.

Premere `a` (accept all) il primo giorno. L'auto-accept ha senso quando sai cosa stai facendo. Il primo giorno guarda ogni diff. Il primo giorno è la fase in cui impari **come** Claude pensa: chi salta questa fase si ritrova bug strani due settimane dopo senza sapere da dove arrivano.

Pensare che funzioni «tanto è AI». Funziona perché tu stai nel loop. Senza di te è un programma che modifica file e basta. Con te è uno sviluppatore in più al tavolo. Il moltiplicatore sei tu.

Cosa porti a casa

- **Code è un agente, non una chat.** Vive nel tuo computer, legge i tuoi file, esegue comandi, verifica da solo.
- **L'agentic loop ha cinque passi.** Prompt → contesto → azione → verifica → loop. Da imparare a memoria.
- **Tre cose da sapere subito.** Finestra di contesto finita, permesso richiesto a ogni azione, possibilità di errore. Memorizzale.
- **Quattro modi per usarlo.** Terminale (il riferimento), IDE (i diff visivi), desktop (i lavori lunghi), web (da remoto su GitHub).
- **Il moltiplicatore sei tu.** Senza human-in-the-loop, Code è un'auto senza freni.

Capitolo 2 — Setup in 10 minuti

Quanto ci vuole per avere Claude Code operativo sul tuo computer?

Dieci minuti. Se sai dove cliccare, sei. Se non lo sai, leggi questo capitolo e diventano dieci.

Il setup è la parte più sopravvalutata e sottovalutata insieme. Sopravvalutata perché tanti rimandano l'inizio per «paura del terminale» — paura che, lo vedrai, costa di più del fare. Sottovalutata perché il modo in cui imposti il setup determina cosa Claude potrà fare nei mesi successivi senza romperti niente. Permessi larghi oggi, casino domani. Permessi giusti oggi, libertà per anni.

Andiamo per pezzi: installazione, login, scelta della cartella, permessi, primo prompt utile.

Installazione: un comando per ogni sistema

L'installazione è ufficialmente documentata da Anthropic e funziona via quattro canali. La maggior parte delle installazioni si fa con un comando solo. Scegli il tuo.

macOS, Linux, WSL — il comando di riferimento:

```
curl -fsSL https://claude.ai/install.sh | sh
```

Una riga. Esce installato e si auto-aggiorna. È la via consigliata da Anthropic e quella per cui le funzionalità nuove arrivano per prime.

Homebrew (Mac, gestore di pacchetti diffuso):

```
brew install anthropic/claude/claude-code
```

Funziona, ma niente auto-update — devi dare tu `brew upgrade` ogni tanto.

Windows con PowerShell: c'è un comando `Invoke-RestMethod` analogo al curl. Su CMD funziona anche curl. Per chi preferisce, c'è `winget install Anthropic.ClaudeCode` — comodo, ma anche qui niente auto-update.

npm (se sei già nel mondo Node):

```
npm install -g @anthropic-ai/claude-code
```

Una volta installato, vai nella cartella del progetto e scrivi:

```
claude
```

Si avvia. Ti chiede di scegliere un tema colore. Poi ti chiede come vuoi accedere. Hai due strade.

Login: account o API key

Account Pro, Max o Enterprise. È la via consigliata se Claude Code è il tuo strumento di lavoro quotidiano. Apri il browser, fai login, torni al terminale, sei dentro. Se la tua azienda ha un piano Enterprise, scegli quello: l'integrazione con i sistemi di policy aziendali è fatta apposta per non rompere il GDPR e i regolamenti interni.

API key. È la via per chi usa Code in modalità integrata con altri sistemi, in CI/CD, o per misurare i consumi a richiesta. La key si genera dalla console di Anthropic; la incolli e parti. Da quel momento paghi a token consumati invece che a flat mensile.

Per la maggioranza degli utenti, account piano Pro/Max è la scelta giusta. Per chi automatizza, API key.

La cartella: il confine di quello che Claude vede

Qui c'è una regola che vale oro: **Claude vede tutto ciò che è nella cartella in cui lo lanci e nelle sue sottocartelle. Niente al di fuori.**

Tradotto: se lanci `claude` dalla home directory (`~`), gli stai dando in mano tutto il tuo disco — documenti personali, foto, scaricati, eventuali credenziali in chiaro. Non si fa.

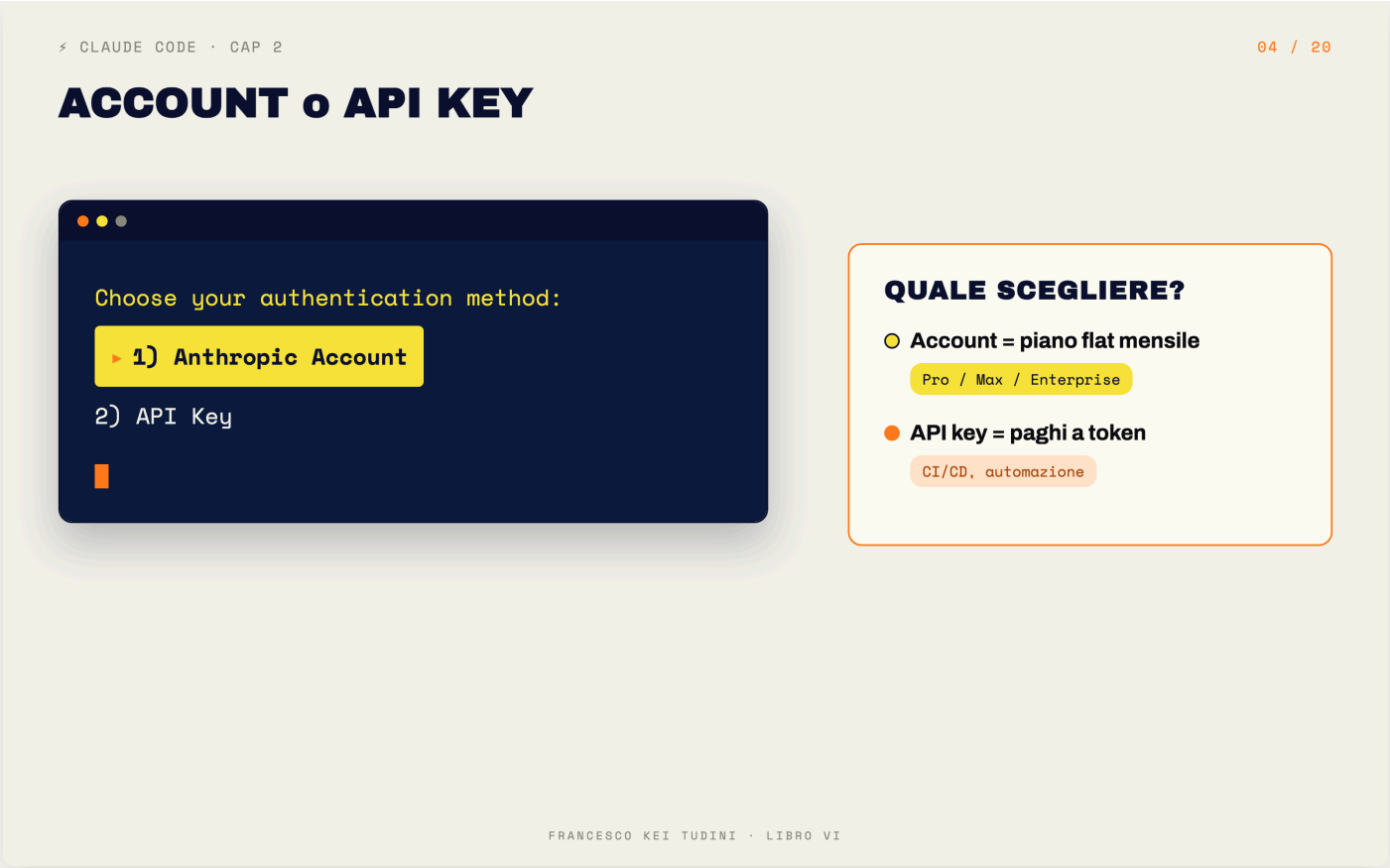
Apri il terminale, vai con `cd` nella cartella del progetto specifico:

```
cd ~/progetti/mio-sito-web
claude
```

Da quel momento la sua «giurisdizione» è quella cartella. Se la cartella contiene 3.000 file e 28 sottocartelle, ok, ma sono *quelli* e basta. Niente di sopra. Niente di laterale.

Se non hai ancora un progetto, creane uno vuoto e lancialo lì. Anche solo per provare. Una cartella «test-claude» con dentro tre file di esempio. Il piacere di iniziare in un ambiente pulito ti farà capire prima cosa succede.

Permessi: i tre modi, e quando usarli



Una volta dentro, Claude di default ti chiede il permesso per ogni azione che modifica qualcosa. Premi `Shift + Tab` per ciclare tra tre modalità di permesso:

Normal (la modalità di partenza). Ogni modifica a file, ogni comando shell ti viene mostrato in un diff colorato. Verde quello che aggiunge, rosso quello che toglie. Tu rispondi `y` (accetta), `n` (rifiuta), `e` (modifica prima di accettare), `a` (accetta tutto da qui in avanti). È la modalità da usare il primo giorno e, in molti contesti, anche il secondo mese. Stai nel loop, vedi tutto.

Auto-Edit. Le modifiche ai file vengono auto-approvate, ma i comandi shell (lancia un test, installa una libreria, esegui un push Git) ti chiedono ancora il permesso. È la modalità giusta quando il piano l'hai già discusso e ti fidi di Claude sulle modifiche di codice, ma non vuoi che esegua niente al sistema senza il tuo OK.

YOLO. Tutto auto-approvato. Modifiche e comandi. Da usare solo in situazioni controllate: container Docker isolato, sandbox, lavori monouso dove un errore si annulla buttando via il container. Anthropic stessa lo chiama «not recommended» nel materiale di formazione. Tradotto: non farlo a meno che tu non sappia cosa stai facendo.

C’è anche una quarta modalità — **Plan Mode** — che vediamo nel prossimo capitolo. In Plan Mode Claude usa solo strumenti in lettura e prepara un piano scritto. Non tocca niente finché tu non approvi. È la modalità d’oro per i task complessi.

I permessi granulari: `.claude/settings.json`

Hai un altro livello, più fine, per dire a Claude *quali comandi specifici* può eseguire senza chiederti e *quali deve sempre rifiutare*. Si configura in un file JSON nella cartella del progetto, dentro `.claude/settings.json`:

```
{
  "permissions": {
    "allow": [
      "Bash(npm test)",
      "Bash(npm run lint)",
      "Bash(git status)",
      "Bash(git diff)"
    ],
    "deny": [
      "Bash(rm -rf *)",
      "Bash(sudo *)",
      "Bash(git push origin main)"
    ]
  }
}
```

Letto in italiano: «permetti automaticamente test, lint, status di Git. Vieta sempre rm-rf, sudo, push diretto su main.» È una rete di sicurezza che ti permette di lavorare velocemente sui comandi innocui senza dover dare l’OK quaranta volte al giorno, e ti blocca a priori i comandi pericolosi.

Questo file lo commiti in Git. Tutti i tuoi colleghi che clonano il progetto ereditano gli stessi permessi. È la prima cosa pro: smettere di pensare ai permessi caso per caso e codificarli una volta.

Il primo prompt utile: «Cosa fa questo progetto?»

Non «ciao Claude». Non «aiutami con Python». Il primo prompt vero, su qualunque progetto, è uno solo:

```
> Cosa fa questo progetto? Leggi i file principali e spiegamelo
  in 200 parole.
```

Quello che succede, lo vedi a schermo. Claude legge il README. Apre `package.json` o `pyproject.toml` o `pom.xml`. Scorre le cartelle principali. Forse apre due file chiave. E ti scrive il riassunto.

Quel riassunto è la tua prima verifica: **ha capito davvero?** Se sì, stai parlando con un collaboratore allineato. Se no — se ti racconta cose che non sono nel progetto — la cartella è sbagliata, il README è obsoleto, manca qualcosa. Meglio scoprirlo nei primi due minuti che dopo due ore di lavoro.

Il secondo prompt che ti consiglio:

```
> Crea un file CLAUDE.md che spieghi a un nuovo sviluppatore
  come è strutturato questo progetto, le convenzioni che vede
  nel codice, e i comandi principali per lanciare i test.
```

In trenta secondi hai un file di memoria che resterà nel progetto e che Claude leggerà a ogni sessione futura. Su CLAUDE.md torno per disteso nel Capitolo 4 — qui ti basta sapere che esiste, che è la base di tutto, e che il momento giusto per crearlo è *adesso*. Non «poi». Adesso.

Errori da non fare nel setup

Lanciare Claude dalla home directory. Te l'ho già detto, lo ripeto perché è l'errore numero uno. Dai a Claude la cartella del progetto, non l'intero disco.

Saltare il file di permessi. È il primo giorno, hai voglia di vedere cosa fa, e ti dici «poi lo configuro». Poi non lo configuri mai. Configuralo prima del primo task vero. Cinque righe di JSON ti salvano da `rm -rf` su una cartella sbagliata.

Usare YOLO mode per «andare più veloce». Vai più veloce verso il disastro. La velocità vera arriva dopo, quando hai imparato i flussi. Il giorno uno, modalità Normal.

Non creare CLAUDE.md. Senza CLAUDE.md, ogni sessione Claude ricomincia da zero. Tu spendi il primo quarto d'ora a rispiegargli cosa stai costruendo. Con CLAUDE.md, è già operativo dalla prima riga.

Cosa porti a casa

- **Installazione: un comando.** `curl ...`, oppure `brew install`, oppure `npm install -g @anthropic-ai/claude-code`. Dieci secondi.

- **Login: account o API key.** Per uso quotidiano, account Pro/Max/Enterprise. Per automazione, API key.
- **Cartella: la giurisdizione di Claude.** Lancialo solo dentro la cartella del progetto. Mai dalla home.
- **Permessi: tre modi + file granulare.** Inizia in Normal. Imposta `allow` / `deny` in `.claude/settings.json`. Eviti `rm -rf` e `sudo` per sempre.
- **Primo prompt utile.** «Cosa fa questo progetto?» seguito da «Crea un CLAUDE.md». In quattro minuti hai un collaboratore operativo.

Capitolo 3 — Il flusso che vale tutto il libro: explore → plan → code → commit

Se di questo libro porti via una cosa sola, sia questa: quattro parole in fila.

Explore. Plan. Code. Commit.

Anthropic le ripete in ogni corso ufficiale. Le ripetono gli ingegneri che pubblicano sul blog di engineering. Le ripete chi usa Claude Code da un anno e l'ha visto rompere e ricostruire decine di progetti. È il flusso. Non «un» flusso. **Il** flusso.

Funziona così: non chiedere mai a Claude di scrivere codice prima di averlo fatto esplorare. Non farlo esplorare senza chiedergli un piano. Non far partire il codice senza aver approvato il piano. Non chiudere mai senza un commit pulito. Quattro fasi, in quest'ordine, sempre. Se ne salti una, paghi il prezzo nelle altre tre.

Ora le smonto fase per fase.

Fase 1: Explore — far capire a Claude dove sta

L'errore numero uno di chi inizia: aprire Claude e scrivere «aggiungimi una funzionalità X». Senza che abbia letto niente del progetto. Senza che sappia con che linguaggio sei. Senza che abbia visto le tue convenzioni. Risultato: ti propone un codice che, sì, fa

quello che hai chiesto, ma è scritto in uno stile che non c'entra con il resto, usa una libreria che non hai installato, rompe due test che non sapeva esistessero.

Esplorare significa fargli capire il terreno *prima* di costruire.

Prompt tipici di una fase di explore fatta bene:

```
> Leggi i file principali di questo progetto e dimmi:
  - in che linguaggio è scritto
  - quali sono le directory principali
  - qual è il framework usato (se ce n'è uno)
  - dove si trovano i test
```

Oppure, se sai già quasi tutto:

```
> Cerca nel codice tutte le funzioni che gestiscono i pagamenti.
  Dimmi in che file stanno e come si chiamano. Non modificare niente.
```

Nota le ultime parole: «non modificare niente». In fase di explore, Claude legge. Punto. Nessun edit. Nessun comando shell pericoloso. È una fase di osservazione.

Quanto dura? Dipende dal progetto. Su una piccola app, due minuti. Su un codebase enterprise, anche un quarto d'ora — e va bene così, perché quei quindici minuti ti risparmiano ore di codice fuori standard.

Fase 2: Plan — il momento d'oro

Plan Mode è la modalità più sottovalutata di Claude Code. È quella che separa chi usa lo strumento in modo amatoriale da chi lo usa in modo professionale.

Si attiva con Shift+Tab finché non vedi «Plan Mode» in basso. In Plan Mode Claude usa **solo** strumenti in lettura: legge file, esegue ricerche, naviga il codebase. Non scrive niente. Non tocca niente. Il suo output è un *piano scritto*: «per fare X farò questi cinque passi, in quest'ordine, modificando questi file, e prevedo di lanciare questi test alla fine».

Tu leggi il piano. È **la** finestra per correggere prima del danno.

Esempio reale di prompt in Plan Mode:

```
> La mia app ha bisogno di una dark mode in tutta l'interfaccia.
  Voglio un toggle nell'header che cambi il tema. Trova un colore
  di contrasto coerente con il tema chiaro attuale. Preparami un
  piano.
```

Claude entra in azione: legge i file CSS, capisce dove è definito il tema, identifica l'header, controlla la struttura dei componenti, e ti scrive qualcosa tipo:

> Piano: 1. Aggiungere variabili CSS per il tema scuro in `src/styles/theme.css`. 2. Modificare `src/components/Header.jsx` per aggiungere il toggle. 3. Creare un hook `useTheme` in `src/hooks/useTheme.js`. 4. Aggiornare `App.jsx` per leggere il tema dal localStorage. 5. Aggiungere un test per il toggle in `src/components/Header.test.jsx`.

Tu leggi. Vedi che al passo 4 c'è scritto `localStorage` ma il tuo team non vuole usare localStorage per il tema (avete una preferenza utente lato server). Glielo dici:

```
> Il tema non va salvato in localStorage, va salvato nelle
  preferenze utente lato server con la API /api/preferences.
  Rivedi il piano.
```

Claude rivede. Ti propone un piano corretto. Tu approvi. **Adesso** parte il codice.

Quei tre minuti di Plan Mode ti hanno risparmiato un commit sbagliato, una pull request da rifare, mezza giornata di review.

Fase 3: Code — la fase più sopravvalutata

Curioso ma vero: la fase «code» è quella che richiede meno cervello tuo, se hai fatto bene le prime due.

Claude ha letto. Claude ha pianificato. Tu hai approvato. Da qui in poi è esecuzione. Lui modifica i file uno per uno, ti mostra ogni diff, tu approvi (o stai in Auto-Edit se ti fidi della singola operazione). Lancia i test. Se passano, va avanti. Se falliscono, prova a correggere. Loop.

Cosa fai tu? Resti nel loop. Leggi i diff. Annulli ciò che non ti convince. Correggi a voce — «no, questa funzione chiamala `parseInvoiceData`, non `processInvoice`» — e lui

ricorregge. Il tuo lavoro qui è quello di un revisore-collaboratore: vedi il codice mentre nasce, lo aggiusti al volo, non aspetti la pull request per scoprire che è una catastrofe.

Una cosa cambia tutto in questa fase: dare a Claude **gli strumenti per verificare da solo**. Tipicamente sono due cose.

Una suite di test che lui può lanciare. Se hai `npm test`, lui lo lancia dopo ogni modifica significativa, vede l'output, capisce se ha rotto qualcosa. Se non hai test, sei tu il test — e l'occhio umano vede meno bene.

Un linter o un formatter che lui può eseguire. Prettier per JS/TS, Black per Python, gofmt per Go, rustfmt per Rust. Sono comandi banali ma garantiscono che il codice sia formattato come il resto del progetto.

Più strumenti gli dai, più Claude verifica da solo, meno tu devi controllare a mano. È il principio cardine di tutti gli agenti: **impalcatura tecnica più ricca = autonomia più affidabile**.

Fase 4: Commit — la firma finale è tua

Il codice gira. I test passano. Hai finito? Quasi.

Prima del commit, due passaggi che separano i dilettanti dai pro.

Passo uno: code review. Lanci un subagent (li vediamo nel prossimo capitolo) o chiedi a Claude stesso, in un thread pulito, di rivedere quello che ha appena scritto. Sembra strano — l'autore che recensisce se stesso — e infatti il trucco è: il revisore lavora in un altro contesto, con un altro system prompt, focalizzato solo sulla qualità e sicurezza. Trova cose che il «costruttore» non aveva notato: una variabile non usata, una gestione errori mancante, un import duplicato.

Passo due: messaggio di commit. Chiedi a Claude:

```
> Scrivi un messaggio di commit per le modifiche di questa sessione.
  Stile: prima riga 50 caratteri max, modo imperativo. Body con
  i tre cambiamenti principali a punto elenco.
```

Lui legge il diff completo, ti scrive un messaggio pulito. Tu controlli. Approvi. Commit.

Se vuoi anche aprire una pull request:

```
> Crea una pull request da questo branch verso main. Titolo chiaro,
  descrizione che spieghi cosa fa, perché, e come testarla.
```

Claude esegue `gh pr create` (la CLI di GitHub), inserisce titolo e body, apre la PR. La firma resta tua: tu sei il committer, tu apri la pull, tu rispondi alle review. L'AI ha solo accorciato i passaggi.

Quando saltare una fase? Mai.

Lo dico chiaro: **non saltare mai una fase del flusso**.

«Ho già esplorato l'altro giorno, salto explore.» — la finestra di contesto di Claude è cambiata, non se lo ricorda con la stessa freschezza. Cinque secondi di re-explore e sei a posto.

«È una modifica banale, salto plan.» — sono le modifiche «banali» quelle che ti rompono tre cose collaterali. Plan Mode su una modifica banale dura trenta secondi.

«Lo committo dopo.» — «dopo» significa che alla fine della giornata hai sette modifiche stratificate, non sai più cosa hai cambiato e perché, e il rollback diventa un'archeologia.

Il flusso è come la cintura di sicurezza: ti annoia finché non ti serve.

Aggancio alle quattro abitudini: la Delega vera è sul piano

Tieni a mente le quattro D: Delega, Descrizione, Discernimento, Diligenza.

In Claude Code la **Delega** vera non è «scrivimi il codice». Quella è esecuzione. La delega vera è **fargli pianificare**. Il piano è la parte intellettuale del lavoro. Se tu ti tieni il piano (lo fai a mente) e gli deleghi solo l'esecuzione, hai sprecato la metà del valore. Se gli deleghi il piano e poi lo valuti, tu fai il consulente — lui fa il junior brillante che ha letto tutto il codice in dieci secondi. Sei in posizione di leva. Cambia il rapporto.

La **Descrizione** è il prompt di explore e quello di plan: «leggi questi file», «proponi un piano con questi vincoli».

Il **Discernimento** è leggere il piano e dire «no, qui no».

La **Diligenza** è la review, il commit, il push. La firma.

Esempi pratici di flusso completo

Uno sviluppatore freelance. Cliente chiede una pagina di login con OAuth. Explore: «leggi la struttura del progetto e dimmi se c'è già un sistema di autenticazione». Plan:

«proponi come integrare OAuth con Google, mostrami che file cambierai e quali test aggiungerai». Code: Claude scrive, lui verifica i diff. Commit: subagent reviewer, messaggio di commit, pull request. Tempo: 50 minuti invece di quattro ore.

Una piccola software house. Bug in produzione, l'app crasha al login. Explore: «leggi gli ultimi log e i file relativi al login». Plan: «identifica la causa probabile e proponi una correzione». Code: la patch. Commit: hot-fix con messaggio chiaro, push, deploy. Tempo: 20 minuti invece di due ore.

Un imprenditore non tecnico (anticipo del Capitolo 6). Vuole uno script che gli pulisca un CSV di lead in arrivo da una landing page. Explore: «leggi questo CSV, dimmi che colonne ha, dove sono i dati sporchi». Plan: «proponi uno script Python che normalizzi i numeri di telefono, sistemi i nomi in title case, elimini i duplicati per email». Code: lo script. Commit: nessuno (non è un progetto in Git) — solo salvataggio e prova. Tempo: 15 minuti invece di non saperlo fare per niente.

Errori da non fare

Saltare Plan Mode «perché ho fretta». Plan Mode dura tre minuti. Se hai fretta, è proprio Plan Mode che ti serve di più, non meno.

Fare un prompt unico monstre. «Esplorami questo progetto, fammi un piano, scrivi il codice e fai il commit.» Sembra efficiente. È il modo migliore per ritrovarsi commit di cento file con cose dentro che non avresti voluto. Una fase per volta.

Approvare un piano senza leggerlo. Il piano scritto è un contratto. Se non lo leggi e lo approvi al volo, hai firmato in bianco.

Saltare la review prima del commit. Tre minuti di subagent reviewer ti salvano una settimana di bug strani in produzione.

Cosa porti a casa

- **Quattro fasi, in quest'ordine, sempre.** Explore, Plan, Code, Commit.
- **Plan Mode è la modalità d'oro.** Tre minuti di piano scritto valgono ore di codice corretto in corsa.

- **Code è la fase con meno cervello tuo, se hai fatto bene le prime due.** Stai nel loop, leggi i diff, correggi a voce.
- **Il commit ha sempre una review prima e un messaggio dopo.** Mai un commit nudo a fine giornata.
- **Le 4D mappano sul flusso.** Delega = Plan. Descrizione = prompt. Discernimento = review piano e codice. Diligenza = commit.

Capitolo 4 — Memoria, competenze, deleghe, regole: CLAUDE.md, skills, subagents, hooks

Perché la stessa AI ieri ti ha dato un codice pulito e oggi un disastro?

Perché ieri aveva la memoria giusta. Oggi non l'aveva. Punto.

Tra Claude Code «usato da chi ha provato due volte» e Claude Code «usato da chi sa quello che fa», la differenza non è il modello, non è il prompt magico, non è la fortuna. È **quanto Claude conosce il tuo progetto prima di iniziare**. Più contesto operativo gli dai a priori, meno deve cercarlo per ogni task. Meno cerca, meno sbaglia. Meno sbaglia, meno tempo perdi a correggere.

Claude Code mette in mano quattro strumenti per dare contesto in modo strutturato. Si chiamano CLAUDE.md, skills, subagents, hooks. Hanno nomi tecnici, ma funzionano in modo intuitivo se li traduci nel tuo italiano.

- **CLAUDE.md** è la **memoria di progetto**. Sempre attiva, sempre letta.
- **Skills** sono **competenze a richiesta**. Caricate solo quando servono.
- **Subagents** sono **deleghe** a un Claude figlio con una finestra di contesto separata.
- **Hooks** sono **regole inviolabili** che scattano in automatico.

Quattro strumenti. Funzioni diverse. Si combinano. Adesso li smonto uno a uno.

CLAUDE.md: la memoria sempre attiva

CLAUDE.md è un file markdown, lo metti nella radice del tuo progetto, e Claude lo legge **automaticamente** all'inizio di ogni sessione. Punto. Non devi caricarlo a mano. Non devi referenziarlo. C'è, e lui lo legge.

Cosa ci metti dentro? Tutto quello che vorresti dire a un nuovo sviluppatore che entra nel team il primo giorno, e che ti scoccia ripetere mille volte:

- A cosa serve il progetto, in cinque righe.
- Quale stack tecnologico usi.
- Le convenzioni di stile del codice (camelCase o snake_case? indentazione a 2 o 4 spazi?).
- Come si lanciano i test.
- Come si fa il deploy.
- Cosa **non** deve essere toccato (file di configurazione protetti, cartelle critiche).
- Quali strumenti e librerie sono il default del team.

Esempio reale, scarno e funzionale:

```
# CLAUDE.md

## Cos'è questo progetto

![](img/L06/L06_09_quattro_d.png)

API REST per la gestione preventivi di uno studio commercialista.
Stack: Node.js 20, Express, PostgreSQL, Prisma ORM.

## Convenzioni

- Codice in TypeScript (strict mode).
- Naming: camelCase per variabili, PascalCase per classi.
- Stringhe: virgolette singole.
- Indentazione: 2 spazi.

## Comandi utili

- `npm test` per i test (Jest)
- `npm run lint` per ESLint
- `npm run dev` per dev server
- `npm run db:migrate` per Prisma migration

## Cose da non toccare

- `prisma/migrations/` mai modificato a mano, solo via CLI.
- `.env.production` mai committato.
- File in `src/legacy/` da deprecare, non aggiungere codice lì.

## Stile commit

![](img/L06/L06_10_quattro_strumenti.png)
```

`Conventional Commits: feat:, fix:, docs:, chore:, refactor:.`

Vent'una righe. Cinque minuti per scriverlo. Cambia tutto.

C'è un comando rapido per generarlo da zero: `/init`. Claude scansiona il progetto, capisce lo stack, propone una bozza di CLAUDE.md. Tu la rivedi, la pulisci, la commiti. Da quel momento *ogni* sessione di Claude Code parte già allineata.

Una cosa importante: **CLAUDE.md va aggiornato.** Quando aggiungi una libreria importante, quando cambi un'abitudine, quando deprecate una cartella — aggiungilo. È memoria viva, non un README dimenticato. Se l'AI ti propone codice che ignora una convenzione, è probabile che la convenzione non sia in CLAUDE.md. Aggiungila, e da domani lei la rispetta.

Skills: le competenze caricate quando servono

CLAUDE.md è sempre attivo. Ma se ci metti dentro ogni regola dettagliata che esiste, diventa lungo, dispersivo, e occupa finestra di contesto inutilmente in sessioni che non c'entrano niente con quella regola.

Soluzione: **Skills**. Una skill è una cartella con dentro un file `SKILL.md` (più eventuali script, template, riferimenti) che descrive una **competenza specifica**. Claude carica la skill solo quando la richiesta dell'utente la richiama, in base alla sua descrizione.

Esempio: una skill chiamata `pdf-handling` con dentro un SKILL.md che spiega come lavorare con i PDF (quali librerie usare, come estrarre testo, come gestire OCR). Quando tu chiedi «aiutami a estrarre testo da questi PDF», Claude vede la skill, la carica, applica le sue regole. Quando chiedi tutt'altro, la skill non si carica e non sporca il contesto.

Tre cose da sapere sulle skills:

La descrizione è quello che le fa scattare. Se scrivi una descrizione vaga («fa cose con i file»), la skill non parte mai al momento giusto. Descrizione precisa, con verbi e contesti: «Crea, leggi, modifica, e fa OCR su file PDF».

Le skills hanno cartelle annesse. Sotto `SKILL.md` puoi mettere `scripts/`, `references/`, `assets/`. Claude apre questi sotto-file solo quando glielo dice il SKILL.md, e gli script li *esegue*, non li legge — quindi non sprecano context. È una potenza enorme: una skill può «sapere» 50 pagine di documentazione senza che mai entrino tutte nel context.

Le skills si condividono. Cartella `.claude/skills/` nel progetto, commit in Git, tutto il team ha la skill. Oppure cartella `~/.claude/skills/` per le tue personali, valgono in ogni progetto.

Esempio pratico: una skill `report-financial` che insegna a Claude come si scrive un report finanziario aziendale nel formato che usate. Quando tu chiedi «fammi il report del trimestre», parte. Quando chiedi «fissa questo bug», non parte. Pulizia.

Subagents: la delega in finestra separata

Hai un task complesso da fare. Esempio: «esplora un codebase di duecentomila righe e trovami dove gestiamo i refund». Se lo fai nel thread principale, Claude legge quindici file, ognuno entra in context, e dopo dieci minuti la finestra è piena di file letti per arrivare a una risposta di due righe.

I **subagents** risolvono esattamente questo. Un subagent è un Claude figlio che parte con un **nuovo system prompt**, riceve dal Claude padre un task scritto, lavora in una **finestra di contesto separata**, e alla fine restituisce *solo un riassunto* al padre. La sua intera conversazione viene scartata.

Cosa significa in pratica:

- Il padre ha mantenuto il context pulito.
- Tu hai la risposta sintetica che ti serviva.
- Hai perso la visibilità su tutti i passi intermedi del figlio (è il trade-off).

Claude Code ha **subagent built-in** già pronti:

- **general-purpose** — task multi-step che richiedono esplorazione e azione.
- **Explore** — ricerche veloci nel codebase.

- **Plan** — preparazione del piano in plan mode.

Li chiami a voce: «usa il subagent Explore per trovare dove gestiamo i refund».

Puoi anche **crearti subagent custom**. Comando `/agents` → «crea nuovo agente». Scegli scope (progetto o personale), descrivi lo scopo, scegli quali strumenti gli dai, dai un colore visivo. Claude genera la configurazione.

Esempio: un subagent `code-reviewer` con accesso solo agli strumenti di lettura (`Read`, `Grep`, `Glob`), system prompt focalizzato su sicurezza e qualità, descrizione «recensisce pull request e file modificati». Lo richiami con «fai una review del codice appena scritto». Lui legge, scrive un report, torna al padre. Nessuna modifica ai file (perché non ha gli strumenti per farla). Pulito.

Tre pattern che vedi spesso nei team:

- **Reviewer read-only**: solo `Read`, `Grep`, `Glob`. Identifica problemi, non li corregge.
- **Researcher**: aggiungi `WebSearch`. Cerca documentazione, riporta sintesi.
- **Implementer focalizzato**: aggiungi `Edit`, `Bash`. Per task isolati grossi che ti gonfierebbero il context principale.

Una variante avanzata: il subagent può **preload skills** (caricare di colpo nel suo context una o più skill già pronte) e avere **memoria persistente** tra le sessioni. Ma per ora bastano i fondamentali.

Hooks: le regole che devono scattare sempre

CLAUDE.md è una linea guida. Le skills sono competenze. I subagent sono deleghe. I **hooks** sono ordini.

Un hook è uno script che parte automaticamente in risposta a un evento. I due eventi principali:

- **PreToolUse** — scatta *prima* che Claude usi uno strumento. Se l'hook esce con exit code 2, l'azione viene **bloccata**. Lo stderr torna a Claude come spiegazione.
- **PostToolUse** — scatta *dopo* che Claude ha usato uno strumento. Tipicamente per pulizia, formattazione, logging.

Esempi concreti:

PostToolUse per il formatter. Ogni volta che Claude modifica un file `.ts`, parte uno script che esegue `prettier --write` su quel file. Risultato: il codice è sempre formattato, sempre. Non è una richiesta a Claude di formattare — è una regola di sistema che applichi tu.

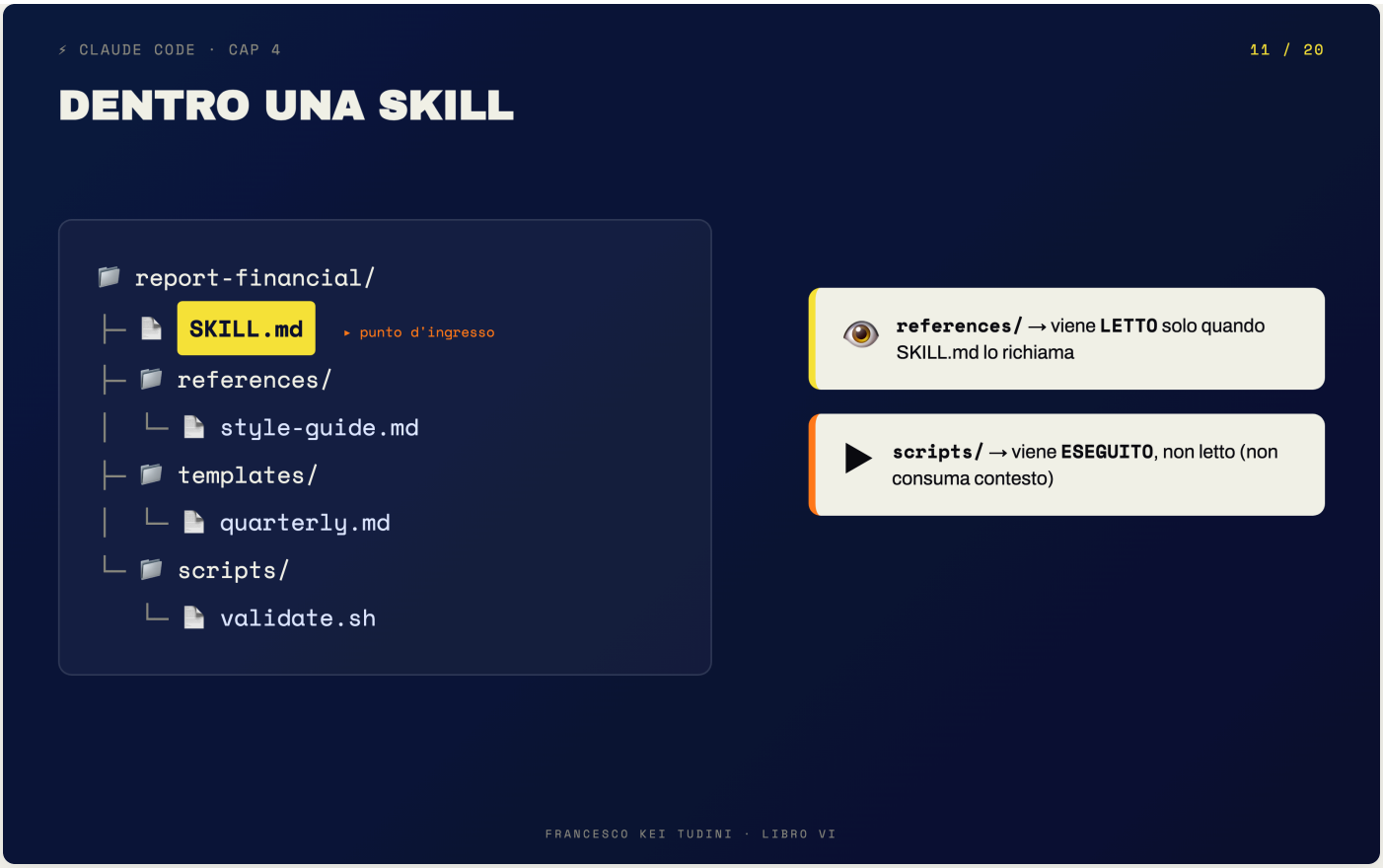
PreToolUse per bloccare comandi pericolosi. Ogni volta che Claude prova a lanciare un comando bash, il tuo hook controlla la stringa. Se contiene `rm -rf` su una directory critica, exit code 2, comando bloccato, Claude riceve indietro un errore tipo «non puoi cancellare questa cartella, ragiona diversamente». Claude *ragiona diversamente* — non riprova lo stesso comando.

PreToolUse per bloccare push su main. Lo stesso schema. Se il comando è `git push origin main`, blocca. Forzi tutto il team a passare da pull request.

Si configurano in `.claude/settings.json` (o via comando `/hooks` interattivo). Si committano in Git, tutto il team li eredita.

La regola d'oro che ripetono nei materiali Anthropic è questa: **se qualcosa deve succedere ogni volta senza eccezioni, non metterlo in un prompt. Mettilo in un hook.** I prompt sono suggerimenti. Gli hook sono leggi.

Mettiamo tutto insieme: una configurazione pro-tipo



Un setup serio in un progetto reale ha tutti e quattro:

- **CLAUDE.md** con stack, convenzioni, comandi (sempre on).
- **Skills** per le competenze specifiche del dominio (es. una skill `invoice-format` per come si formattano le fatture, una `report-style` per il tono dei report).
- **Subagents custom** per i task ripetitivi pesanti (un `code-reviewer` per ogni PR, un `data-explorer` per analizzare CSV in cartelle separate).
- **Hooks** per le regole non negoziabili: formatter dopo ogni edit, blocco su comandi pericolosi, log di tutto in un file di audit.

Il giorno in cui aggiungi tutti e quattro questi pezzi, smetti di «provare Claude Code» e cominci a **lavorare con Claude Code**. È una differenza qualitativa.

Errori da non fare

Mettere tutto in CLAUDE.md. CLAUDE.md è la memoria di progetto sempre on. Se ci ficchi dentro centocinquanta righe di regole specifiche su come fare un report finanziario,

sporchi il context di ogni sessione, anche quando stai lavorando su tutt'altro. Per le cose specifiche di un dominio, usa skills.

Skills con descrizione vaga. Se la descrizione di una skill è «aiuta con i documenti», Claude non sa quando attivarla. Descrizione precisa e operativa: «Crea, legge, modifica file Excel e CSV per analisi finanziaria».

Subagent senza limiti sugli strumenti. Un subagent reviewer che ha gli strumenti `Edit` e `Bash` può scrivere e cancellare codice mentre dovrebbe solo recensirlo. Restringere gli strumenti è metà del lavoro di sicurezza.

Hook che fanno troppe cose insieme. Un hook lento blocca il flusso di Claude per secondi a ogni azione. Un hook deve essere veloce e fare una cosa sola. Per la pulizia complessa, lancia un subagent.

Cosa porti a casa

- **Quattro strumenti, quattro funzioni.** CLAUDE.md = memoria. Skills = competenze a richiesta. Subagents = deleghe in context separato. Hooks = regole inviolabili.
- **CLAUDE.md è la prima cosa.** Crealo con `/init`, puliscilo, tienilo aggiornato. Cambia tutto.
- **Skills per la specializzazione.** Solo quello che serve in alcuni momenti, descritto in modo preciso.
- **Subagents per il contesto pulito.** Quando un task gonfierebbe la finestra principale, deleghi a un figlio.
- **Hooks per le regole non negoziabili.** «Deve succedere sempre» = hook, non prompt.

Capitolo 5 — Best practice di engineering: come lavorano i pro

Tra il setup curato e quello «alla buona», quanto cambia la performance?

Secondo i test pubblicati da Anthropic Engineering nel post «Quantificare il rumore infrastrutturale nelle eval di coding agentico» (Gian Segato, 5 febbraio 2026), la differenza misurata su benchmark di coding agentico — nello specifico Terminal-Bench 2.0 — è stata **fino a 6 punti percentuali** tra setup con risorse ristrette e setup con risorse abbondanti. Sei punti. Più del divario fra i modelli top in classifica. Stesso modello, stesso task, configurazione diversa: il punteggio cambia di sei punti.

Tradotto: se tu usi Claude Code con un'impalcatura tecnica approssimativa e il tuo collega lo usa con un'impalcatura curata, lui prende dal modello cose che a te non riesce. Non è il modello che cambia. Cambia *quanto bene è stato messo in condizione di lavorare*.

Le best practice di engineering che vediamo in questo capitolo sono esattamente quelle che spostano quei sei punti.

Test-driven da subito

La prima best practice è anche la più semplice: **dare a Claude una suite di test che possa lanciare in autonomia**.

Il principio è banale. Claude scrive del codice. Lui stesso esegue i test. Se passano, va avanti. Se falliscono, vede l'errore, capisce la causa, corregge, rilancia. Loop chiuso. Senza che tu debba dire una parola.

Senza test, il loop si rompe. Claude propone una modifica. Tu la accetti. Lanci i test a mano. Falliscono. Glielo dici. Lui prova una correzione. Approvi. Lanci. Fallisce ancora. Dieci minuti di ping-pong su un bug che con una suite di test girava in trenta secondi.

Cosa significa «test-driven» con Claude:

1. Se hai già test, mettili in CLAUDE.md con il comando per lanciarli (`npm test`, `pytest`, `cargo test`).
2. Se non hai test, **chiedi a Claude di scriverli prima del codice**. Esempio: «scrivi i test per la funzione che parserà i CSV. Falli passare con una funzione fittizia, poi scriveremo la funzione vera».
3. Lascia a Claude i permessi per eseguire la suite (allow `Bash(npm test)`).

Risultato: ogni modifica viene verificata. Le regressioni si vedono subito. Tu fai meno il revisore manuale e più il decisore.

Iterazione visuale: screenshot e mockup

Se stai lavorando su qualcosa che ha un'interfaccia — pagina web, app mobile, dashboard — un trucco da pro è **iterare con immagini, non solo con parole**.

Funziona così: tu descrivi a parole la modifica. Claude la implementa. Tu fai uno screenshot. Lo carichi nella conversazione. Scrivi: «ecco com'è uscita. Il pulsante deve essere allineato a destra, non a sinistra». Claude vede l'immagine, capisce il problema spazialmente, corregge.

Funziona anche per costruire ex novo: gli dai un mockup (anche disegnato a mano, fotografato) come allegato, dici «implementa questo». Claude legge l'immagine, capisce

il layout, scrive l'HTML/CSS/React corrispondente. Una iterazione visiva equivale spesso a venti righe di descrizione testuale.

Anthropic stessa raccomanda questo flusso nel materiale ufficiale. Lo trovi nell'articolo «Claude Code: best practices» del blog engineering. Funziona perché le interfacce sono fatte di rapporti spaziali, e i rapporti spaziali si descrivono meglio con un'immagine che con mille parole.

Headless mode: Claude Code in pipeline

Tutto quello che hai visto finora è interattivo: tu davanti al terminale, Claude che ti chiede permessi. Ma Claude Code ha anche una modalità **headless** — senza interfaccia, programmabile, da inserire in pipeline.

Il comando base è `claude -p`:

```
claude -p "review this PR for security issues" --output-format json
```

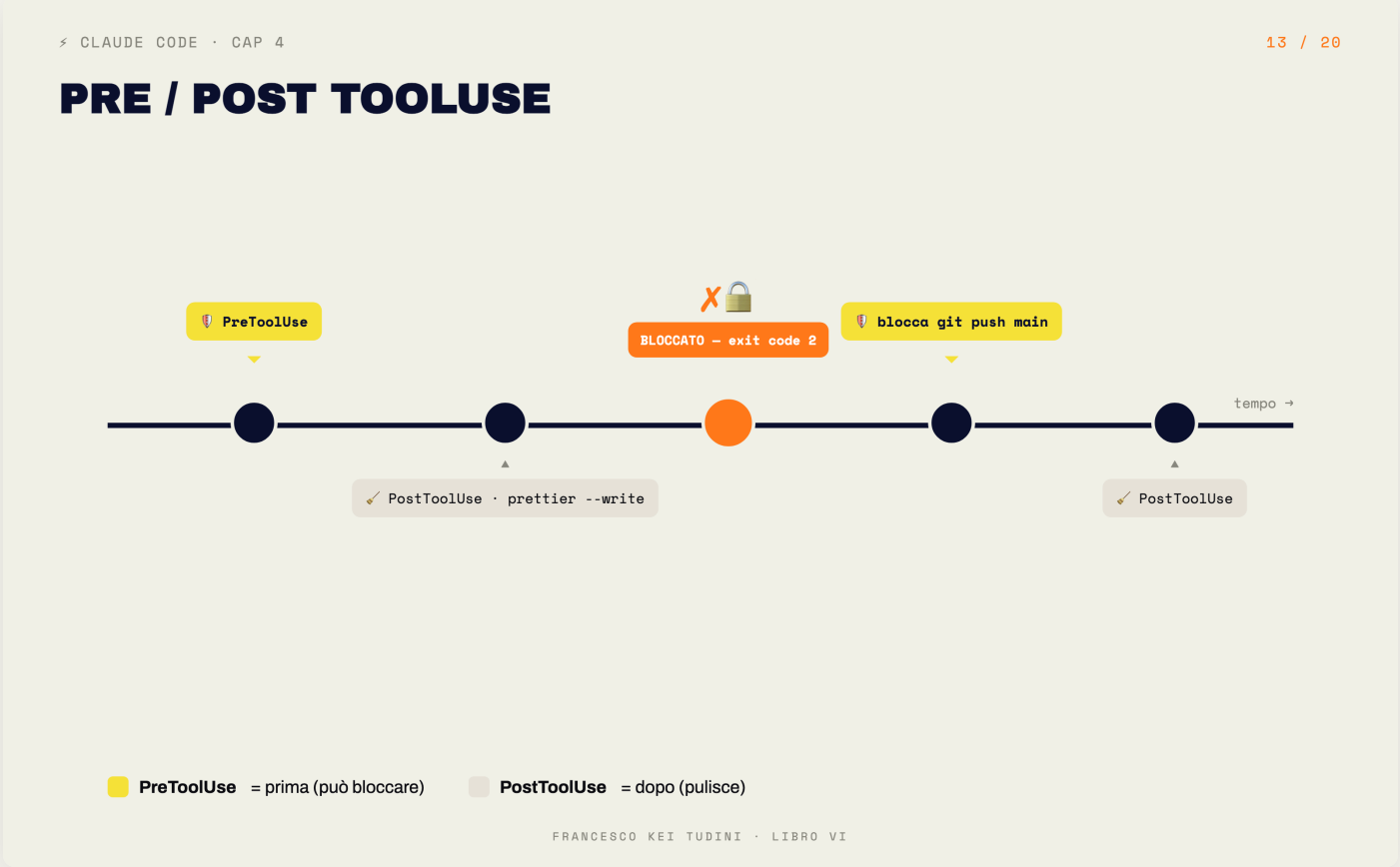
Tradotto: «Claude, fai questa cosa, dammi l'output in JSON, esci». Nessuna interattività. Stampa il risultato. Termina. Pronto per essere parsato da uno script.

A cosa serve in pratica:

- **CI/CD.** Ogni volta che apri una pull request, una pipeline lancia Claude in headless per fare una review automatica. Restituisce un commento sulla PR.
- **Batch processing.** Hai 200 file Markdown da tradurre. Uno script lancia Claude su ciascuno in headless. Risultato: 200 traduzioni in serie senza che tu apra il terminale 200 volte.
- **Integrazioni custom.** Chiunque sappia scrivere uno script può chiamare Claude come fosse una libreria — input, output, fatto.

L'headless mode è il momento in cui Claude Code smette di essere uno strumento individuale e diventa un pezzo dell'infrastruttura del team.

Multi-Claude: più istanze in parallelo



Se hai un task grosso, puoi farlo girare a un Claude solo, in serie. Oppure puoi dividerlo a Claude diversi che lavorano in parallelo. La seconda strada è quella dei pro.

Esempio. Devi aggiornare le dipendenze di sicurezza in tre microservizi diversi. Tre task isolati, indipendenti. Strada vecchia: un Claude, tre task in fila, due ore. Strada multi-Claude: tre terminali aperti, ognuno con un Claude su un microservizio, lavorano insieme, quaranta minuti.

Anthropic chiama questo pattern **multi-Claude workflow**. Funziona quando i task sono **isolati** (non condividono file) e **deterministici** (non c'è scelta interpretativa da coordinare). Su queste due condizioni, paralleli vincono sempre.

Su Mac o Linux è banale: aprire tre finestre del terminale, lanciare `claude` in tre cartelle diverse. Su Windows + WSL stesso flusso. Nell'app desktop di Claude puoi anche far girare più sessioni di Cowork in parallelo.

Per task non isolati, multi-Claude diventa rischioso: due Claude che toccano lo stesso file in contemporanea si pestano i piedi. La regola: parallelo solo su silos separati.

Gestione del contesto: i tre comandi a memoria

Hai una sessione che sta diventando lunga. Hai letto venti file. Hai modificato sei. Il context si sta riempiendo. Tre comandi da sapere a memoria:

`/context` — ti mostra lo stato attuale. Quante token hai usato, quali categorie pesano di più (conversazione, file letti, output di comandi, CLAUDE.md, risultati di tool). Una grafica chiara di cosa sta dentro.

`/compact` — Claude legge tutta la conversazione, la riassume, butta via i dettagli secondari, lascia solo quello che serve. Ti restituisce molto spazio. Costo: perdi alcuni dettagli intermedi. Quando farlo: stai lavorando su una feature e *vuoi continuare* sulla stessa con più spazio.

`/clear` — buttala via tutta, ricominci da zero. Quando farlo: hai finito una feature e ne cominci una nuova che non c'entra. È pulizia totale.

La regola pratica: **compact quando vuoi continuare lo stesso lavoro con più spazio. Clear quando cambi lavoro.** Per le cose che vuoi che Claude ricordi *sempre*, mettile in CLAUDE.md. CLAUDE.md non si perde col compact né col clear.

Mai il `/clear` quando hai un piano in pancia

Una sottigliezza che frega anche chi è esperto: se hai appena finito una fase di Plan Mode con un piano dettagliato, *non* fare `/clear`. Cancelli il piano. Devi rifarlo.

Se vuoi liberare spazio prima di cominciare l'esecuzione del piano, usa `/compact` — il riassunto preserva il piano come parte importante. Oppure, ancora meglio, **scrivi il piano in un file `PLAN.md` nel progetto**: a quel punto è su disco, non in context. Claude può rileggerselo all'occorrenza.

Questo è un esempio del principio generale: **se una cosa è importante, mettila su disco, non in context**. Il context è memoria volatile; il disco è memoria persistente.

La regola d'oro dell'engineering con AI

La riassumo in una frase. **Se deve succedere sempre senza eccezioni, è un hook. Se è una linea guida che vuoi sia rispettata di solito, è in CLAUDE.md. Se è una competenza specifica di un dominio, è una skill. Se è un compito grosso che gonfierebbe il context principale, è un subagent.**

Quattro tipi di «memoria» ben divisi. Quattro strumenti, quattro slot. Quando devi aggiungere una nuova regola al tuo setup, fermati trenta secondi e chiediti: in che slot va? Se è chiaro, agisci. Se non è chiaro, ti stai inventando una categoria nuova — e probabilmente stai sovra-ingegnerizzando.

Questa disciplina mentale, da sola, ti porta nel 10% degli utenti di Claude Code che usano davvero il sistema. Gli altri mettono tutto in CLAUDE.md finché diventa illeggibile. O peggio, scrivono prompt da venti righe ogni volta ricostruendo a mano regole che dovrebbero essere strutturate.

Esempi pratici di flusso pro

Uno sviluppatore freelance solo. CLAUDE.md per ogni progetto cliente. Skills personali (`~/.claude/skills/`) per le competenze ricorrenti: «scrivi report PR-style», «documenta API in OpenAPI». Hooks per il formatter (prettier dopo ogni edit). Subagent `code-reviewer` da chiamare prima di ogni commit.

Una piccola software house, 5 persone. CLAUDE.md committato nel repo, condiviso. `.claude/settings.json` con permessi e hook condivisi. Skills di team in `.claude/skills/` (versionate in Git). Subagent custom per la code review pre-merge. Pipeline CI con Claude headless per audit di sicurezza.

Un imprenditore non tecnico che ha cominciato a buttare giù tool interni. CLAUDE.md minimale ma chiaro sul progetto. Niente skills custom (non gli servono ancora). Subagent built-in `Explore` quando vuole capire un suo vecchio codice. Hook PreToolUse che blocca `rm -rf` e push su main, scritto con l'aiuto di Claude stesso.

Errori da non fare

Saltare i test perché «sono semplice modifica». Le semplici modifiche sono le più infide. Una suite di test ti protegge da regressioni che a occhio non vedresti.

Non usare headless dove ne avresti bisogno. Se la stessa cosa la stai facendo a mano la quinta volta, è un automatismo headless. Quel quarto d'ora che ci metti a scriverlo si ripaga in due giorni.

Multi-Claude su task accoppiati. Due Claude sullo stesso file = caos. Multi-Claude solo su silos veri.

Tenere tutto in context invece che su disco. Se ti accorgi che stai facendo `/compact` ogni dieci minuti, mancano CLAUDE.md, mancano skills, manca documentazione su disco. Il context non è l'archivio del progetto.

Cosa porti a casa

- **Sei punti di performance valgono l'impalcatura tecnica.** Il modo in cui imposti il setup vale come scegliere un modello migliore.

- **Test-driven non è un dogma, è una leva.** Più Claude può verificare da solo, meno tu devi correggere a mano.
- **Iterazione visuale per le UI.** Screenshot e mockup valgono mille parole.
- **Headless = Claude in pipeline.** CI/CD, batch, integrazioni custom.
- **Multi-Claude solo su silos isolati.** Parallelo vince, ma con regole.
- **Tre comandi del contesto:** `/context`, `/compact`, `/clear`. A memoria.

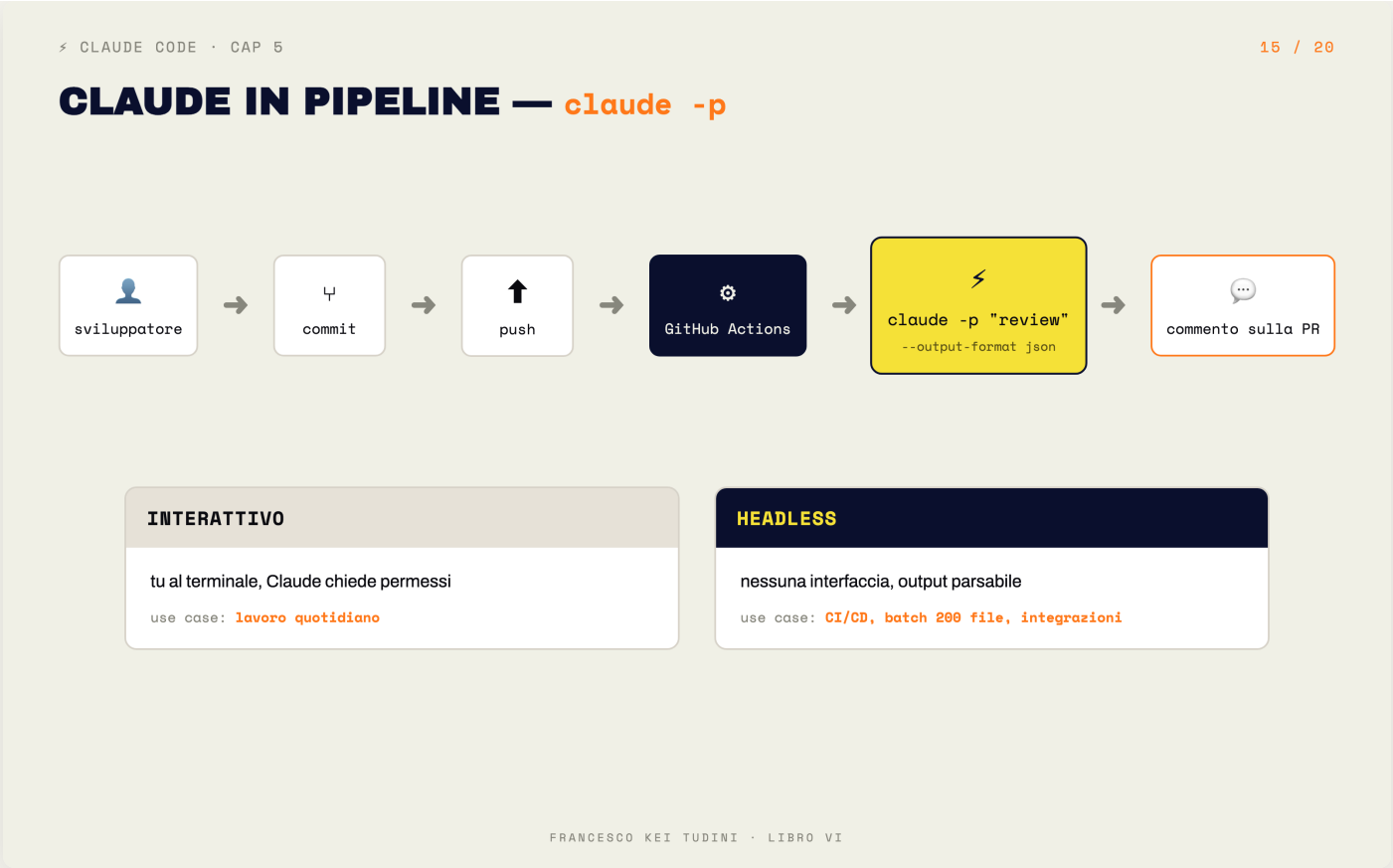
Capitolo 6 — Programmare anche se non sai programmare

«Non so programmare» è un'identità, non una sentenza.

E in ogni caso, da quando esiste Claude Code, è anche un'identità che vale molto meno di quanto valeva tre anni fa. Cose che dieci anni fa richiedevano un team — automatizzare un foglio di calcolo che diventa report mensile, montare un sito interno che gestisce ordini, scrivere uno script che parsi le email dei clienti e tiri fuori i dati — oggi le fai in un weekend. Senza aver mai scritto una riga di codice prima.

Lo dico chiaro perché è la promessa centrale di questo capitolo, e non voglio venderti aria fritta: non ti sto dicendo che diventerai uno sviluppatore. Ti sto dicendo che puoi **far costruire software che ti serve, senza essere uno sviluppatore**. È diverso. Il codice esiste, gira, fa il suo lavoro. L'autore principale è Claude. Tu sei il committente, il revisore, il decisore. Il software è tuo. La firma è tua.

Cosa significa davvero «programmare senza programmare»



Te lo riassumo in tre verbi.

Tu descrivi. A parole, in italiano, cosa vuoi che faccia il programma. «Voglio uno script che legga questo CSV di lead, normalizzi i numeri di telefono italiani (aggiunga +39 se manca, levi gli spazi), elimini i duplicati per email, e mi salvi il risultato in un nuovo CSV pulito».

Claude scrive. Genera il codice. Crea i file. Installa le librerie necessarie. Ti fa girare lo script su un esempio.

Tu verifichi. Apri il CSV pulito. Vedi se ha senso. Se ci sono righe strane, dici «questa riga qua è sbagliata, controlla», e Claude indaga. Quando il risultato funziona su esempi reali, hai uno strumento.

Non hai capito il codice. Non hai imparato Python. Hai però **ottenuto il risultato**. Per il 90% delle automazioni interne di un'impresa media, è quello che serve.

Da dove cominciare: tre progetti per un weekend

Ti propongo tre progetti tipici, tutti fattibili da chi non programma, tutti utili da subito.

Progetto 1: pulizia automatica di file e cartelle. Hai una cartella «Scaricati» con dentro 4.000 file degli ultimi sei anni. Vuoi uno script che li sposti in sottocartelle per anno e tipologia (PDF in una, immagini in un'altra, fogli Excel in un'altra), che cancelli i duplicati esatti, che mantenga i file più recenti. Tempo per costruirlo con Claude: due ore. Tempo che ti risparmia per sempre: ore di pulizia manuale ogni volta che dici «devo riordinare il computer».

Progetto 2: un'app interna per gestire una lista di clienti. Hai un foglio Excel dei clienti che è diventato impraticabile (50 colonne, 800 righe, mille colori). Vuoi un'app web semplice — gira sul tuo computer, accessibile dal browser, una pagina dove cerchi un cliente, una dove inserisci uno nuovo, una dove fai modifiche. Database locale, niente in cloud. Tempo con Claude: un sabato pomeriggio. Strumento utile per anni.

Progetto 3: un mini-tool che fa una cosa che fai sempre. Esempio: ogni lunedì copi-incolli i dati di vendita della settimana da un foglio Excel a un report Word, formatti, salvi come PDF, mandi via email a tre persone. Tre ore ogni lunedì, da anni. Vuoi uno script che apra il file Excel di vendita, estragga i numeri, generi il Word con il formato standard, lo trasformi in PDF, lo alleghi a un'email pronta da inviare. Tempo con Claude: un pomeriggio per costruirlo, dieci minuti di test ogni lunedì successivo. Risparmio: due ore e mezza alla settimana, per sempre.

Tre progetti, tre weekend al massimo, tre strumenti che funzionano. Nessun corso di programmazione. Solo Claude Code, un computer, un po' di pazienza.

Come si parla a Claude se non sai programmare

Tre regole, in italiano.

Regola uno: descrivi il *cosa*, non il *come*. Tu non sai come si fa. Non sforzarti di indovinare la libreria, il linguaggio, il file. Descrivi cosa vuoi che succeda. Esempio cattivo: «scrivi uno script Python con pandas che faccia drop_duplicates su email». Esempio buono: «voglio uno script che, dato un CSV di clienti, mi tolga le righe duplicate considerando uguale chi ha la stessa email».

Claude sceglie linguaggio, libreria, struttura. Tu controlli il risultato.

Regola due: parti dal piccolo. Non chiedere mai «fammi una piattaforma completa per la gestione dei clienti» al primo prompt. Cominci con un pezzo. «Fammi una pagina web che mostri questa lista di nomi». Quando funziona, aggiungi: «adesso fammi aggiungere un nuovo nome con un form». Quando funziona, aggiungi modifica. Eliminazione. Ricerca. Crescita per incrementi. Ogni incremento lo verifichi prima di chiedere il successivo.

Regola tre: pretendi spiegazioni in italiano. Ogni volta che Claude fa qualcosa che non capisci, fermati e chiedi: «spiegami in italiano semplice cosa fa questa parte di codice. Senza tecnicismi, come lo spiegheresti a un amico imprenditore». Lui te lo spiega. Tu impari quel pezzo. Dopo un mese, hai una mappa mentale di cosa fa cosa — non sai scriverlo, ma sai leggerlo. Sufficiente per essere il committente lucido.

Cosa non puoi delegare comunque

Te lo dico ora, non alla fine, perché è cruciale: ci sono tre cose che restano tue anche se non sai programmare.

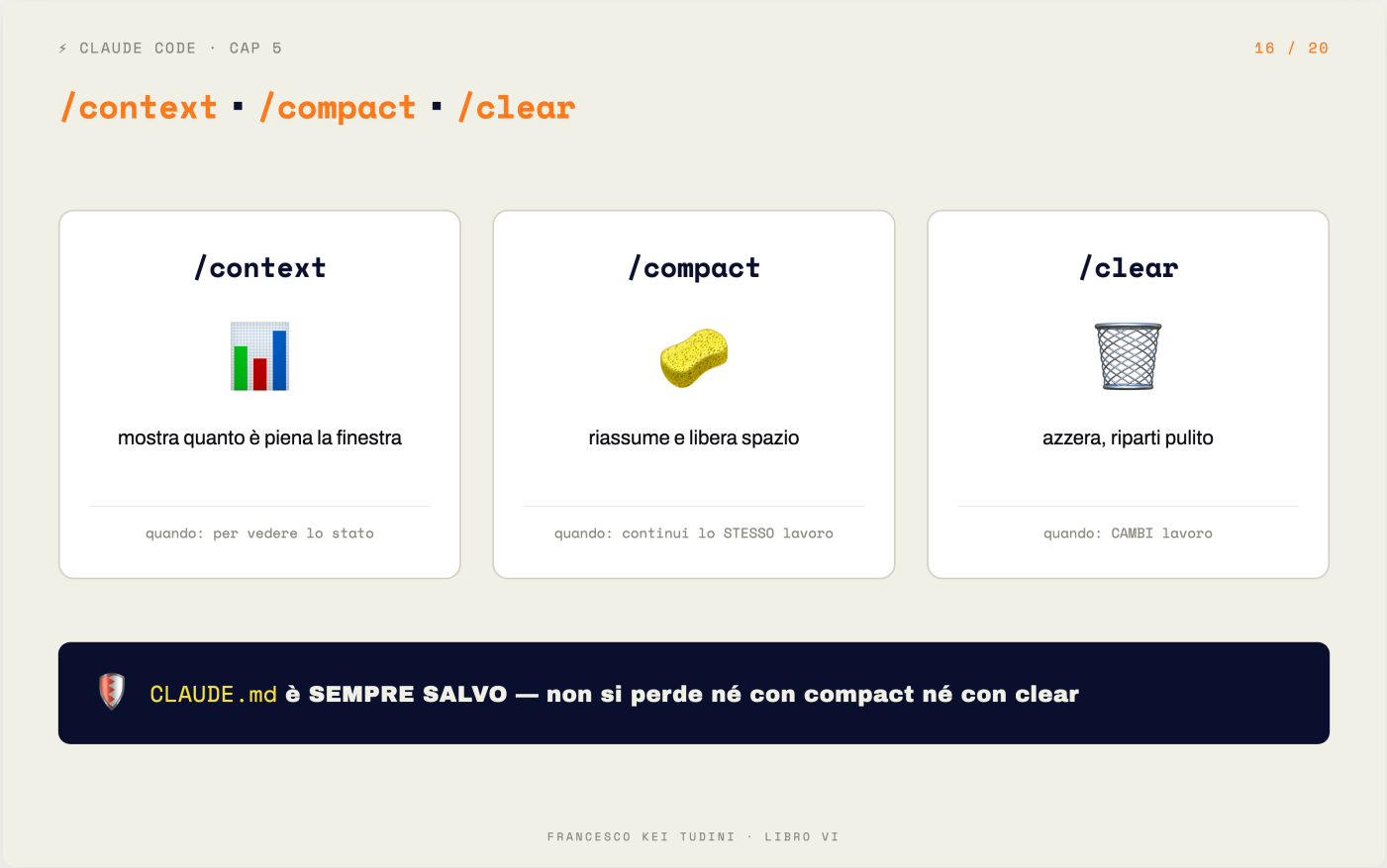
Capire l'output. Lo script ha pulito il CSV. Tu apri il file e *guardi*. Le righe ti sembrano coerenti? I numeri di telefono hanno tutti il +39? I duplicati sono spariti? Se non guardi, ti fidi di una cosa che non sai cosa fa. Il «non so programmare» non è una scusa per non *leggere* il risultato. Il risultato è in italiano: una lista di clienti, una tabella di vendite, un report. Lo sai leggere. Leggilo.

Capire gli errori. Quando qualcosa va storto, Claude ti dice perché. Spesso in italiano. «Il file CSV ha una colonna chiamata `e-mail` mentre lo script cercava `email`. Cosa vuoi fare?». A questo punto sei tu che decidi: rinomino la colonna nel file? Modifico lo script? Faccio uno script più robusto che riconosce entrambe? Tre scelte. Tutte ragionevoli. Devi prenderne una.

Verificare prima di fare cose irreversibili. Se lo script cancella file, *spostalo prima in una cartella di backup* e fai una prova lì. Se modifica un database, *fai un dump* prima. Se manda email, prima fai un test con il tuo indirizzo. Le cose irreversibili — cancellazioni,

modifiche di massa, invii — vanno sempre provate su una copia. Sempre. Senza questa abitudine, un giorno Claude esegue alla lettera la cosa che gli hai chiesto e ti accorgi di averla chiesta male solo quando i file non ci sono più.

La Diligenza in pratica



Torniamo alla **Diligenza** — la quarta D, quella della responsabilità. In Claude Code, per chi non sa programmare, la Diligenza è cruciale e ha tre regole spicce:

1. **Il codice gira sul tuo computer. La firma è tua.** Se manda un'email sbagliata a un cliente, l'errore è tuo, non di Claude. Se cancella un file importante, idem.
2. **Ogni cosa che modifica dati va testata su una copia.** Sempre. Prima i test su una copia, poi (mai prima) sulla cosa vera.
3. **Tieni traccia di cosa fanno i tuoi strumenti.** Un piccolo file `STRUMENTI.md` nel tuo computer dove scrivi: «questo script fa X, vive nella cartella Y, l'ho creato il giorno Z, l'ultima modifica è del giorno W». Sei mesi dopo non te lo ricordi. Quel file ti salva.

Senza queste tre cose, programmare senza programmare diventa pericoloso. Con queste tre cose, è una superpotenza.

Esempi pratici

Un imprenditore non tecnico che vende olio. 200 clienti in Excel, l'ottanta percento dei riordini sono di clienti vecchi. Vuole uno strumento che ogni mese gli dica chi non ha ordinato da più di 90 giorni, per chiamarli. Con Claude Code, in un sabato, costruisce uno script che legge l'export delle vendite da Shopify, incrocia con il foglio clienti, sputa una lista «da chiamare» ordinata per valore cliente. Tempo di costruzione: 4 ore. Tempo risparmiato per sempre: 2-3 ore al mese di analisi manuale.

Una commercialista con uno studio piccolo. Riceve fatture in PDF dai clienti. Le legge una a una, copia partita IVA, importo, data in un foglio. Cinquanta fatture al mese, due minuti l'una = quasi due ore. Con Claude Code, costruisce in un weekend uno script che apre tutti i PDF in una cartella, estrae i dati con un parser, li scrive in un foglio Excel. Tempo: 4 ore costruzione. Risparmio: 100 minuti al mese. Restituzione dell'investimento in due mesi.

Uno studente di Architettura. Vuole un piccolo sito personale per il portfolio. Non vuole pagare 800 euro a un web designer. Con Claude Code, in un weekend, costruisce un sito statico in HTML/CSS, con una galleria immagini, un curriculum, una pagina di contatti. Lo carica su un servizio gratuito (Netlify, GitHub Pages — Claude gli spiega come). Tempo: 6 ore di lavoro. Risultato: sito online che fa quello che lui voleva, in libertà totale.

Errori da non fare

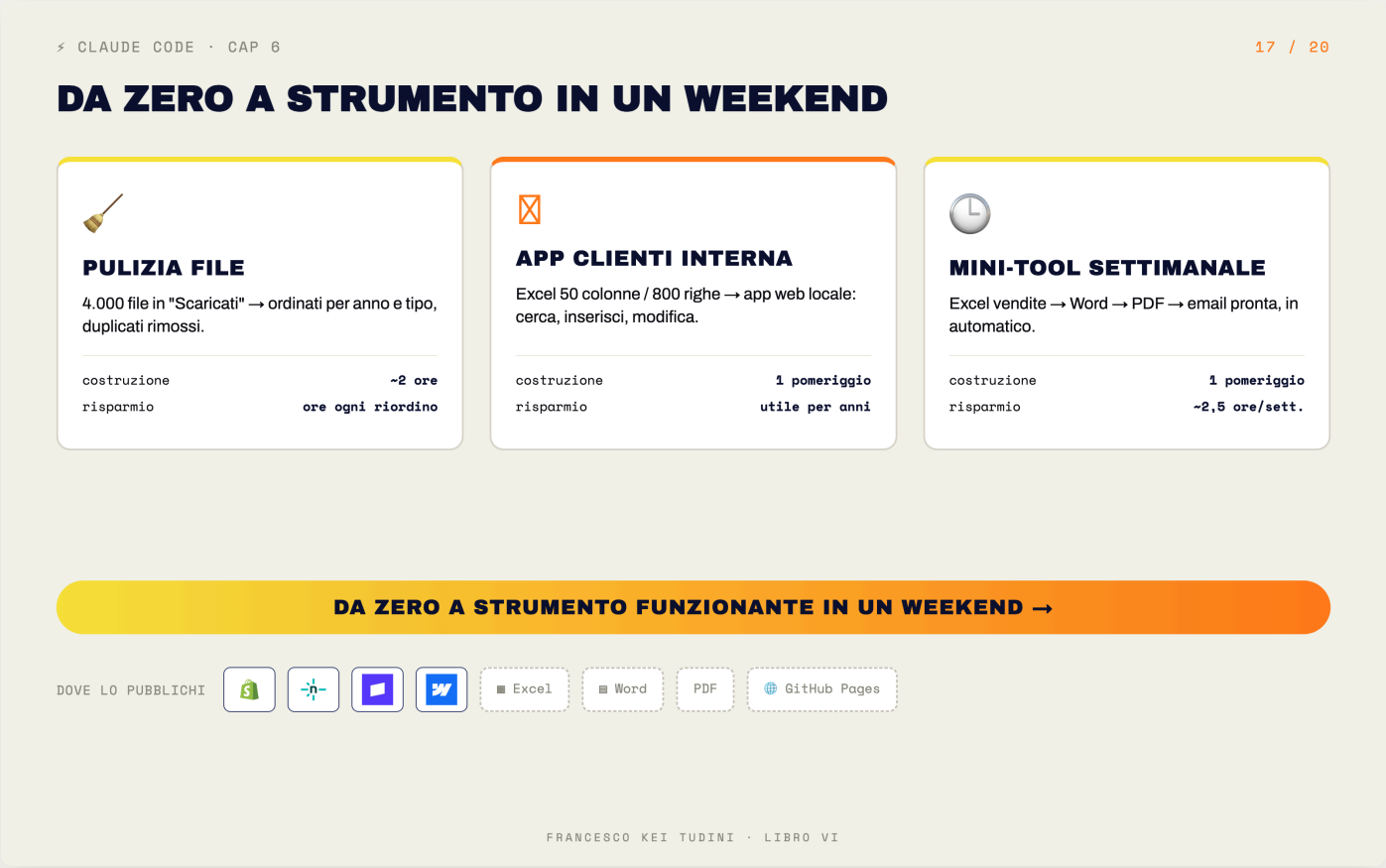
Pretendere il prodotto completo al primo prompt. «Voglio un CRM completo» è un prompt da cento ore di lavoro. Cominci piccolo. Una pagina. Una funzione. Un caso d'uso. Cresci da lì.

Non leggere il codice nemmeno una volta. Anche se non lo capisci tutto, *guardarlo* aiuta. Vedi i nomi delle funzioni. Vedi cosa Claude commenta. Vedi i pattern. Dopo dieci sessioni, una parte la capisci. Dopo cento, sei sorpreso di quanto leggi.

Saltare i test su una copia per le cose irreversibili. Già detto, lo ripeto. Cancellazioni, invii, modifiche di massa: prima su copia. Sempre.

Non commentare cosa fanno i tuoi tool. Sei mesi dopo non sai cosa hai costruito. Una riga di descrizione nel file, in italiano, prima della prima funzione: «Questo script pulisce il CSV dei lead della landing page. Lo usa Maria ogni venerdì».

Cosa porti a casa



- **«Non so programmare» vale molto meno di prima.** Con Claude Code costruisci strumenti veri senza saper scrivere codice.
- **Tre verbi: tu descrivi, Claude scrive, tu verifichi.** Funziona per il 90% delle automazioni utili a una piccola impresa.
- **Parti dal piccolo.** Una pagina, una funzione, un caso d'uso. Cresci per incrementi.
- **Pretendi spiegazioni in italiano.** Non per imparare a programmare, ma per essere un committente lucido.
- **La firma resta tua.** Capire l'output, capire gli errori, testare su copia. Sempre.

Capitolo 7 — I sette errori del primo mese

Gli errori non sono dell'AI. Sono del flusso.

Lo dico così, dritto, perché è la frase più importante di questo libro: quando Claude Code fa qualcosa di stupido o di pericoloso, nove volte su dieci è il flusso che è sbagliato, non il modello. E il flusso, lo controlli tu.

Qui sotto, i sette errori che vedo più spesso nei primi trenta giorni. Se li conosci prima di farli, li eviti. Se li fai senza averli letti, li scopri quando paghi il prezzo.

Errore 1 — Auto-accept su tutto, il primo giorno

Lo abbiamo già accennato. Il giorno uno, vedere i diff è didattica pura: capisci come Claude pensa, come scrive, dove tende a sbagliare. Se attivi YOLO mode il primo giorno, salti l'apprendistato e cominci a fidarti di un collega di cui non hai mai letto una riga di codice. Risultato: bug strani in due settimane, senza sapere da dove vengano.

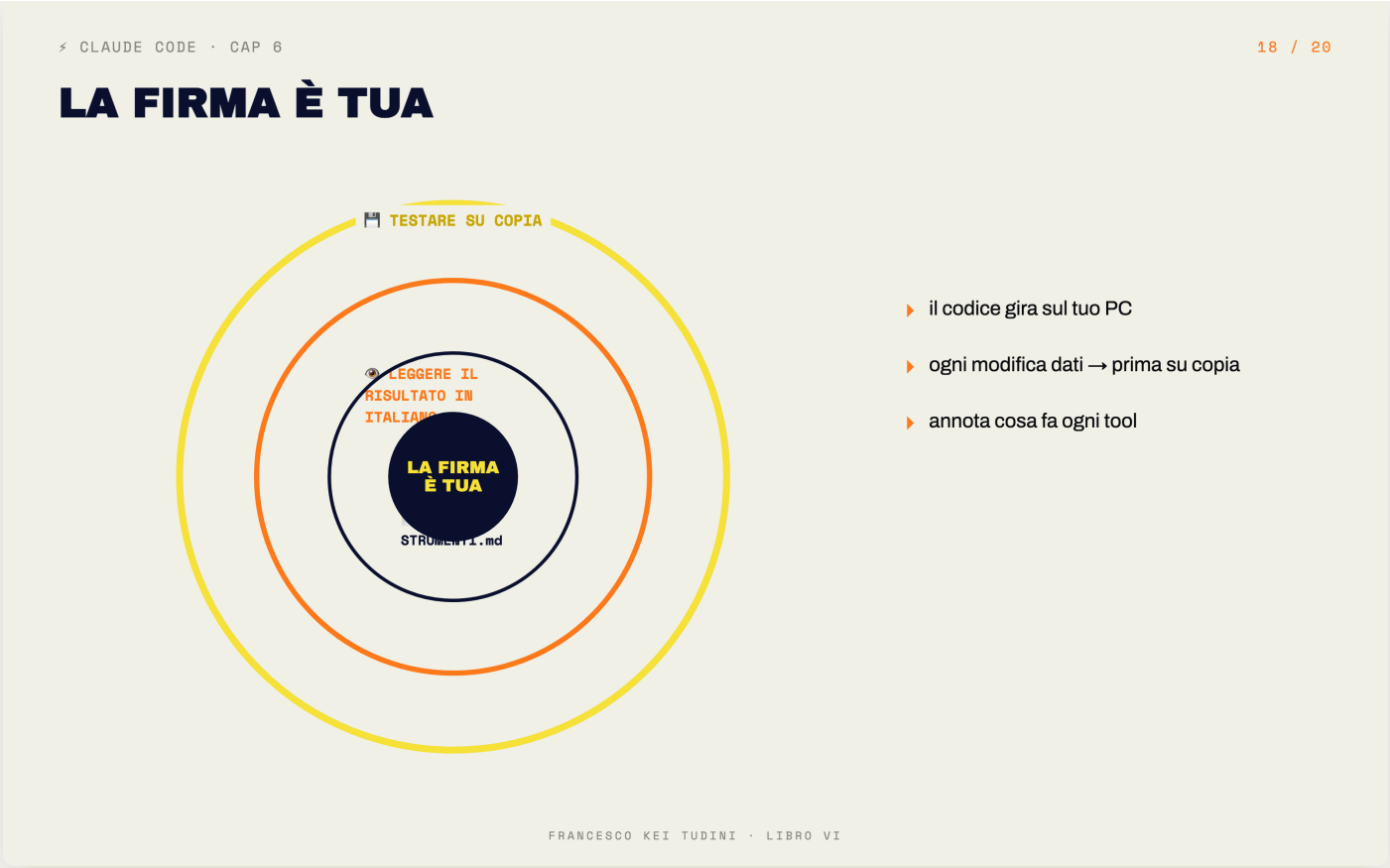
Correzione: modalità Normal per i primi dieci giorni almeno. Poi Auto-Edit *solo* sui task ripetitivi che hai già visto fare bene. YOLO praticamente mai, e comunque solo in sandbox isolata.

Errore 2 — CLAUDE.md vuoto o mai creato

L'errore più diffuso e più costoso: non avere CLAUDE.md. Ogni sessione Claude ricomincia da zero. Tu rispieghi tutto. Il primo quarto d'ora di ogni sessione è speso a dirgli cose che sono nel progetto da sei mesi.

Correzione: primo prompt utile in qualsiasi progetto nuovo → `/init`. Rivedi quello che Claude ha generato, aggiungi le tre o quattro regole specifiche del tuo team, committalo. Cinque minuti, valgono cinque ore al mese.

Errore 3 — Ignorare Plan Mode



«Ho fretta, salto il piano, vado al codice.» Ogni volta che senti questa frase nella tua testa, fermati. Plan Mode dura tre minuti. Plan Mode mancato significa codice fuori standard, refactor inutile, file modificati che non andavano toccati. Costo: ore di pulizia.

Correzione: Plan Mode per **qualunque** task non banalissimo. La regola: se la modifica tocca più di un file o cambia la logica, prima il piano. Sempre.

Errore 4 — Non usare i subagent per non sporcare il contesto principale

Hai bisogno di un'esplorazione lunga? La fai nel thread principale, riempi la finestra di contesto di file letti che non ti servono al thread, dopo dieci minuti devi compactare. Soluzione che già esiste: il subagent **Explore**. Lavora in finestra separata, ti torna solo il riassunto. Contesto principale pulito.

Correzione: ogni volta che stai per dire «leggi tutta questa cartella e dimmi», anteponi «usa il subagent Explore per...». Cambia il rapporto.

Errore 5 — Niente hook su comandi pericolosi

`rm -rf` su una cartella sbagliata. `git push --force origin main`. `npm publish` in produzione senza il flag giusto. Sono comandi che Claude *potrebbe* eseguire se glielo chiedi (e a volte anche se glielo chiedi in modo ambiguo). Senza hook, non c'è una rete di sicurezza che li blocchi prima dell'esecuzione.

Correzione: un PreToolUse hook che intercetta i comandi shell e blocca quelli pericolosi. Cinque righe di script + due righe in `.claude/settings.json`. Da fare il primo giorno utile.

Errore 6 — Permessi troppo larghi

`"allow": ["Bash(*)"]` significa: «Claude può eseguire qualsiasi comando bash senza chiedere». Bella mossa, brutta sicurezza. Un giorno chiede di eseguire una cosa che non avresti voluto, e tu nemmeno te ne accorgi perché non te l'ha chiesto.

Correzione: la lista `allow` deve essere **esplicita**. `Bash(npm test)`, `Bash(npm run lint)`, `Bash(git status)`. Comandi specifici, non wildcard. La regola: in dubbio, non c'è. Meglio dare ogni tanto un OK in più che dare carta bianca per sempre.

Errore 7 — Non leggere i diff

I 7 ERRORI DEL PRIMO MESE

✗ ERRORE	✔ CORREZIONE
✗ Auto-accept su tutto il giorno 1	✔ Normal per i primi 10 giorni
✗ CLAUDE.md vuoto o mai creato	✔ `/init` , poi aggiorna
✗ Ignorare Plan Mode	✔ Plan su tutto ciò che tocca >1 file
✗ Non usare i subagent	✔ usa il subagent Explore
✗ Niente hook su comandi pericolosi	✔ PreToolUse blocca `rm -rf` / push main
✗ Permessi troppo larghi `Bash(*)`	✔ allow esplicito, comando per comando
✗ Non leggere i diff	✔ ogni diff è una decisione

CINQUE MINUTI LA SERA. ZERO DISASTRI LA MATTINA DOPO.

Il diff è il momento in cui vedi cosa Claude *sta per fare*. Verde = aggiunge. Rosso = toglie. Se lo guardi distratto e premi `y`, hai approvato qualcosa di cui non sai niente. Tre giorni dopo, il bug. Una settimana dopo, ti ricordi di aver approvato qualcosa di strano «ma non ti ricordavi cosa».

Correzione: ogni diff è una decisione. Se non hai tempo per leggerlo, non lo approvi — lo metti in coda, mangi un panino, torni, leggi, decidi. Il tempo per leggere il diff è una delle poche regole non negoziabili.

La checklist di fine giornata

Cinque minuti, prima di chiudere il portatile. Cinque domande:

1. **CLAUDE.md è aggiornato?** Ho imparato qualcosa di nuovo sul progetto oggi che il prossimo Claude non saprà ricostruire da solo?
2. **Le modifiche di oggi sono committate?** Niente «lo committo domani».
3. **C'è un test che è fallito e non ho ancora sistemato?** Se sì, scrivilo come TODO nel commit message o in un file.

4. **Ho approvato qualche diff senza leggerlo bene?** Se sì, vai a vedere ora cosa hai approvato.
5. **Domani, prima riga della prossima sessione: cosa farò?** Una frase. Per non perdere venti minuti la mattina a riorientarti.

Cinque minuti. Per anni.

Cosa porti a casa

- **Gli errori non sono dell'AI. Sono del flusso.** Tu controlli il flusso, tu eviti gli errori.
- **Modalità Normal i primi dieci giorni.** Auto-accept lo guadagni nel tempo.
- **CLAUDE.md è obbligatorio, non opzionale.** `/init` il primo giorno. Aggiornamento continuo.
- **Plan Mode su tutto ciò che non è banalissimo.** Tre minuti che valgono ore.
- **Subagent per non sporcare il contesto.** Hook per i comandi pericolosi. Permessi espliciti.
- **Cinque minuti a fine giornata.** Checklist breve, disastri evitati.

Chiusura libro

Hai fatto sei capitoli più questa lista finale di errori. Adesso sai cos'è Claude Code, come si installa, come lo usi nel flusso giusto, come lo configuri con CLAUDE.md/skills/subagents/hooks, come lavorano i pro, come costruisci software senza saper programmare, e cosa non fare il primo mese.

Tutto questo, però, vale solo se lo metti in pratica. Il flusso explore → plan → code → commit, la disciplina dei permessi, l'impalcatura di CLAUDE.md, skills, subagents e hooks: non sono nozioni da ricordare, sono abitudini da costruire. E le abitudini si costruiscono su un progetto vero. Scegli la cosa più piccola che ti serve davvero — uno script che ti pulisce un file, un mini-tool che ti toglie un'ora di lavoro a settimana — e costruiscila questa settimana.

Non sei solo. Ho aperto una community gratuita su Skool dove ti guido passo passo a costruire i tuoi primi strumenti, anche senza programmare, insieme ad altri che partono da dove parti tu. Entra, presenta la tua prima idea, e costruiamola insieme.

→ https://www.skool.com/l-ai-e-la-nuova-elettricita-8966/about

Il momento giusto per iniziare a costruire è adesso. Non «poi». Adesso.

Back-cover

Programmare con Claude Code — guida pratica anche se non sai programmare

Una guida pratica all'AI per imprese e professionisti italiani.

Claude Code è uno strumento diverso da qualsiasi cosa tu abbia mai visto: un agente che vive nel tuo computer, legge i tuoi file, esegue comandi, modifica codice, verifica il suo lavoro. Funziona per chi programma da vent'anni e per chi non sa cos'è un terminale.

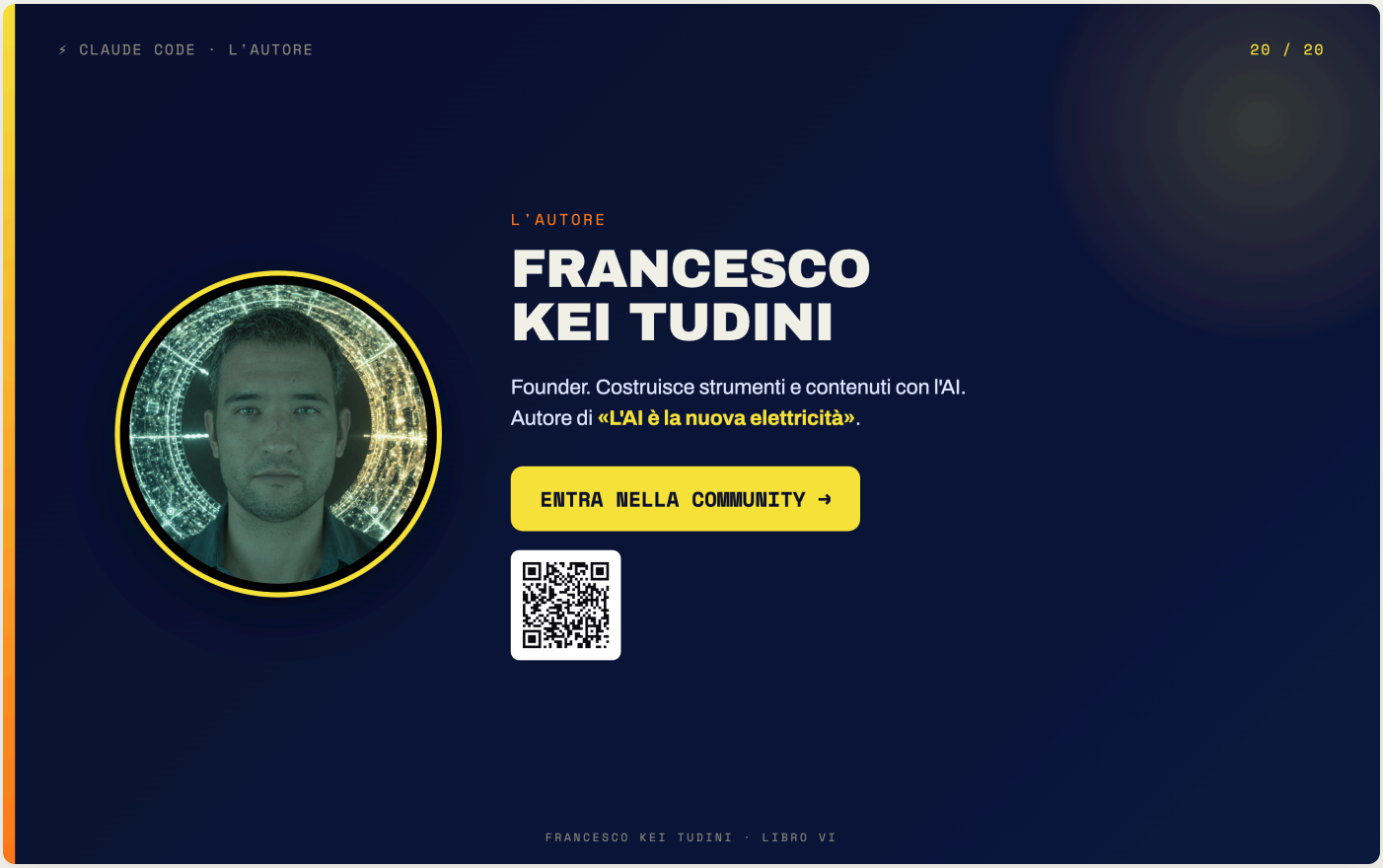
Questo libro ti porta dal primo comando di installazione fino a costruire i tuoi strumenti interni — anche se la programmazione non è il tuo mestiere.

In 55 pagine impari:

- come installare Claude Code in dieci minuti, su qualunque sistema operativo;
- il flusso che vale tutto il libro — explore → plan → code → commit, con esempi reali;
- CLAUDE.md, skills, subagents, hooks — gli strumenti che separano l'uso amatoriale da quello professionale;
- come costruire automazioni e mini-app interne anche se non hai mai scritto codice;
- i sette errori del primo mese — e come evitarli prima di farli.

Per **freelance, professionisti, imprenditori, piccole software house** che vogliono usare l'AI come uno sviluppatore in più al tavolo. Senza fronzoli, con numeri, con il taglio di chi ha visto cosa funziona davvero in azienda.

Francesco Kei Tudini è consulente e imprenditore AI. Aiuta imprese e professionisti italiani a integrare l'AI nei propri flussi di lavoro.



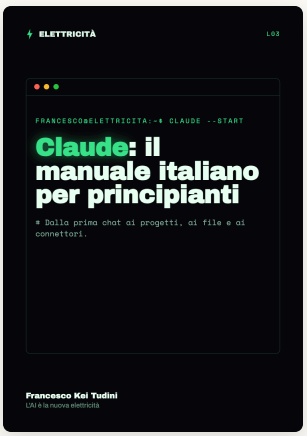
Entra nella community gratuita →

Tutti i libri di Francesco Kei Tudini

Inquadra il QR per aprire ciascun libro.

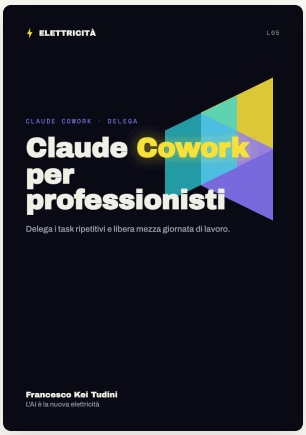
Capire l'intelligenza artificiale

Usare l'AI senza farti fregare



Claude: il manuale italiano

Claude per professionisti



Claude Cowork per professionisti

Claude Code: la guida pratica

L'intelligenza artificiale nella tua impresa

Studiare con l'AI

Insegnare nell'era dell'AI

Costruisci il tuo business con l'AI