Guida Pratica a Claude Code CLI

Installazione, workflow e best practice per iniziare

> *Versione 3.1 — aprile 2026*
>
> *Tutti i contenuti sono stati verificati sulla documentazione ufficiale Anthropic.*

Presentazione

Questa guida è un'introduzione pratica a **Claude Code**, la CLI agentica di Anthropic che porta il modello Claude direttamente nel terminale come collaboratore operativo, capace di leggere codice, eseguire comandi, modificare file e gestire workflow completi.

Il documento è pensato per **sviluppatori che vogliono iniziare a usare Claude Code in modo professionale**, senza affidarsi al passaparola o a tutorial frammentati. Trovi qui il percorso completo dall'installazione ai workflow avanzati, con esempi concreti tratti da scenari WordPress/PHP e progetti generici Node/TypeScript.

A chi si rivolge:

- Sviluppatori web (PHP, JavaScript, Python) con familiarità con il terminale e Git
- Professionisti che vogliono integrare l'AI nel proprio workflow quotidiano in modo consapevole
- Team tecnici che stanno valutando l'adozione di strumenti AI agentici nei processi di sviluppo
- Partecipanti ai workshop Mavida su vibe coding e AI-assisted development

Cosa troverai:

I primi sei capitoli coprono le basi: cosa fa Claude Code, come installarlo, come strutturare il primo progetto, i comandi essenziali e il Plan Mode. I capitoli 7-13 approfondiscono i meccanismi che fanno la differenza tra un uso casuale e uno professionale: memoria persistente con `CLAUDE.md`, gestione del contesto, sicurezza, Skill, plugin, MCP e subagent. I capitoli finali presentano workflow pratici, tips avanzati e una riflessione onesta su quando la CLI supera la chat e quando invece è meglio restare in browser.

Cosa non troverai:

Questa non è una reference esaustiva: per quello c'è la documentazione ufficiale (linkata nell'Allegato B). L'obiettivo è mettere il lettore in condizione di lavorare produttivamente in una o due giornate, sapendo dove approfondire quando serve. Non troverai nemmeno hype sulle capacità dell'AI: il tono è tecnico, onesto sui limiti e attento ai rischi reali (sicurezza, prompt injection, costi nascosti dei token).

Come leggerla:

Se sei alle prime armi, leggi in sequenza almeno fino al capitolo 7 (CLAUDE.md). Se invece hai già installato Claude Code e cerchi best practice specifiche, usa l'indice come riferimento tematico. In fondo trovi un glossario dei termini ricorrenti (Allegato A) e le fonti ufficiali per verifiche e approfondimenti (Allegato B).

Indice

1. Cos'è Claude Code

Claude Code è la CLI (Command Line Interface) sviluppata da Anthropic che porta il modello Claude direttamente nel terminale. Non si tratta di una semplice chat testuale: Claude Code è un **agente autonomo** capace di leggere il codice del progetto, eseguire comandi shell, modificare file, gestire Git e interagire con servizi esterni tramite il protocollo MCP (Model Context Protocol).

A differenza degli assistenti integrati negli IDE (come Copilot), Claude Code opera a livello di **progetto**, non di singolo file. Questo significa che può rispondere a richieste come:

- *"Analizza l'architettura di questo progetto e spiegami come è organizzata l'autenticazione"*
- *"Rifattorizza il modulo dei pagamenti mantenendo tutti i test verdi"*
- *"Trova la root cause di questo bug e correggilo"*

Il modello di lavoro è un **ciclo iterativo**: Claude riceve un obiettivo, esplora il codice con strumenti di lettura, pianifica un'azione, esegue modifiche o comandi, verifica i risultati e procede.

Quando conviene usarlo (e quando no)

Conviene per:

- Onboarding rapido su progetti esistenti
- Refactoring guidato con test di copertura
- Automazione di task ripetitivi (generazione di boilerplate, migrazioni)
- Bug hunting e debugging su basi di codice estese
- Integrazione in pipeline CI/CD (modalità headless `-p`)

Non conviene per:

- Task triviali risolvibili in pochi secondi a mano
- Contesti dove la riservatezza del codice è critica senza opportune policy aziendali
- Chi non ha tempo di imparare a scrivere prompt chiari e verificabili

2. Prerequisiti e piani di abbonamento

Piani compatibili

Claude Code **non è incluso nel piano gratuito**. Serve uno dei seguenti:

Piano	Costo (indicativo)	Indicato per
Claude Pro	$20/mese	Uso individuale moderato, sviluppatori freelance
Claude Max 5x	$100/mese	Uso intensivo, accesso esteso a Opus
Claude Max 20x	$200/mese	Workflow near-autonomous, sessioni multi-agente
Teams / Enterprise	Custom	Organizzazioni con esigenze di compliance
API (Anthropic Console)	Pay-per-token	CI/CD, automazioni, uso sporadico

> *Nota: il prezzo pay-per-token dell'API dipende dal modello. Sonnet 4.6 ha un pricing di $3 per milione di token in input e $15 per milione in output (dati indicativi, verificare sempre sul sito Anthropic).*

Requisiti di sistema

- **macOS**: 13.0 (Ventura) o superiore
- **Linux**: Ubuntu 20.04+, Debian 10+, o distribuzioni equivalenti
- **Windows**: Windows 10 (1809+) o Windows 11, nativo con Git for Windows oppure tramite WSL2 (consigliato)
- **RAM**: minimo 4 GB, consigliati 8 GB per codebase estese
- **Shell**: Bash, Zsh, PowerShell o CMD
- **Connessione internet**: sempre necessaria (il modello gira sui server Anthropic)

3. Installazione

Nel 2025 Anthropic ha introdotto il **native installer** come metodo raccomandato, sostituendo l'installazione via npm (che rimane supportata ma deprecata). Il native installer ha tre vantaggi:

1. Nessuna dipendenza da Node.js
2. Auto-update automatico in background
3. Nessun problema di permessi tipico di `npm install -g`

3.1 Installazione su macOS e Linux

Apri il terminale ed esegui:

```
# Scarica ed esegue lo script di installazione ufficiale
curl -fsSL https://claude.ai/install.sh | bash
```

Cosa fa questo comando, passo passo:

1. `curl` scarica lo script dalla URL di Anthropic
2. `-fsSL` sono quattro flag combinate:
   - `-f` fa fallire curl in caso di errore HTTP (evita di eseguire pagine di errore)
   - `-s` modalità silenziosa (niente progress bar)
   - `-S` mostra comunque gli errori se qualcosa va storto
   - `-L` segue i redirect HTTP
3. La pipe `|` passa lo script scaricato direttamente a `bash` per l'esecuzione
4. Lo script scarica il binario corretto per la tua piattaforma, lo posiziona in `~/.local/bin` e configura l'auto-update

> *Nota di sicurezza: eseguire script scaricati da internet tramite pipe è una pratica che va valutata. Se lavori in contesti enterprise, scarica prima lo script, ispezionalo, e poi eseguilo separatamente.*

3.2 Installazione su Windows

Apri **PowerShell** (non CMD) ed esegui:

```
# Scarica ed esegue lo script PowerShell ufficiale
irm https://claude.ai/install.ps1 | iex
```

Come funziona:

- `irm` è l'alias di `Invoke-RestMethod`: scarica il contenuto della URL
- `iex` è l'alias di `Invoke-Expression`: esegue il contenuto scaricato come script PowerShell

> *Se vedi l'errore `'irm' is not recognized`, sei in CMD invece che in PowerShell. Il prompt di PowerShell mostra `PS C:\>`, quello di CMD mostra solo `C:\>`.*

Installazione nativa su Windows richiede Git for Windows. Installalo prima se non ce l'hai.

3.3 Installazione via WSL2 (consigliata per Windows)

Per i progetti Unix-like, WSL2 offre un ambiente più pulito e compatibile:

```
# Installa WSL2 (richiede riavvio)
wsl --install
```

Dopo il riavvio, apri Ubuntu (installato di default) e usa il comando Linux:

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

3.4 Installazione alternativa via npm (deprecata ma supportata)

Se hai motivi specifici per usare npm (esempio: pinning di versione, ambienti dove npm è lo standard):

```
# Richiede Node.js 18 o superiore
npm install -g @anthropic-ai/claude-code
```

> *Non usare* `sudo`. *Se ottieni errori di permessi, la soluzione corretta è usare* `nvm` *(Node Version Manager), che installa Node nella tua home directory evitando il problema alla radice.*

3.5 Verifica dell'installazione

Dopo l'installazione, verifica che tutto funzioni:

```
# Controlla la versione installata
claude --version

# Diagnostica completa: auth, PATH, MCP, permessi file
claude doctor
```

Il comando `claude doctor` è il tuo migliore amico quando qualcosa non va: esegue una serie di controlli e ti dice esattamente cosa sistemare.

3.6 Autenticazione

Al primo avvio, `claude` apre il browser per l'OAuth:

```
cd ~/mio-progetto
claude
```

Login con il tuo account Anthropic (quello del piano Pro/Max). La sessione viene salvata e persiste tra i riavvii del terminale.

Per ambienti headless (CI/CD, server), usa la API key:

```
# Aggiungi a ~/.zshrc, ~/.bashrc o ~/.profile
export ANTHROPIC_API_KEY="sk-ant-api03-..."
```

4. Il primo progetto end-to-end

Vediamo un flusso completo partendo da zero. Supponiamo di avere un plugin WordPress da analizzare.

Step 1: Posizionati nella directory del progetto

```
cd ~/mavida/wp-access-control-block
```

> *Claude Code usa **sempre** la directory corrente come contesto di lavoro. Non lanciare `claude` dalla home se vuoi lavorare su un progetto specifico.*

Step 2: Inizializza il progetto

```
claude
```

Una volta dentro la sessione interattiva, esegui:

```
/init
```

Questo comando analizza la struttura del progetto e genera automaticamente un file `CLAUDE.md` nella root. Il file contiene:

- Panoramica del progetto (stack tecnologico rilevato)
- Architettura principale
- Comandi di build/test rilevati (da `package.json`, `composer.json`, ecc.)
- Convenzioni del codice

Step 3: Rivedi e personalizza `CLAUDE.md`

Il file generato automaticamente è un punto di partenza. Aprilo e arricchiscilo con informazioni specifiche del tuo progetto (vedi sezione 7 per esempi dettagliati).

Step 4: Prima richiesta

Torna nella sessione Claude e scrivi il primo prompt:

```
Analizza la struttura del plugin e spiegami:
1. Come è organizzato il codice (namespaces, pattern)
2. Come viene registrato il Gutenberg block
3. Dove sono gestiti i controlli di accesso
Non modificare nulla, solo esplora e riporta.
```

Claude leggerà i file rilevanti, produrrà un'analisi e si fermerà in attesa di ulteriori istruzioni.

Step 5: Esci dalla sessione

```
/exit
```

Oppure `Ctrl+D`.

Step 6: Riprendi dove hai lasciato

Quando torni al progetto:

```
cd ~/mavida/wp-access-control-block
claude --continue
```

Il flag `--continue` carica la sessione più recente di questa directory. In alternativa, `claude --resume` mostra una lista di sessioni passate tra cui scegliere.

5. Comandi e scorciatoie essenziali

Flag di lancio (dal terminale)

Flag	Descrizione
`claude`	Avvia una nuova sessione interattiva
`claude --continue`	Riprende la sessione più recente nella directory corrente
`claude --resume`	Mostra l'elenco delle sessioni passate e permette di sceglierne una
`claude -p "prompt"`	Modalità headless (print): esegue un prompt e stampa l'output, ideale per script e CI/CD
`claude --model <nome>`	Specifica il modello (es. `claude-opus-4-6`, `claude-sonnet-4-6`, `claude-haiku-4-5`)
`claude --dangerously-skip-permissions`	Salta le conferme sui comandi (usare con cautela, solo in ambienti isolati)
`claude doctor`	Esegue diagnostica completa
`claude --help`	Mostra la guida dei comandi

Slash command (dentro la sessione)

Comando	Descrizione
`/help`	Elenco completo dei comandi disponibili
`/init`	Genera `CLAUDE.md` analizzando il progetto
`/clear`	Azzera il contesto della conversazione
`/compact`	Comprime la cronologia in un sommario per liberare token
`/undo`	Annulla l'ultima modifica ai file fatta da Claude
`/model`	Cambia modello durante la sessione
`/plan`	Attiva Plan Mode (disponibile dalla v2.1.0)
`/config`	Apre la configurazione interattiva
`/terminal-setup`	Configura scorciatoie terminale (es. Shift+Enter per newline)
`/ide`	Collega Claude Code all'IDE aperto (VS Code, Cursor, ecc.)
`/bug`	Segnala un bug direttamente ad Anthropic
`/exit` o `/quit`	Chiude la sessione

Scorciatoie da tastiera

Scorciatoia	Azione
`Shift+Tab` (due volte)	Attiva/disattiva **Plan Mode**
`Shift+Tab` (una volta, dopo Plan Mode)	Seleziona Auto-Accept Mode (v2.1.2+)
`Ctrl+C`	Interrompe l'operazione in corso
`Ctrl+R`	Ricerca reverse nella cronologia comandi
`Ctrl+O`	Apre il transcript viewer
`Esc Esc`	Torna indietro per rieditare un messaggio precedente (crea un fork della conversazione)
`Shift+Enter`	Newline senza inviare (richiede `/terminal-setup` su alcuni terminali)

6. Plan Mode: pensare prima di scrivere

Plan Mode è probabilmente la feature più importante da padroneggiare per un uso sicuro di Claude Code. È una modalità **read-only** in cui Claude analizza il progetto e propone un piano, ma **non tocca alcun file** finché tu non approvi esplicitamente.

Perché è importante

Senza Plan Mode, Claude tende a essere estremamente veloce nell'eseguire. Chiedi una "piccola correzione" e ti ritrovi 12 file modificati in 15 secondi. Plan Mode inverte questo flusso: prima pensi, poi esegui.

Come attivarlo

Durante una sessione, premi `Shift+Tab` **due volte**. Vedrai un indicatore che conferma l'attivazione.

> *Nota Windows: dalla v2.1.3 di Claude Code su Windows c'è un bug noto sul binding* `Shift+Tab`*. In alternativa usa il comando* `/plan`*.*

Strumenti disponibili in Plan Mode

Claude può usare solo strumenti di lettura e ricerca:

- `Read`, `Glob`, `Grep`: lettura e ricerca nel codice
- `WebFetch`, `WebSearch`: ricerca online
- `Task`: delegare ricerche a subagent
- `TodoRead/TodoWrite`: gestione task

Gli strumenti di **modifica sono bloccati**:

- `Edit`, `MultiEdit`, `Write`: editing file
- `Bash`: esecuzione comandi
- Tutti i tool MCP che modificano stato

Esempio di workflow con Plan Mode

```
[Shift+Tab, Shift+Tab — Plan Mode attivato]

Prompt: "Devo migrare il sistema di logging da error_log() a Monolog.
Analizza tutte le occorrenze e proponi un piano di migrazione incrementale."

Claude risponde con:
- Elenco dei 23 file coinvolti
- Strategia di migrazione in 4 fasi
- Rischi e punti di attenzione
- Stima di complessità per ogni fase

[Rivedi il piano]
[Se ok: Shift+Tab per uscire e approvare]
[Claude esegue il piano]
```

7. CLAUDE.md: la memoria persistente del progetto

Il file `CLAUDE.md` nella root del progetto è il **contratto** tra te e Claude. Viene letto automaticamente a ogni sessione e fornisce il contesto persistente che altrimenti dovresti ripetere ogni volta.

Cosa mettere in CLAUDE.md

Una buona struttura include:

1. **Descrizione del progetto** — cos'è, a chi serve
2. **Stack tecnologico** — linguaggi, framework, versioni
3. **Convenzioni di codice** — naming, pattern, stile commenti
4. **Comandi principali** — build, test, lint, deploy
5. **Architettura ad alto livello** — cartelle chiave, flusso dati
6. **Cosa NON fare** — anti-pattern, regole invalicabili

Esempio 1: Plugin WordPress

```
# WP Access Control Block

## Descrizione
Plugin WordPress che aggiunge un Gutenberg block per controllare
la visibilità dei contenuti in base allo stato di login dell'utente.

## Stack
- PHP 8.1+
- WordPress 6.0+
- JavaScript con @wordpress/scripts (JSX)
- SCSS con metodologia BEM

## Convenzioni
- Commenti in codice: **italiano**
- README e documentazione utente: **inglese**
- Naming PHP: PSR-12, namespace `Mavida\WPAccessControl\`
- Naming JS: camelCase, componenti React in PascalCase
- CSS: BEM strict (`.block__element--modifier`)

## Comandi
- Build: `npm run build`
- Dev watch: `npm run start`
- Lint PHP: `composer lint`
- Lint JS: `npm run lint:js`
- Test PHP: `composer test`

## Struttura
- `src/` — sorgenti JS/SCSS (JSX, SCSS)
- `build/` — output compilato (non toccare manualmente)
- `includes/` — logica PHP lato server
- `block.json` — manifest del blocco

## Regole invalicabili
- NON usare jQuery nei nuovi componenti
- NON committare file in `build/`
- Ogni hook PHP deve avere nonce verification
- I render callback server-side devono essere **escaped** con le funzioni WP
```

Esempio 2: Progetto Node/TypeScript generico

```
# API Analytics Service

## Descrizione
Microservizio REST per raccogliere e aggregare eventi di analytics.

## Stack
- Node.js 22 LTS
- TypeScript 5.4 (strict mode)
- Fastify 4
- PostgreSQL 16 + Prisma ORM
- Vitest per i test

## Convenzioni
- Tutti i tipi esportati devono essere in `src/types/`
- Schema Zod per validazione input (mai fidarsi del client)
- Commenti JSDoc per tutte le funzioni pubbliche
- Niente `any`, usa `unknown` e narrow

## Comandi
- Dev: `pnpm dev`
- Build: `pnpm build`
- Test: `pnpm test`
- Test coverage: `pnpm test:coverage`
- Migrazioni DB: `pnpm prisma migrate dev`

## Architettura
- `src/routes/` — endpoint HTTP
- `src/services/` — logica di business
- `src/repositories/` — accesso dati
- `src/schemas/` — validazione Zod

## Regole invalicabili
- NON importare direttamente Prisma client nei `routes/` — passare per `repositories/`
- Ogni endpoint deve avere uno schema Zod per body/params/query
- Test obbligatori per ogni nuovo service
```

CLAUDE.md gerarchici

Claude Code legge **più file CLAUDE.md in ordine gerarchico:**

- `~/.claude/CLAUDE.md` — regole globali utente
- `<monorepo-root>/CLAUDE.md` — regole del monorepo
- `<project>/CLAUDE.md` — regole del singolo progetto

Questo ti permette di definire preferenze personali una volta sola e sovrascriverle a livello di progetto quando serve.

8. Gestione del contesto

La finestra di contesto è la risorsa più preziosa di Claude Code. I modelli moderni supportano tra i **200.000 token** (default) e **1 milione di token** (in beta per Opus/Sonnet più recenti). Nonostante l'ampiezza, è facile saturarla su progetti grandi.

Quando il contesto si riempie

Segnali di contesto saturo:

- Claude inizia a "dimenticare" cose discusse prima
- Le risposte diventano più lente
- Appaiono warning nella status line (se configurata)

Strumenti per gestire il contesto

`/compact` — comprime la conversazione in un sommario mantenendo solo le informazioni essenziali. Utile a metà sessione quando hai già completato una fase e stai iniziandone un'altra.

```
[Dopo aver completato il refactoring del modulo auth]
/compact
[Claude produce un sommario di ~500 token al posto di ~50.000]
[Ora puoi iniziare una nuova fase con contesto libero]
```

`/clear` — azzera completamente il contesto. Usa quando cambi completamente task e non ti serve nulla di quello che è stato detto prima.

Subagent — per ricerche lunghe che "sporcherebbero" il contesto (es. leggere 50 file per trovare un pattern), delega a un subagent. Vedi sezione 12.

Regola pratica

> *Se ti accorgi di aver fatto tre cose diverse nella stessa sessione, probabilmente avresti dovuto usare `/clear` due volte.*

Sessioni focalizzate producono output migliori e consumano meno token.

9. Sicurezza e gestione dei permessi

Claude Code è un agente **autonomo** che esegue comandi nel tuo sistema. Senza le giuste precauzioni, è un vettore di rischio reale.

Il sistema dei permessi

Di default, Claude chiede conferma prima di eseguire qualsiasi operazione di modifica (scrittura file, comandi shell, chiamate MCP che modificano stato). Le operazioni di lettura sono auto-approvate:

- `Read`, `Glob`, `Grep`, `WebSearch`, `LSP` → nessuna conferma
- `Edit`, `Write`, `Bash`, MCP di scrittura → conferma richiesta

Configurare i permessi in `settings.json`

Puoi definire regole granulari nel file `.claude/settings.json` del progetto:

```
{
  "permissions": {
    "allow": [
      "Bash(npm run test:*)",
      "Bash(npm run lint:*)",
      "Bash(git status)",
      "Bash(git diff)",
      "Read(**)"
    ],
    "deny": [
      "Read(.env*)",
      "Read(**/secrets/**)",
      "Read(**/.aws/credentials)",
      "Bash(rm -rf *)",
      "Bash(sudo *)",
      "Bash(curl * | bash)",
      "Bash(wget * | sh)"
    ]
  }
}
```

Spiegazione passo passo:

1. `allow` — operazioni che Claude può eseguire **senza chiedere conferma**
2. `deny` — operazioni **bloccate fisicamente**, anche se Claude ci prova non succede nulla
3. I pattern usano glob (`**` per qualsiasi percorso, `*` per segmento singolo)
4. `deny` ha **precedenza** su `allow`

Proteggere i segreti

Nonostante `.claudeignore`, ci sono scenari in cui Claude potrebbe leggere file sensibili (prompt injection, errori di configurazione). Usa **sempre** `permissions.deny` per file `.env`, credenziali, chiavi private.

Modalità pericolose

`--dangerously-skip-permissions` salta tutte le conferme. È utile per: - Esecuzione autonoma in ambienti sandbox/Docker - Task lunghi dove non vuoi essere interrotto ogni 30 secondi

Il nome è esplicito: **non è un flag da usare alla leggera**. Linee guida:

- **Mai** su macchine che contengono credenziali produzione
- **Mai** con accesso a repository aziendali sensibili
- **Solo** in container isolati o VM dedicate allo scopo

Prompt injection

Un attaccante potrebbe inserire istruzioni malevole in:

- Commenti di codice che Claude legge
- File README scaricati da dipendenze
- Risposte di servizi MCP non fidati
- Nomi di file manipolati

Difese pratiche:

1. Usa sempre Plan Mode per task su codice di terze parti
2. Rivedi sempre il piano prima di approvarlo
3. Non eseguire Claude Code con privilegi di amministratore
4. Isola i progetti esterni in directory separate con `settings.json` restrittivi

10. Skill: il meccanismo di estensione

Le Skill sono "playbook" specializzati che Claude può consultare automaticamente quando rileva che un certo tipo di task è in gioco. **Importante**: a differenza degli slash command, **le Skill non si invocano con un comando**. Si attivano automaticamente in base alla loro `description`.

Come funziona una Skill

Una Skill è una cartella con un file `SKILL.md` strutturato così:

```
---
name: wordpress-block-builder
description: "Use this skill when building or modifying WordPress
Gutenberg blocks. Triggers on: block.json, @wordpress/scripts,
JSX files in WordPress plugins, Edit/Save components."
---

# WordPress Block Builder

## Convenzioni del progetto
- Usa sempre @wordpress/scripts per il build
- Ogni blocco deve avere block.json, edit.js, save.js, style.scss
- Attributi devono essere tipizzati in block.json

## Pattern consigliati
[...]
```

Il campo `description` determina **quando** Claude userà la Skill. È il pezzo più importante: scrivilo pensando ai trigger concreti del tuo workflow.

Skill native incluse

Claude Code viene distribuito con alcune Skill ufficiali, tra cui:

- **pdf** — creazione, lettura, fill forms, merge/split PDF
- **docx** — creazione e modifica documenti Word
- **pptx** — creazione presentazioni PowerPoint
- **xlsx** — creazione e manipolazione fogli Excel
- **frontend-design** — linee guida per interfacce React/HTML di qualità
- **skill-creator** — skill per creare nuove skill (meta!)
- **mcp-builder** — skill per costruire server MCP custom

Creare una Skill personalizzata

Per il tuo workflow Mavida, potresti creare:

```
.claude/skills/
├── mavida-wordpress/
│   └── SKILL.md          (convenzioni WP di Mavida)
├── n8n-workflow/
│   └── SKILL.md          (pattern per workflow N8N)
└── php-legacy-review/
    └── SKILL.md          (checklist per refactoring PHP legacy)
```

Ogni volta che lavori su un progetto WordPress, Claude riconosce il contesto e applica automaticamente le convenzioni definite in `mavida-wordpress/SKILL.md`.

---

11. Caveman: la skill virale per risparmiare token

Tra le Skill della community che hanno fatto più rumore nel 2026 c'è **Caveman**, creata da Julius Brussee. A fine aprile 2026 il repository ha superato le 24.000 star su GitHub ed è stata discussa su Hacker News e Product Hunt. Vale la pena dedicarle una sezione, sia per capire cosa fa davvero sia per separare il segnale dal rumore marketing.

Cos'è e da dove nasce

Caveman è una **skill di Claude Code** (ora distribuita anche come plugin) che forza il modello a rispondere in stile telegrafico: niente articoli, niente convenevoli, niente hedging, niente meta-commentari. Solo sostanza tecnica. L'idea nasce da un'osservazione virale su Reddit: istruendo un LLM a "parlare come un uomo delle caverne" si riduce drasticamente l'output senza perdere accuratezza tecnica.

Il principio è semplice: Claude di default spende molti token in frasi di cortesia ("Sure, I'd be happy to help"), spiegazioni ridondanti ("The reason this is happening is because") e qualificazioni prudenti ("It might be worth considering"). Caveman le elimina tutte.

Cosa viene compresso e cosa no

Caveman è chirurgico: comprime solo la parte discorsiva della risposta, mantiene intatta la sostanza tecnica:

Contenuto	Caveman lo tocca?
Spiegazioni in linguaggio naturale	✂️ Sì, rimuove filler e articoli
Blocchi di codice	❌ No, restano normali
Termini tecnici (polymorphism, useMemo, etc.)	❌ No, restano esatti
Messaggi di errore citati	❌ No, quote letterali
Commit message e PR	❌ No, inglese tecnico standard
Articoli (a, an, the)	✂️ Eliminati
Convenevoli ("Sure, I'd be happy to")	✂️ Eliminati
Hedging ("It might be worth considering")	✂️ Eliminati

Esempio before/after

Risposta normale di Claude (69 token):

> *"The reason your React component is re-rendering is likely because you're creating a new object reference on each render cycle. When you pass an inline object as a prop, React's shallow comparison sees it as a different object every time, which triggers a re-render. I'd recommend using useMemo to memoize the object."*

Risposta in modalità Caveman (19 token):

> *"New object ref each render. Inline object prop = new ref = re-render. Wrap in* `useMemo`*."*

Stessa diagnosi. Stessa soluzione. Stesso codice. Solo niente imbottitura.

I tre livelli di compressione

Caveman offre tre intensità che puoi scegliere in base al contesto:

- `lite` — rimuove solo convenevoli e filler, mantiene la grammatica completa. Il più leggibile.
- `full` (default) — frasi frammentate, articoli eliminati, stile telegrafico.
- `ultra` — massima compressione, stile da appunti. Solo parole ad alto valore informativo.

Esistono anche varianti sperimentali in cinese classico (wenyan) per sessioni multilingua, ma hanno utilità limitata per chi lavora in italiano o inglese.

Installazione e attivazione

Caveman si installa come plugin dal marketplace dell'autore:

```
# Aggiungi il marketplace di Julius Brussee
claude plugin marketplace add JuliusBrussee/caveman

# Installa la skill
claude plugin install caveman@caveman
```

Una volta installata, attivala dentro una sessione Claude Code con uno di questi trigger:

```
/caveman            # attiva livello full (default)
/caveman lite       # livello leggero
/caveman ultra      # massima compressione
```

Oppure in linguaggio naturale: *"talk like caveman"*, *"caveman mode"*, *"less tokens please"*.

Per disattivare:

```
stop caveman
```

oppure semplicemente: *"normal mode"*.

La verità sui numeri (parte importante)

Il README del progetto parla di **~75% di riduzione dei token**. Questo numero è diventato virale, ma va letto con attenzione perché può generare aspettative sbagliate:

1. **Il 75% si riferisce solo ai token di output in risposte discorsive.** I benchmark indipendenti (misurati con `tiktoken` nel repo ufficiale) mostrano:
   - 68% di riduzione su task di web search
   - 50% di riduzione su code edit

    - 72% di riduzione su Q&A
    - **Media reale: ~61%** con range 22-87% a seconda del task.

2. **Gli output token sono solo una parte del costo totale.** In una sessione Claude Code tipica, il grosso dei token consumati sta nell'**input**: `CLAUDE.md` riletto a ogni turno, cronologia della conversazione, file che Claude apre per rispondere. Alcune analisi indipendenti stimano che l'output sia solo lo 0.6-2.5% del totale.
3. **Il risparmio reale sulla bolletta** si aggira tra il **15% e il 25%** su sessioni di coding intensive. Un utente Reddit ha riportato $1 risparmiato su un abbonamento da $100. Non è il 75% promesso, ma non è neanche trascurabile: su un uso quotidiano diventano cifre a tre zeri l'anno.
4. **Bonus inatteso**: un paper di marzo 2026 suggerisce che i vincoli di brevità possono **migliorare l'accuratezza** del modello su certi benchmark (+26 punti percentuali). Meno imbottitura sembra forzare Claude a rispondere in modo più preciso.

Quando usarla e quando no

Usa Caveman quando:

- Stai facendo coding meccanico ripetitivo (refactoring, debug, linting)
- Sei un utente esperto che non ha bisogno del "perché" dettagliato
- Stai orchestrando agent multipli o background task dove l'output verbose è solo rumore
- Sei vicino al limite del tuo piano e vuoi spremere più sessioni

Non usare Caveman quando:

- Stai imparando un nuovo framework: ti serve la pedagogia, il "perché" è il valore
- Stai facendo onboarding su un codebase sconosciuto: vuoi spiegazioni complete
- Stai facendo revisione architetturale: vuoi sfumature, alternative, trade-off
- Il costo dei token non è un problema rilevante per il tuo volume d'uso

L'ecosistema Caveman

Brussee ha esteso il progetto in una famiglia di strumenti correlati:

- `caveman` — compressione dell'output (quello descritto sopra)
- `caveman-commit` — genera commit message Conventional Commits ultra-concisi, subject ≤50 caratteri
- `caveman-review` — commenti di code review in una riga (`L<line>: <problema>. <fix>.`)
- `caveman-compress` — riscrive `CLAUDE.md` in formato caveman, riducendo ~46% di input token a ogni sessione
- `cavemem` — compressione della memoria persistente
- `cavekit` — orchestrazione build

Il più interessante per chi cerca risparmi reali è probabilmente `caveman-compress`: agisce sull'input, dove il costo effettivo si nasconde. Se orchestri flotte di agent o lavori con `CLAUDE.md` corposi, il risparmio si moltiplica per ogni turno di ogni sessione.

Alternativa zero-install

Se non vuoi installare plugin ma vuoi comunque ridurre la verbosità, aggiungi queste righe al tuo `CLAUDE.md`:

```
## Communication Style

Be concise. No filler. No hedging. State conclusions first, reasoning second.
Respond in short, direct sentences. Code blocks stay normal.
```

Questo semplice prompt ti dà circa il 40-50% della compressione di Caveman senza installare nulla. Non è la stessa cosa — Caveman è più raffinata e ha modalità selezionabili — ma per molti casi d'uso è sufficiente.

Il mio giudizio

Caveman è un esempio interessante di come la community stia iterando sopra Claude Code. Il numero "75%" è marketing, ma il prodotto sottostante è onesto: fa quello che promette, è open source MIT, e ti dà uno strumento concreto per gestire il costo. La parte più preziosa non è la compressione in sé, ma il **principio**: pensare alla verbosità dell'AI come a un costo misurabile, non come a un'inevitabilità. Una volta che inizi a ragionare in questi termini, scrivi `CLAUDE.md` più leggeri, prompt più mirati, e la tua efficienza cresce anche senza plugin.

---

12. Plugin e MCP: integrare servizi esterni

Plugin marketplace

Claude Code supporta un sistema di plugin che può essere esteso con marketplace della community:

```
/plugin marketplace add anthropics/skills
/plugin install <nome-plugin>
```

I plugin possono aggiungere slash command, agent e skill ufficialmente distribuite da Anthropic o dalla community.

MCP (Model Context Protocol)

MCP è il protocollo standard per connettere Claude Code a **servizi esterni**: Google Drive, GitHub, database, Slack, Jira, e molti altri.

Un server MCP espone **tool** che Claude può chiamare come se fossero funzioni native. Esempio di server MCP utili:

- **GitHub** — creare PR, gestire issue, leggere repository
- **PostgreSQL/MySQL** — query, schema inspection, migrazioni guidate
- **Filesystem** — accesso controllato a directory specifiche
- **Puppeteer** — automazione browser headless

Configurare un server MCP

Esempio di configurazione in `.claude/settings.json`:

```
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_PERSONAL_ACCESS_TOKEN": "${GITHUB_TOKEN}"
      }
    }
  }
}
```

Una volta configurato, puoi chiedere a Claude: *"Crea una PR con le modifiche degli ultimi commit e aggiungi come reviewer @marco"*. Claude userà il tool MCP GitHub per farlo.

> *Attenzione ai permessi: un server MCP esegue nel tuo sistema con le tue credenziali. Installa solo server di cui ti fidi o che hai scritto tu. Verifica sempre il codice sorgente prima.*

13. Subagent: delegare task isolati

I **subagent** sono istanze di Claude con il proprio contesto isolato, specializzate per task specifici. A differenza della conversazione principale, ciò che fa un subagent non "sporca" il contesto della sessione corrente.

Quando usarli

- **Ricerche lunghe**: leggere 30 file per trovare un pattern
- **Revisioni parallele**: far analizzare lo stesso codice da punti di vista diversi (sicurezza, performance, stile)
- **Task specialistici ricorrenti**: code review, scrittura test, refactoring

Subagent integrati

Claude Code espone il tool `Task` che crea automaticamente subagent per:

- Ricerca approfondita nel codebase
- Analisi di file multipli
- Esecuzione di piani multi-step isolati

Esempio pratico

```
Prompt principale: "Trova tutte le funzioni PHP che non hanno nonce
verification nel nostro plugin. Delega a un subagent per non
sovraccaricare il contesto."

[Claude crea un subagent con tool Read/Grep]
[Il subagent analizza ~50 file]
[Restituisce alla sessione principale solo il sommario: "Trovate 7
funzioni vulnerabili nei file X, Y, Z"]
```

Il contesto principale riceve solo il risultato, non tutti i file letti.

14. Workflow pratici

13.1 Onboarding su un repository esistente

```
Prompt: "Sei appena stato assegnato a questo progetto. Analizza la
struttura, identifica:
1. Pattern architetturali principali (MVC, hexagonal, ecc.)
2. Come è gestita l'autenticazione
3. Dove sono i punti di integrazione con servizi esterni
4. Convenzioni di naming e stile
5. Eventuali debiti tecnici evidenti

Produci un documento di onboarding in docs/ONBOARDING.md.
Non modificare altro codice."
```

Perché funziona: - Obiettivo chiaro con lista numerata - Output specifico (un file in posizione nota) - Vincolo esplicito (“non modificare altro”)

13.2 Bug hunting con TDD

```
Prompt: "Bug report: quando un utente con ruolo 'editor' prova a
modificare un post 'private', riceve errore 500. Log allegato:
[incolla log].

Workflow richiesto:
1. Attiva Plan Mode e analizza il codice coinvolto
2. Scrivi PRIMA un test che riproduca il bug (deve fallire)
3. Correggi il bug con la modifica MINIMA necessaria
4. Verifica che il test passi
5. Esegui la suite completa per escludere regressioni"
```

Perché funziona: - Forza un approccio TDD disciplinato - Evita le “correzioni veloci” che sopprimono sintomi - Il test scritto prima diventa documentazione del bug

13.3 Refactoring sicuro

```
Prompt: "Il modulo includes/class-order-processor.php è diventato
ingestibile (800 righe, responsabilità multiple). Voglio
rifattorizzarlo.

Fase 1 — CARATTERIZZAZIONE (Plan Mode):
- Identifica tutte le responsabilità attualmente mescolate
- Proponi una scomposizione in classi più piccole
- Elenca i test che DEVONO esistere prima di toccare il codice

Fermati qui e aspetta la mia approvazione del piano."
```

Dopo approvazione:

```
"Procedi con la Fase 2:
- Scrivi i test di caratterizzazione che bloccano il
  comportamento attuale
- Eseguili e conferma che passano tutti
- Fai un commit con messaggio 'test: caratterizzazione pre-refactoring'"
```

E poi:

```
"Fase 3 — refactoring incrementale:
- Estrai una responsabilità alla volta
- Dopo ogni estrazione, esegui i test
- Se fallisce anche UN solo test, fermati e chiedi"
```

13.4 Audit di performance

```
Prompt: "Analizza il build di produzione e identifica i 5 problemi
di performance a più alto impatto. Per ciascuno:
- File e linee coinvolti
- Impatto stimato (ms, KB, richieste HTTP)
- Fix proposto
- Complessità del fix (bassa/media/alta)

Ordina per rapporto impatto/complessità. Non modificare nulla."
```

15. Tips per utenti avanzati

Vim mode

Se vieni da Vim, abilita la modalità in `/config` → Editor mode. Avrai navigazione con `hjkl`, comandi `d`, `y`, `p`, ecc.

Custom slash command

Puoi creare slash command personalizzati salvando file Markdown in `.claude/commands/`:

```
←!— .claude/commands/security-audit.md —→
Esegui un audit di sicurezza focalizzato su:
1. SQL injection nelle query dirette
2. XSS negli output non escaped
3. CSRF senza nonce verification
4. Path traversal nelle operazioni filesystem
5. Credenziali hardcoded

Per ogni issue trovata: file, riga, severity (low/medium/high/critical),
fix suggerito.
```

Ora da sessione puoi lanciare `/security-audit` e Claude esegue il prompt salvato.

Modalità headless per CI/CD

Il flag `-p` (print) esegue Claude in modalità non-interattiva, perfetta per pipeline:

```
# Esempio GitHub Actions
claude -p "Review the changes in this PR and flag any security issues" \
       --output-format json > review.json
```

Il `--output-format json` produce output strutturato parsabile da step successivi.

Recap delle sessioni

Se lasci il terminale e torni dopo 3+ minuti, Claude Code mostra automaticamente un riepilogo di quello che è stato fatto. Ottimo per context-switching. Puoi forzarlo con `/recap`.

Checkpoint Git strategici

Prima di task rischiosi, chiedi esplicitamente:

> *"Prima di procedere, fai un commit con messaggio 'checkpoint pre-refactoring' così abbiamo un punto di ritorno sicuro."*

Se qualcosa va storto, `git reset --hard HEAD~1` ti riporta al punto precedente.

Fork della conversazione

Premi `Esc` due volte per tornare a un messaggio precedente e rieditarlo. Crea un “ramo” della conversazione — utile quando un prompt non ha dato il risultato sperato e vuoi riformulare senza perdere tutto.

16. Conclusioni: perché la CLI e non solo la chat

Dopo aver affrontato installazione, comandi, Plan Mode, CLAUDE.md, Skill e tutto il resto, resta una domanda legittima che vale la pena esplicitare: *perché usare Claude Code CLI quando posso semplicemente incollare il codice in una chat del browser?*

La chat resta uno strumento validissimo, e anzi in alcuni scenari è la scelta più efficace. Ma tre differenze fanno della CLI uno strumento qualitativamente diverso, non solo una variante del canale.

1. Contesto persistente: smettere di presentarsi ogni volta

Nella chat tradizionale, ogni nuova sessione parte da zero. Il modello non sa nulla del tuo progetto: le convenzioni di naming, lo stack, le regole invalicabili, i comandi di build. Ogni volta devi ri-spiegare, oppure sperare che Claude indovini dal codice che incolli.

Con la CLI, `CLAUDE.md` **è un contratto permanente**. Viene letto automaticamente a ogni sessione, è gerarchico (globale utente, monorepo, progetto), e puoi rigenerarlo con `/init` ogni volta che il progetto evolve. I comandi `--continue` e `--resume` ti permettono di riprendere conversazioni interrotte giorni prima nello stesso stato di contesto.

Questo cambia il modo in cui pensi al tuo ambiente di sviluppo: invece di riconfigurare mentalmente l'AI a ogni apertura, configuri una volta e lavori. La stessa logica dei file `.editorconfig`, `.eslintrc`, `.gitignore` che consolidano le regole del progetto una volta per tutte.

2. Autonomia agentica: esegue, non solo suggerisce

Nella chat, Claude produce testo. Tu sei il ponte umano: copi il codice nell'editor, salvi, apri il terminale, esegui il test, leggi l'errore, torni in chat, incolli l'errore, aspetti la correzione, copi la correzione, incolli nell'editor, e così via. Ogni passaggio è un'interruzione del flusso.

Nella CLI, Claude **è** il terminale. Legge i file, esegue i comandi, vede gli errori, fa commit, apre PR, interroga database via MCP, naviga cartelle, lancia test. Il ciclo "scrivi → testa → correggi" diventa una conversazione continua senza uscire dallo strumento:

```
Tu:      "Il test auth/login.test.js fallisce. Capisci perché e correggi."

Claude:  [legge il test]
         [legge il codice sotto test]
         [esegue npm test -- auth/login.test.js]
         [analizza l'output]
         [identifica il bug]
         [modifica il codice]
         [riesegue il test]
         [tutto verde]
         "Corretto. Il problema era nella gestione del token expiry.
          Ho modificato validateToken() alle righe 34-38."
```

Questa autonomia ha un rovescio della medaglia — motivo per cui ci sono capitoli interi sulla sicurezza e su Plan Mode — ma quando ben gestita moltiplica la produttività in modo non lineare. Non fai una cosa più veloce: fai cose che in chat semplicemente non faresti perché il costo di orchestrazione manuale è troppo alto.

3. Integrazione nel workflow reale

Lo sviluppo professionale non è solo scrivere codice: è git, test suite, linting, CI/CD, code review, dipendenze, ambienti. La chat vive **accanto** a questo workflow; la CLI vive **dentro**.

Git nativo. Claude Code fa commit, apre branch, risolve merge conflict, scrive commit message Conventional Commits, gestisce stash. Non gli spieghi il diff: lo legge direttamente da `git diff`.

Test e lint in loop. La CLI esegue la test suite, legge gli errori del linter, riprova finché non passa. Non c'è copia-incolla tra finestre, non c'è "aspetta che ti mando l'output".

CI/CD headless. Il flag `-p` trasforma Claude in un tool da pipeline:

```
claude -p "Review the changes in this PR and flag any security issues" \
       --output-format json > review.json
```

Inserisci questo step in un workflow GitHub Actions e hai code review AI automatica a ogni push. Prova a fare la stessa cosa con una chat in browser.

Quando la chat resta la scelta giusta

Per onestà: ci sono casi in cui aprire chat.anthropic.com è la mossa migliore:

- **Brainstorming concettuale** senza codice specifico — "Quali pattern posso usare per implementare un feature flag system?"
- **Apprendimento di un framework nuovo** — ti serve la pedagogia, non l'esecuzione
- **Domande architetturali astratte** — "Vale la pena introdurre CQRS in questo contesto?"
- **Revisione di singoli snippet** da codice che non hai localmente
- **Discussioni con Claude su argomenti non-coding** — scrittura, analisi documenti, pianificazione

La regola pratica: se la risposta è **"codice da integrare nel mio progetto"**, usa la CLI. Se la risposta è **"un'idea, un principio, una spiegazione"**, la chat basta.

In sintesi

Claude Code CLI non è "Claude-in-chat con un'interfaccia diversa". È uno strumento agentico che trasforma un assistente linguistico in un **collega junior operativo**: può fare cose, non solo consigliarle. Per chi sviluppa professionalmente, la differenza è la stessa che passa tra avere un consulente che manda email e avere uno stagista al tavolo accanto. Entrambi utili, contesti diversi.

Il mio consiglio, se stai iniziando: installa Claude Code, prova un progetto piccolo e non critico, scrivi un `CLAUDE.md` decente, usa sempre Plan Mode per i task non banali, e dopo una settimana valuta. La curva è ripida i primi due giorni, poi si appiana.

Chi sono

Mi chiamo **Maurizio Pelizzone**, sono Senior Software Architect, Co-titolare di **Mavida snc** (Torino) dal 2001 e formatore tecnico specializzato nell'ecosistema WordPress.

In oltre vent'anni ho progettato e portato in produzione più di 200 progetti web per clienti nazionali, specializzandomi in architetture WordPress enterprise, sviluppo plugin e temi custom, migrazioni di piattaforme legacy e ottimizzazione performance/security. Dal 2010 sono **Speaker ricorrente ai WordCamp italiani** (Milano, Bologna, Torino, WordCamp Italia), con talk su Custom Post Types, Hardening & Security, Solutions Architecture e Full Site Editing.

Dal 2023 ho aggiunto un "superpotere" al mio workflow: l'Intelligenza Artificiale, usandola come leva strategica per scrivere codice più pulito, scalabile e intelligente. Non come sostituto del mestiere, ma come amplificatore: vent'anni di PHP, WordPress e architetture software mi permettono di distinguere un suggerimento brillante da una scorciatoia rischiosa, e di dirigere l'AI verso soluzioni che reggono il peso della produzione.

Oggi affianco alla consulenza tecnica per PMI e aziende nazionali un'attività di **formazione su AI tools, prompt engineering, vibe coding e automazione con N8N**, convinto che la combinazione tra esperienza artigianale del codice e utilizzo consapevole dell'AI sia la direzione in cui il nostro mestiere sta evolvendo.

Questa guida nasce da quella convinzione: uno strumento come Claude Code non sostituisce lo sviluppatore, ma cambia il modo in cui lavora. Per usarlo bene servono le stesse virtù di sempre — rigore, curiosità, capacità di verificare — più un po' di disciplina nuova.

📧 maurizio@mavida.com — 🔗 linkedin.com/in/mauriziopelizzone — 🌐 maurizio.mavida.com

Allegato A — Glossario

Termini ricorrenti nella guida e nell'ecosistema Claude Code, utili come riferimento rapido.

Agente (agentic) — Sistema AI capace di eseguire azioni nel mondo reale (comandi, modifiche file, chiamate API), non solo di produrre testo. Claude Code è un agente a differenza della chat classica.

CLAUDE.md — File Markdown nella root del progetto che contiene contesto persistente: stack, convenzioni, comandi, regole. Letto automaticamente a ogni sessione.

CLI (Command Line Interface) — Interfaccia a riga di comando. Lo strumento `claude` si usa da terminale anziché da browser.

Headless mode — Esecuzione non-interattiva tramite flag `-p`. Claude riceve un prompt, produce output, esce. Usata per CI/CD e automazioni.

MCP (Model Context Protocol) — Protocollo standard per connettere Claude a servizi esterni (GitHub, database, Slack, filesystem controllato). I server MCP espongono tool che Claude può chiamare come funzioni native.

Native installer — Metodo di installazione ufficiale introdotto da Anthropic nel 2025: un comando `curl` o `PowerShell` senza dipendenze da Node.js, con auto-update.

OAuth — Protocollo di autenticazione usato al primo avvio di `claude`. Apre il browser, logghi con l'account Anthropic, la sessione persiste.

Plan Mode — Modalità read-only attivata con `Shift+Tab` due volte. Claude analizza e propone un piano ma non modifica nulla finché non lo approvi.

Plugin — Pacchetto distribuito tramite marketplace che estende Claude Code con slash command, agent e skill. Gestiti con `claude plugin install`.

Prompt engineering — Arte di formulare istruzioni efficaci per un LLM. Per Claude Code significa definire obiettivi, vincoli e criteri di verifica in modo che il modello produca output prevedibili.

Prompt injection — Attacco in cui istruzioni malevole vengono iniettate in file, commenti o risposte di servizi esterni per manipolare il comportamento dell'AI.

REPL (Read-Eval-Print Loop) — Ciclo interattivo leggi-esegui-stampa. La sessione interattiva di Claude Code è un REPL.

Sessione — Conversazione in corso con Claude Code, persistente tra i riavvii. Ogni sessione ha il proprio contesto e cronologia.

Skill — Modulo specializzato (cartella con `SKILL.md`) che Claude attiva automaticamente quando la descrizione della skill matcha il contesto del task. Non si invocano con slash command.

Slash command — Comando che inizia con `/` dentro una sessione interattiva (es. `/init`, `/compact`, `/plan`). Diversi dai flag di lancio che iniziano con `--`.

Subagent — Istanza isolata di Claude creata dal tool `Task` per eseguire ricerche o task specializzati senza "sporcare" il contesto della sessione principale.

Token — Unità di misura del testo per un LLM (approssimativamente 4 caratteri in inglese, un po' meno in italiano). I costi API sono calcolati in token di input e output. Claude Code usa token ogni volta che legge un file, riceve un prompt o produce una risposta.

Vibe coding — Termine diventato popolare nel 2024-2025 per descrivere lo stile di sviluppo AI-assistito: invece di scrivere codice manualmente, si scrive un prompt strutturato che descrive cosa deve fare, e l'AI genera l'implementazione.

WSL2 (Windows Subsystem for Linux) — Ambiente Linux integrato in Windows 10/11. Consigliato per usare Claude Code su Windows evitando molti problemi di compatibilità.

Allegato B — Fonti

Per approfondire o verificare specifiche aggiornate:

Documentazione ufficiale Anthropic:

- **Panoramica Claude Code:** https://docs.claude.com/en/docs/claude-code/overview
- **CLI reference:** https://code.claude.com/docs/en/cli-reference
- **Setup e installazione:** https://code.claude.com/docs/en/setup
- **Interactive mode e scorciatoie:** https://code.claude.com/docs/en/interactive-mode
- **Cheatsheet ufficiale:** https://support.claude.com/en/articles/14553413-claude-code-cheatsheet
- **Repository GitHub:** https://github.com/anthropics/claude-code
- **Pacchetto npm:** https://www.npmjs.com/package/@anthropic-ai/claude-code

Risorse della community citate nella guida:

- **Caveman skill (Julius Brussee):** https://github.com/JuliusBrussee/caveman

---

Guida redatta da Mavida snc — aprile 2026