Paolo Dalprato

Vibe coding con Claude Desktop

Ultimo aggiornamento: 4 aprile 2026

Creare software senza saper programmare, fino a poco tempo fa era uno slogan pubblicitario, oggi è una possibilità concreta. Il *vibe coding*, termine coniato da Andrej Karpathy nel febbraio 2025, descrive un approccio alla creazione di software in cui si comunica all'intelligenza artificiale ciò che si desidera ottenere, e l'AI scrive il codice necessario.

Questo manuale è rivolto a professionisti che utilizzano già Claude e vogliono iniziare a costruire strumenti, automazioni e applicazioni per il proprio lavoro. Non è necessaria alcuna esperienza di programmazione, i requisiti necessari sono infatti la capacità di analizzare assieme al saper descrivere con chiarezza ciò che si vuole realizzare.

Da dove nasce questo manuale

Il manuale nasce da un'esperienza diretta, nel corso degli ultimi mesi ho utilizzato il vibe coding con Claude per costruire una serie di progetti di natura molto diversa tra loro, dalle applicazioni web ai server MCP, dai simulatori educativi agli strumenti di automazione.

Tra i progetti realizzati ci sono un'applicazione web in React per la navigazione strutturata di scenari decisionali, server MCP per l'integrazione con servizi esterni, un sistema completo di documentazione con generazione automatica di PDF, simulatori interattivi per la formazione, skill personalizzate per Claude e strumenti utility per l'elaborazione di documenti. Nessuno di questi progetti è stato sviluppato con competenze di programmazione pregresse. Tutti sono nati dalla collaborazione con Claude Desktop, seguendo il metodo che questo manuale descrive.

Cosa contiene questo manuale

Il percorso si sviluppa in tre blocchi. Il primo introduce il fenomeno del vibe coding, da dove nasce, cosa significa in pratica, come si colloca nel panorama più ampio dell'uso dell'AI per lo sviluppo software. Il secondo blocco entra nell'operatività con la descrizione dell'ambiente di lavoro, le tre modalità di Claude Desktop (Chat, Cowork e Code), gli strumenti da configurare, il metodo per progettare prima di costruire. Il terzo blocco è interamente pratico, dai progetti guidati di complessità crescente alle best practice suggerite, con attenzione alla sicurezza e alla qualità, e una prospettiva su cosa viene dopo il primo esperimento riuscito.

A chi si rivolge

Il manuale è pensato per professionisti che non sono sviluppatori ma vogliono acquisire la capacità di creare strumenti digitali per il proprio lavoro. Non insegna a programmare ma a collaborare con un'AI che programma al posto nostro, mantenendo il controllo sul risultato.

Sotto il cofano

Il vibe coding si basa su un principio semplice, i modelli linguistici di grandi dimensioni (LLM) sono diventati sufficientemente capaci da generare codice funzionante a partire da descrizioni in linguaggio naturale. Claude Desktop può accedere ai file sul computer, eseguire comandi e interagire con servizi esterni, trasformando una conversazione in un progetto software funzionante.

Indice

- Chiarirsi le idee
- La fase di progettazione
- Descrivere ciò che si vuole
- Il metodo spec.md + todo.md
- Il file CLAUDE.md

Il primo progetto

- Scegliere il progetto giusto
- Preparazione
- Costruzione guidata
- Il ciclo fondamentale
- Cosa abbiamo imparato

Progetti più complessi

- Un sistema di documentazione
- Un'arena di discussione tra agenti
- Cosa cambia quando la complessità cresce

Lavorare bene con Claude

- Gestire gli errori
- Mantenere il contesto
- Ottenere risposte migliori
- Quando Claude non fa quello che dice d'avere fatto
- Salvare il lavoro

Sicurezza e qualità

- Quattro livelli di attenzione
- Cosa può andare storto
- I controlli che si possono fare
- La qualità di ciò che non si vede

Condividere e pubblicare

- Pubblicare il codice su GitHub
- Rendere il progetto accessibile
- Quanto costa davvero
- Dove andare da qui

Cos'è il vibe coding

Il termine *vibe coding* è stato coniato da Andrej Karpathy, cofondatore di OpenAI ed ex direttore AI di Tesla, con un post su X (https://x.com/karpathy/status/1886192184808149383) il 2 febbraio 2025. La definizione originale descrive un approccio alla creazione di software in cui ci si affida completamente all'AI, si accettano le modifiche senza leggere il codice generato e si lascia che il progetto cresca oltre la propria comprensione tecnica.

È da considerare insieme una proposta e una provocazione, il metodo infatti non si esaurisce nel chiedere alla macchina ma è il punto di partenza di un processo.

Da un tweet a un fenomeno globale

Il post originale di Karpathy su X raccoglie milioni di visualizzazioni in pochi giorni. L'idea è semplice e radicale allo stesso tempo, l'AI si è evoluta così tanto da essere capace da scrivere codice funzionante, quindi tanto vale lasciarla fare. Nel post Karpathy racconta di accettare tutte le modifiche generate dall'AI senza leggerle, di incollare i messaggi di errore senza aggiungere commenti e di lasciare che il progetto cresca oltre la sua comprensione.

Il concetto si diffonde in tre fasi distinte. Nei primi due mesi, tra febbraio e marzo 2025, la fase di viralità coinvolge il mondo delle startup, tanto che Merriam-Webster inserisce "vibe coding" nel suo dizionario di slang.

La seconda fase, quella della legittimazione, si estende da aprile a novembre 2025. Il CEO di Google Sundar Pichai dichiara pubblicamente di fare vibe coding con Replit, descrivendolo come un'esperienza piacevole e sorprendente. A novembre 2025 il Collins Dictionary nomina "vibe coding" parola dell'anno (https://blog.collinsdictionary.com/language-lovers/collins-word-of-the-year-2025-ai-meets-authenticity-as-society-shifts/), riconoscendo il passaggio da gergo tecnico a fenomeno culturale.

La terza fase arriva tra la fine del 2025 e l'inizio del 2026, ed è quella della maturazione critica. Gli studi sulla sicurezza del codice generato dall'AI rivelano percentuali allarmanti di vulnerabilità. Gli incidenti in produzione si moltiplicano. Lo stesso Karpathy, nel febbraio 2026, dichiara il vibe coding "superato" e propone un concetto nuovo, quello di agentic engineering (https://x.com/karpathy/status/2019137879310836075), che enfatizza la supervisione umana sugli agenti AI piuttosto che la cieca accettazione dei risultati.

Questa evoluzione in tre fasi racconta molto di come il mondo tecnologico elabora le novità, dall'entusiasmo iniziale alla scoperta dei limiti fino alla ricerca di un equilibrio. Il manuale segue un percorso analogo, partendo dall'attrattiva immediata del vibe coding per arrivare a un approccio più consapevole.

Lo spettro del vibe coding

Il vibe coding non è una pratica monolitica, nella pratica infatti esiste uno spettro ampio di approcci, dal completo affidamento all'AI fino alla supervisione attenta di ogni riga generata. Capire dove ci si posiziona su questo spettro è fondamentale per gestire rischi e aspettative.

In pratica lo spettro si articola su tre posizioni principali. Nella prima posizione l'AI funziona come assistente di scrittura, genera il codice ma l'utente lo rivede e lo comprende. Nella seconda posizione l'AI opera come un collega junior, produce soluzioni complete che l'utente valuta nel loro funzionamento senza necessariamente leggere ogni riga. Nella terza posizione, quella del vibe coding puro, l'utente descrive il risultato desiderato e accetta ciò che l'AI produce senza entrare nel merito tecnico.

I rischi si concentrano in modo prevedibile. Nella prima posizione sono minimi, perché la comprensione del codice consente di individuare problemi. Nella seconda posizione i rischi crescono, soprattutto per questioni non visibili nel funzionamento esterno, come vulnerabilità di sicurezza o gestione inefficiente dei dati. Nella terza posizione i rischi sono massimi, perché l'assenza di verifica rende invisibili anche problemi gravi.

Questo manuale parte dalla terza posizione, quella più naturale per un non-sviluppatore, e accompagna progressivamente verso la seconda. L'obiettivo non è trasformare il lettore in uno sviluppatore, ma fornirgli gli strumenti per valutare ciò che l'AI produce e per riconoscere le situazioni che richiedono attenzione.

Perché è rilevante per i non-sviluppatori

Per creare codice fino al 2024 servivano competenze di programmazione o un budget per commissionare il lavoro a chi le possedeva. Oggi un professionista in qualsiasi campo può descrivere uno strumento di cui ha bisogno e vederselo costruire in tempo reale.

Kevin Roose (https://www.nytimes.com/2025/02/27/technology/personaltech/vibecoding-ai-software-programming.html), giornalista del New York Times, ha introdotto il concetto di *software for one* per descrivere questa nuova categoria di software. Si tratta di strumenti creati da una singola persona per risolvere un proprio problema specifico, senza alcuna pretesa di distribuzione commerciale. Un foglio di calcolo non basta, un'applicazione professionale è eccessiva, e allora si costruisce esattamente ciò che serve. È forse l'applicazione più immediata del vibe coding per i professionisti, e nella pratica è spesso il punto di partenza.

L'esperienza di chi lavora con questo approccio su progetti di natura diversa suggerisce però che il quadro è più ampio di quanto il termine "vibe coding" lasci intendere. C'è un aspetto che spesso sfugge nella discussione, cioè il fenomeno non riguarda solo la programmazione. Delle due parole che compongono il termine, quella che porta il peso concettuale è "vibe", non "coding". Il *coding* è il dominio di applicazione, il *vibe* è il metodo. E quel metodo, la delega alla macchina di un compito produttivo con gradi variabili di specifica e di controllo, si applica già oggi alla scrittura, al design, all'analisi dati, alla produzione di contenuti.

Chi lavora con l'AI per scrivere testi si muove sullo stesso spettro descritto nella sezione precedente. Al livello più basso c'è chi incolla un prompt generico e pubblica il risultato senza rileggere. Al livello intermedio c'è chi dà istruzioni più precise, rilegge e aggiusta. Al livello più alto c'è chi configura l'AI con istruzioni di stile precise, fornisce specifiche dettagliate e rivede ogni passaggio.
Come si vede è lo stesso identico spettro del vibe coding, applicato a un dominio diverso.

Questa osservazione ha una conseguenza pratica importante per il lettore di questo manuale. È probabile che stia già utilizzando un approccio "vibe" nel proprio campo professionale, senza chiamarlo così. Il passaggio al vibe coding è dunque un'estensione di un metodo già familiare, non l'adozione di qualcosa di completamente nuovo.

La differenza critica nel passare da un dominio all'altro sta nel tipo di controllo che si è in grado di esercitare. Il controllo sul lavoro dell'AI opera su due livelli distinti. Il primo è l'analisi del meccanismo, leggere il codice generato, comprenderne la struttura, individuare vulnerabilità o inefficienze. Questo richiede competenza tecnica nel dominio e uno sviluppatore esperto lo fa naturalmente, così come uno scrittore professionista individua i problemi strutturali in un testo.

Il secondo livello è il test del risultato. L'applicazione fa quello che è stato chiesto? Funziona con input diversi? Produce effetti non richiesti? Questo tipo di controllo non richiede competenza di programmazione ma richiede sapere cosa si è chiesto, sistematicità nel provare casi diversi e attenzione agli effetti collaterali. Il test del risultato è sempre importante, ma diventa lo strumento di controllo principale quando non si è in grado di analizzare il meccanismo sottostante. Meno si padroneggia il dominio, più il test sistematico del risultato diventa essenziale. Per chi si avvicina al vibe coding senza esperienza di programmazione, imparare a testare efficacemente è una competenza centrale, ed è uno dei percorsi principali di questo manuale.

Va anche notato che il vibe coding non è un approccio riservato ai principianti. Sviluppatori esperti lo hanno adottato con entusiasmo, riconoscendo che ridefinisce il modo di lavorare di chi il codice lo sa già scrivere. La competenza non diventa superflua ma cambia forma, dal saper scrivere codice al saper dirigere un'AI che lo scrive.

Le aspettative, tuttavia, vanno calibrate. La complessità dei progetti raggiungibili si distribuisce su livelli distinti. Progetti semplici come landing page, calcolatrici specializzate e utility a funzione singola si completano in ore. Progetti di media complessità come dashboard, applicazioni CRUD e prototipi di SaaS richiedono giorni o settimane di lavoro. Progetti complessi con autenticazione, pagamenti e integrazioni multiple richiedono settimane di impegno costante e restano oltre il perimetro sicuro del vibe coding senza supervisione professionale, come pure le applicazioni enterprise e tutto ciò che gestisce dati sensibili.

La distanza tra "funziona per me sul mio computer" e "è pronto per essere usato da altri" è il punto che più spesso viene sottovalutato. Un'applicazione può comportarsi correttamente in fase di test e presentare problemi con dati reali, con utenti simultanei, con condizioni impreviste. Questo non deve scoraggiare ma è un passaggio da valutare con molta attenzione, ciascun livello di complessità ha il suo contesto d'uso legittimo.

Dal vibe coding all'agentic engineering

L'evoluzione del pensiero di Karpathy è istruttiva, avendo proposto come successore naturale del vibe coding quello che ha chiamato *agentic engineering*. La definizione che ne dà è precisa, "agentic" perché nella pratica corrente non si scrive più il codice direttamente nel 99% dei casi ma si orchestrano agenti AI che lo fanno, "engineering" per sottolineare che esiste un'arte, una scienza e una competenza specifica nel farlo bene.

Il passaggio di nomenclatura non è cosmetico, il vibe coding evidenzia l'andare oltre il controllo stretto linea per linea di codice ("dimenticati che il codice esiste"), l'agentic engineering enfatizza la supervisione intelligente e di conseguenza il ruolo di regista dell'utente.

Un dettaglio significativo completa il quadro. Per il suo progetto più recente, Nanochat, Karpathy ha scelto di scrivere il codice interamente a mano, osservando che gli agenti AI non funzionavano ancora abbastanza bene per quel tipo di lavoro. L'inventore del vibe coding ha dunque scelto il coding tradizionale quando la posta in gioco era alta. Non è una contraddizione ma una conferma, lo strumento va scelto in base al contesto.

Questo manuale usa "vibe coding" nel titolo perché è il termine che il pubblico cerca e riconosce. Il percorso che propone, tuttavia, accompagna il lettore dal vibe coding verso un approccio più vicino all'agentic engineering, dove la fiducia nell'AI si combina con la capacità di valutare i risultati e di intervenire quando necessario.

L'ambiente di lavoro

Prima di iniziare a costruire qualsiasi progetto, è necessario preparare l'ambiente di lavoro. Questo capitolo si concentra sulla configurazione specifica di Claude Desktop per il vibe coding, non sull'installazione dell'applicazione, per tale aspetto rimando a questo articolo (https://www.ai-know.pro/claude-desktop-installazione-e-configurazione/) sul mio blog.

Claude Desktop come piattaforma di sviluppo

Claude Desktop non è un'interfaccia di chat. O meglio, la chat è una delle tre modalità di lavoro disponibili, ma non l'unica e non sempre la più indicata. L'applicazione è una piattaforma estensibile che può accedere ai file sul computer con modalità diverse, eseguire comandi, interagire con servizi esterni e produrre artefatti complessi. Quando si usa Claude Desktop per il vibe coding, il modello linguistico è solo una parte del sistema, l'altra parte è l'insieme di connessioni e strumenti che gli permettono di agire oltre la conversazione.

Le tre modalità si chiamano Chat, Cowork e Code, e ciascuna corrisponde a un modo diverso di collaborare con l'AI.

Le differenze principali: - **Chat** è l'ambiente interattivo classico di chat, dove si scambia un messaggio alla volta e si mantiene il controllo su ogni passaggio. - **Cowork** è la modalità agentica pensata per il lavoro di sviluppo, dove Claude riceve un compito, pianifica i passaggi, opera sui file locali e restituisce un risultato finito. - **Code** è la modalità simil-terminale orientata allo sviluppo software, progettata per chi lavora con il codice in modo sistematico.

La distinzione è importante per il vibe coding perché ciascuna modalità si posiziona diversamente sullo spettro della delega descritto nel capitolo precedente. Chat favorisce la delega guidata, dove si osserva ogni passaggio e si interviene in ogni step. Cowork favorisce la delega piena, dove si definisce l'obiettivo e si lascia lavorare l'agente. Code propone un contesto ottimizzato per i progetti software, con strumenti specifici come l'integrazione Git e la gestione dei permessi.

Il capitolo successivo entra nel merito di ciascuna modalità con un confronto dettagliato. Per il momento è sufficiente sapere che esistono, che la scelta dell'ambiente influenza il tipo di controllo che si esercita sul lavoro dell'AI, e che le tre modalità non sono equivalenti per ogni tipo di compito.

Nella mia esperienza ho visto che Chat è l'ambiente che uso per scrivere (anche questo manuale lo sto scrivendo in Chat) o per fare brainstorming all'inizio di un progetto importante, per arrivare alle specifiche da dare poi a Cowork, che a sua volta eccelle nel completamento autonomo di compiti strutturati. La ragione plausibile è che l'architettura agentica di Cowork distribuisce le proprie risorse tra pianificazione, coordinamento e esecuzione, mentre Chat concentra tutta la propria attenzione sulla risposta al messaggio ricevuto.

> **Nota**
>
> Al momento della stesura di questo manuale (aprile 2026) Cowork è in *research preview*, la fase in cui Anthropic rilascia una funzionalità completa ma ancora soggetta a cambiamenti nell'interfaccia e nelle funzionalità specifiche. La direzione di sviluppo è consolidata e Cowork è destinato a diventare un ambiente stabile di Claude Desktop.

Per utilizzare Claude Desktop come piattaforma di sviluppo è necessario un piano a pagamento. Il piano Pro costa 20 dollari al mese (17 con abbonamento annuale) e include l'accesso a tutte e tre le modalità, ai progetti, all'integrazione con Google Workspace e ai connettori MCP. Per chi lavora intensivamente, i piani Max (100 e 200 dollari al mese) offrono limiti di utilizzo più alti e accesso anticipato alle novità. Il piano gratuito consente di esplorare la chat e le desktop extension, ma i limiti di utilizzo lo rendono inadeguato per sessioni di lavoro prolungate come quelle richieste dal vibe coding.

Il protocollo MCP

Il *Model Context Protocol* (MCP) è uno dei meccanismi che permettono a Claude Desktop di agire oltre la conversazione. Il suo ruolo varia a seconda della modalità di lavoro. In Chat, MCP è il canale principale attraverso cui Claude accede ai file, esegue comandi e interagisce con servizi esterni, senza server MCP configurati la chat può solo leggere e generare testo. Cowork e Code hanno invece meccanismi propri per accedere ai file e eseguire comandi, e utilizzano MCP come estensione per funzionalità aggiuntive. Il capitolo successivo entra nel dettaglio di queste differenze.

L'analogia più diffusa descrive MCP come "l'USB-C dell'intelligenza artificiale", un connettore universale che permette a qualsiasi applicazione AI di comunicare con qualsiasi strumento esterno attraverso un'interfaccia standardizzata. L'analogia è efficace ma va completata con un dettaglio, così come USB-C funziona perché nessun produttore di laptop ne controlla le specifiche, MCP funziona perché è uno standard aperto e neutrale.

La storia del protocollo è interessante, Anthropic ha rilasciato MCP nel novembre 2024 già come standard aperto, nel giro di un anno il protocollo è stato adottato anche da Cursor, Microsoft Copilot, Gemini, Visual Studio Code e ChatGPT. Nel dicembre 2025 Anthropic ha donato MCP alla Linux Foundation (https://www.ai-know.pro/anthropic-ha-regalato-mcp/), che lo ha inserito nella neonata *Agentic AI Foundation* insieme a contributi di OpenAI, Google, Microsoft, Amazon Web Services e altri. L'ecosistema conta oggi oltre 10.000 server pubblicati e SDK ufficiali in tutti i principali linguaggi di programmazione.

Per chi fa vibe coding questi numeri significano una cosa concreta, quando l'utente ha bisogno di leggere un file, creare una cartella, eseguire uno script o consultare una documentazione esterna, molto probabilmente da qualche parte esiste già un server MCP adatto.

Sotto il cofano

Un server MCP è un programma che gira in locale sulla macchina dell'utente e comunica con Claude Desktop attraverso un protocollo standardizzato. Claude invia richieste al server (ad esempio "leggi il contenuto di questo file") e riceve risposte strutturate. Il server può essere scritto in qualsiasi linguaggio, i più comuni sono Node.js e Python. La comunicazione avviene tramite *stdio*, cioè lo stesso canale che i programmi usano per leggere e scrivere dati nel terminale.

Ogni server espone un insieme di "strumenti" che Claude può invocare durante la conversazione. L'utente non deve preoccuparsi del protocollo in sé, ma deve sapere quali server sono attivi e cosa permettono di fare.

I server MCP essenziali

La quantità di server MCP disponibili può disorientare. Per iniziare a fare vibe coding non servono decine di strumenti ma pochi server ben scelti, sufficienti a coprire le operazioni fondamentali. La selezione che segue è particolarmente rilevante per chi lavora in modalità Chat, dove MCP è il canale principale di interazione con il sistema. In Cowork e Code molte di queste funzionalità sono già integrate, ma i server MCP restano utili per estendere le capacità di base.

Claude Desktop offre due modi per aggiungere server MCP, le Desktop Extension e la configurazione manuale. Le Desktop Extension sono pacchetti preconfigurati che si installano con un clic dalla directory integrata nell'applicazione. La configurazione manuale richiede la modifica di un file JSON e offre il controllo completo su ogni parametro. Per la maggior parte dei server comuni le Desktop Extension sono la scelta più semplice, la configurazione manuale resta necessaria per server personalizzati o non presenti nella directory. Una differenza pratica rilevante riguarda i prerequisiti software, le Desktop Extension includono tutto il necessario per funzionare, mentre la configurazione manuale richiede l'installazione di strumenti aggiuntivi come Node.js, come descritto nella sezione sul file di configurazione.

Partenza: accesso ai file e controllo del sistema

Il primo server da installare in Chat è *Filesystem*, il server ufficiale che permette a Claude di leggere, scrivere e organizzare file nelle cartelle specificate dall'utente. Senza Filesystem o un server equivalente, la modalità Chat non ha accesso al filesystem e può solo operare all'interno della finestra di conversazione. Con Filesystem attivo, può creare file di progetto, modificare codice esistente, organizzare cartelle e produrre documenti, tutto a partire dalla conversazione.

Desktop Commander è un'alternativa a Filesystem da valutare con attenzione visto che può operare ovunque nel computer dell'utente. Oltre alla gestione dei file, Desktop Commander permette a Claude di eseguire comandi nel terminale, gestire processi, effettuare ricerche nei file e modificare documenti con precisione chirurgica. Per il vibe coding è uno strumento particolarmente utile perché consente di eseguire il codice generato, verificare i risultati e correggere gli errori senza uscire dalla conversazione.

Produttività: documentazione e ricerca

Context7 risolve un problema specifico e frequente. I modelli linguistici hanno una data limite oltre la quale le informazioni diventano inaffidabili, e le librerie software si aggiornano continuamente. Context7 permette a Claude di consultare la documentazione aggiornata delle librerie e dei framework utilizzati nel progetto, riducendo il rischio di generare codice basato su API obsolete o su funzioni rinominate.

Claude Desktop include già una funzionalità nativa di ricerca web, sufficiente per la maggior parte delle esigenze durante una sessione di vibe coding. I server MCP di ricerca web aggiungono valore in situazioni specifiche, ad esempio quando si vuole utilizzare un motore di ricerca particolare, quando si ha bisogno di recuperare il contenuto completo di una pagina web e non solo un riassunto, o quando si lavora in modalità Code dove la ricerca nativa potrebbe non essere disponibile. Per chi inizia, la ricerca nativa è più che adeguata.

Strumenti avanzati

Con la crescita dei progetti, possono emergere esigenze più specifiche. Server per l'automazione del browser permettono a Claude di interagire con pagine web, compilare moduli, estrarre dati. Server per database consentono di leggere e scrivere su database locali come SQLite, spesso usato nei progetti di vibe coding per la sua semplicità. Server per servizi cloud collegano Claude a piattaforme come GitHub, Google Drive o Slack.

Questi strumenti avanzati non servono per iniziare. Diventano rilevanti quando un progetto cresce oltre lo stadio del prototipo e richiede integrazioni con sistemi esterni. Il consiglio è di partire con i server essenziali e aggiungere strumenti aggiuntivi quando emerge un bisogno concreto.

Il file di configurazione

I server MCP installati tramite Desktop Extension non richiedono alcuna configurazione manuale, Claude Desktop li gestisce in autonomia. Per i server che non sono disponibili nella directory, o per quelli che richiedono parametri personalizzati, la configurazione avviene attraverso un file JSON che Claude Desktop legge all'avvio.

Il file si chiama `claude_desktop_config.json` e si trova in una posizione diversa a seconda del sistema operativo. Su macOS il percorso è `~/Library/Application Support/Claude/claude_desktop_config.json`, su Windows è `%APPDATA%\Claude\claude_desktop_config.json`, su Linux `~/.config/Claude/claude_desktop_config.json`.

Il modo più rapido per accedere al file è attraverso Claude Desktop stesso. Aprire le impostazioni, selezionare la sezione Developer e fare clic su "Edit Config". Se il file non esiste, viene creato automaticamente.

La struttura del file è un oggetto JSON con una chiave principale `mcpServers`, sotto la quale ogni server è identificato da un nome e configurato con tre proprietà: il comando da eseguire per avviarlo, gli argomenti da passare al comando e, opzionalmente, le variabili d'ambiente necessarie.

Un prerequisito importante per la configurazione manuale è l'installazione di Node.js sulla propria macchina. La maggior parte dei server MCP utilizza il comando `npx` per avviarsi, che fa parte dell'ecosistema Node.js. Le Desktop Extension non hanno questo requisito perché Claude Desktop include un ambiente Node.js integrato, ma per la configurazione manuale è necessario installare Node.js separatamente, scaricandolo dal sito ufficiale nodejs.org (https://nodejs.org/). Per verificare se Node.js è già presente, è sufficiente aprire un terminale e digitare `node --version`.

Un esempio concreto, la configurazione del server Filesystem per dare a Claude accesso alla cartella dei progetti e al desktop:

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "C:\\Users\\nome-utente\\Desktop",
        "C:\\Users\\nome-utente\\Progetti"
      ]
    }
  }
}
```

La voce `nome-utente` va sostituita con il proprio nome utente di sistema. I percorsi indicano le cartelle a cui Claude avrà accesso, ed è possibile aggiungerne o rimuoverne secondo le proprie esigenze.

Per aggiungere più server è sufficiente inserire ulteriori voci dentro l'oggetto `mcpServers`, separate da virgole. L'errore più comune nella configurazione è proprio un errore di sintassi JSON, una virgola mancante o in eccesso, una parentesi non chiusa. Claude Desktop non segnala gli errori nel file di configurazione, semplicemente ignora i server mal configurati. È consigliabile usare un editor con validazione JSON, come Visual Studio Code, per evitare errori silenziosi.

> **Attenzione**
>
> I percorsi nel file di configurazione devono essere assoluti, non relativi. Su Windows i backslash vanno raddoppiati (`\\`) perché il formato JSON usa il backslash come carattere di escape.

Dopo ogni modifica al file di configurazione è necessario chiudere completamente Claude Desktop e riavviarlo. Chiudere la finestra non è sufficiente, occorre uscire dall'applicazione (su macOS tramite il menu Claude → Esci, su Windows dalla system tray).

Per verificare che i server siano correttamente collegati, aprire una nuova conversazione in Claude Desktop, fare clic sul pulsante "+" nella barra inferiore e selezionare "Connettori". La lista mostra tutti i server attivi e gli strumenti disponibili.

Suggerimento

Per i server disponibili nella directory delle Desktop Extension è sempre preferibile l'installazione con un clic rispetto alla configurazione manuale. La configurazione manuale è lo strumento giusto quando serve un server non presente nella directory, quando si sviluppa un proprio server MCP personalizzato o quando è necessario un controllo preciso sugli argomenti e le variabili d'ambiente.

Chat, Cowork o Code?

Claude Desktop offre tre modalità di lavoro. Sono le tre tab visibili nella parte superiore dell'applicazione, e scegliere quella giusta è una delle prime competenze operative da acquisire.

Le tre modalità a confronto

Le tre tab non sono varianti cosmetiche della stessa interfaccia, sono ambienti diversi, con modi diversi di accedere ai file e con livelli diversi di autonomia. I modelli linguistici disponibili sono gli stessi in tutte e tre.

Il modo più rapido per capire la differenza è pensare a tre situazioni concrete. Nella prima si vuole ragionare su come strutturare un progetto, esplorare le alternative, decidere passo dopo passo. Per questo serve **Chat**, dove si procede un messaggio alla volta e si controlla ogni risposta prima di andare avanti. Nella seconda situazione si sa già cosa serve, si possono dare istruzioni complete fin dall'inizio e si può aspettare che Claude finisca il lavoro da solo. Per questo serve **Cowork**. Nella terza situazione si sta sviluppando un'applicazione software e servono strumenti come Git, un sistema di permessi e accesso al terminale. Per questo serve **Code**.

Quello che cambia tra le tre modalità non è il tipo di risultato (testo, codice, documenti) ma quanto si vuole seguire il lavoro di Claude mentre lo fa. Con Chat si guida ogni passaggio. Con Cowork si spiega cosa serve e Claude fa il lavoro da solo. Con Code si lavora fianco a fianco con Claude su un progetto software, con strumenti pensati per quello.

La tabella seguente riassume le differenze principali.

	Chat	Cowork	Code
Come si lavora	Un messaggio alla volta	Si danno le istruzioni e si aspetta il risultato	Simile a un terminale, semi-autonomo
Accesso ai file locali	Tramite server MCP	Integrato (selezione cartella) o tramite MCP	Integrato (cartella di lancio) o tramite MCP
Dove viene eseguito il codice	Sandbox nella chat, o tramite MCP sul sistema locale	Macchina virtuale isolata	Direttamente sul computer
Progetti	Sì	Sì	No
Istruzioni permanenti	Istruzioni di progetto	Istruzioni globali di Cowork + CLAUDE.md	CLAUDE.md + settings.json
Git integrato	No (disponibile tramite MCP)	No	Sì
Sistema di permessi	Basato sui server MCP	Consenso a livello di cartella	Granulare, per singola operazione
Computer use	No	Sì (research preview)	Sì (research preview)
Dispatch (compiti da mobile)	No	Sì	Sì
Attività programmate	No	Sì	No
Plugin	No	Sì	Sì

Chat: la conversazione come metodo di lavoro

Chat è l'ambiente che la maggior parte degli utenti conosce già. Si scrive un messaggio, Claude risponde, si affina la richiesta, Claude aggiusta. Il lavoro procede un messaggio alla volta, e il controllo è massimo perché ogni risposta è visibile e può essere corretta prima di procedere.

Quello che Chat può fare dipende molto dai server MCP installati. Senza server MCP, Chat lavora solo all'interno della finestra di conversazione e può generare testo, analizzare documenti caricati manualmente

e creare contenuti (tabelle, grafici, codice) visibili nella chat. Con server MCP come Filesystem o Desktop Commander, Chat diventa capace di leggere e scrivere file sul computer, eseguire comandi nel terminale, effettuare ricerche e interagire con servizi esterni. La differenza è sostanziale, ed è il motivo per cui il capitolo precedente dedica una sezione intera alla configurazione dei server MCP.

Per il vibe coding Chat va bene in due fasi. La prima è la progettazione, dove si ragiona con Claude su cosa costruire, si esplorano le alternative, si definiscono le specifiche. La seconda è quando serve la costruzione interattiva, dove si genera codice, lo si testa, si correggono gli errori e si ripete il ciclo fino al risultato desiderato, tutto all'interno della stessa conversazione. In Chat il codice generato è visibile in ogni step, le modifiche sono tracciabili messaggio per messaggio e si può intervenire a ogni passaggio.

Nella mia esperienza Chat produce testi di qualità superiore rispetto a Cowork. I due ambienti hanno architetture e modalità di funzionamento diverse, probabilmente la differenza è dovuta a questo. Per la scrittura di contenuti dove la qualità del testo è la priorità, Chat resta l'ambiente preferibile.

I Progetti di Chat sono un'altra funzionalità importante. Un Progetto raggruppa conversazioni, documenti di riferimento e istruzioni personalizzate in un contesto condiviso. Le istruzioni di progetto vengono applicate automaticamente a ogni conversazione all'interno del Progetto, senza doverle ripetere. Per il vibe coding questo significa poter definire una volta le convenzioni del progetto, lo stack tecnologico, le regole di stile del codice, e ritrovarle attive in ogni sessione di lavoro.

Cowork: dare le istruzioni e ricevere il risultato

In Cowork si lavora in modo diverso. Invece di procedere un messaggio alla volta, si descrivono le istruzioni per il lavoro da fare e Claude pianifica da solo i passaggi necessari, li esegue e restituisce il risultato finito. Durante il lavoro si può osservare cosa sta facendo Claude, intervenire per correggere la rotta, oppure semplicemente aspettare che finisca.

L'accesso ai file nativo di Cowork funziona in modo diverso da Chat, con un meccanismo integrato nell'applicazione. Si seleziona una cartella di lavoro nella directory dell'utente attraverso il checkbox "Work in a Folder" nella parte inferiore dell'interfaccia, e Cowork ottiene accesso a quella cartella e a tutte le sue sottocartelle. Claude può leggere, modificare, creare e eliminare file all'interno di quel perimetro. La prima volta che si seleziona una cartella compare una finestra di dialogo che chiede se si vuole consentire a Claude di modificare i file al suo interno, con la possibilità di acconsentire solo per la sessione corrente oppure in modo permanente.

Il codice prodotto da Cowork viene eseguito in un ambiente protetto e separato dal resto del computer. Questo significa che se qualcosa va storto durante l'esecuzione, il danno resta confinato a quell'ambiente. I file creati da Claude vengono poi salvati nella cartella di lavoro scelta.

Attenzione

Su Windows, Cowork accetta solo cartelle che si trovano all'interno della directory utente (`C:\Users\nome-utente`). Cartelle su dischi secondari o in altre posizioni vengono rifiutate, è una scelta di sistema. Per lavorare su cartelle al di fuori della directory utente è necessario utilizzare un server MCP come Filesystem o Desktop Commander.

Sotto il cofano

L'ambiente protetto di Cowork è tecnicamente una macchina virtuale (VM) Linux che gira sul computer dell'utente. Claude esegue il codice dentro la VM e poi trasferisce i risultati al filesystem locale. Questa separazione impedisce che un errore nel codice generato possa danneggiare file o configurazioni del sistema operativo principale, ovviamente se non si sono attivati server MCP per accedere a tutto il computer locale.

Cowork include diverse funzionalità aggiuntive. I plugin aggiungono capacità specifiche per ruolo e settore. Le attività programmate permettono di definire compiti ricorrenti che Claude esegue automaticamente secondo un calendario, a condizione che il computer sia acceso e Claude Desktop sia aperto. I Progetti di Cowork raggruppano compiti correlati in spazi di lavoro persistenti con file, istruzioni e memoria propri, come i Progetti di Chat.

Cowork ha anche un proprio sistema di istruzioni permanenti. Le *istruzioni globali*, configurabili da Impostazioni > Cowork, vengono applicate a ogni sessione e permettono di stabilire regole di comportamento valide sempre, ad esempio il formato di output preferito, il tono della comunicazione o le regole da seguire prima di eliminare file. In aggiunta, Cowork legge automaticamente i file nella cartella di lavoro selezionata, compresi eventuali file di istruzioni come `CLAUDE.md`, e li usa come contesto per il lavoro da svolgere.

La funzionalità *computer use*, disponibile in research preview, permette a Claude di interagire direttamente con il desktop, cioè aprire applicazioni, navigare il browser, compilare fogli di calcolo. Quando è attiva, Claude opera al di fuori dell'ambiente protetto e interagisce con il sistema reale, il che richiede cautela aggiuntiva. Claude chiede il permesso prima di accedere a ciascuna applicazione, e alcune applicazioni sensibili (piattaforme finanziarie, sanitarie) sono bloccate per impostazione predefinita.

Dispatch permette di assegnare compiti a Cowork dal telefono. Claude lavora sul desktop mentre si è altrove, e invia una notifica quando il compito è completato. Il risultato è consultabile dalla stessa conversazione sia da mobile che da desktop. È attivabile anche in Code.

Per il vibe coding, Cowork è l'ambiente naturale quando si sa già cosa serve e le istruzioni si possono dare complete fin dall'inizio. Organizzare file, generare documentazione a partire da template, creare report da dati esistenti, applicare trasformazioni a insiemi di documenti sono tutti compiti in cui funziona bene dare le istruzioni e aspettare il risultato.

Code: lo sviluppo software nel terminale

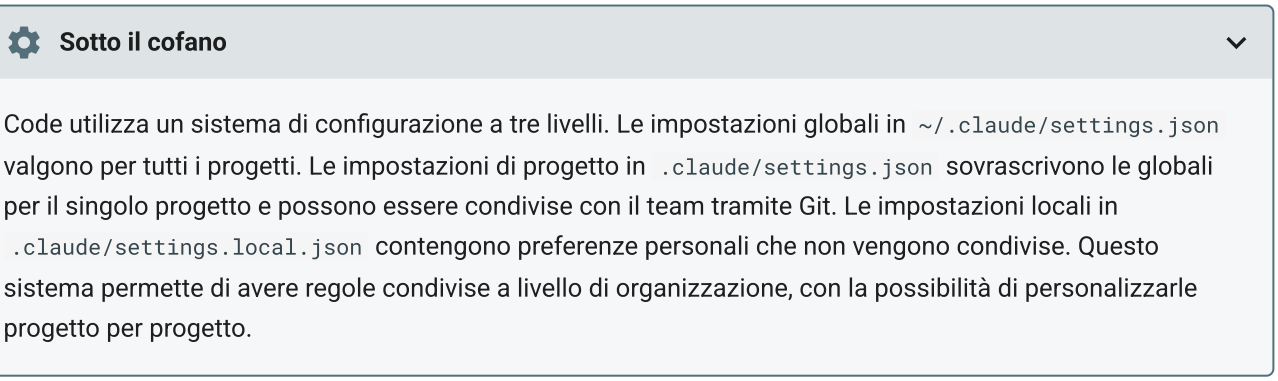
Code è lo strumento di sviluppo software da riga di comando, all'interno dell'interfaccia grafica di Claude Desktop. Per chi ha familiarità con il terminale è l'ambiente più diretto per lo sviluppo software. Per chi non ce l'ha, l'interfaccia grafica ne riduce la barriera d'ingresso, anche se la logica operativa resta quella del terminale.

A differenza di Cowork, Code non usa un ambiente protetto separato. Lavora direttamente sui file del computer, nella cartella da cui viene lanciato e nelle sue sottocartelle. Ha strumenti integrati per leggere, modificare e cercare file senza bisogno di server MCP, e può eseguire comandi con i permessi dell'utente del sistema operativo. Questo lo rende più veloce ma anche più rischioso, perché le modifiche sono immediate e reali.

Per gestire questo rischio, Code ha un sistema di permessi. Nella modalità predefinita chiede conferma prima di ogni modifica ai file o esecuzione di comandi. Si possono allentare questi controlli progressivamente, fino alla modalità che approva automaticamente le modifiche ai file o quella che elimina completamente le richieste di conferma. Quest'ultima è pensata per ambienti controllati e va usata con estrema cautela sul proprio computer di lavoro.

Code integra Git in modo nativo, il che significa che le modifiche al codice possono essere tracciate, confrontate e annullate senza uscire dall'ambiente. Per un non-sviluppatore che si avvicina al vibe coding, Git è uno strumento da imparare gradualmente, e il capitolo dedicato alla sicurezza ne tratterà gli aspetti essenziali.

Sia Code che Cowork utilizzano il file `CLAUDE.md`, un documento nella cartella di lavoro che contiene istruzioni permanenti per il progetto. Claude lo legge automaticamente all'inizio di ogni sessione e ne segue le indicazioni. È l'equivalente delle istruzioni di progetto in Chat, ma vive nella cartella del progetto come un file qualsiasi. In Code il file viene tipicamente versionato insieme al codice tramite Git.



Come si integrano le tre modalità

Le tre modalità non sono compartimenti stagni. Una conversazione avviata in Chat può produrre specifiche che vengono poi realizzate in Cowork o Code. Dispatch permette di inviare compiti a Cowork o a Code da

qualsiasi dispositivo, trasformando il telefono in un telecomando per il desktop. I server MCP configurati sono disponibili in tutte e tre le modalità.

Nella mia esperienza il modo di lavorare più naturale segue due fasi. La prima è sempre in Chat, dove si ragiona su cosa si vuole ottenere, si esplorano le alternative e si arriva a un insieme di istruzioni chiare. La seconda fase è l'esecuzione, e qui la scelta dipende dal progetto. Per compiti ben definiti si passa a Cowork, per sviluppo software si usa Code, e per progetti dove si vuole mantenere il controllo su ogni passaggio si resta in Chat.

La cosa più importante è sempre la stessa, indipendentemente dalla modalità scelta. Dedicare tempo a definire con chiarezza cosa si vuole ottenere, perché è lì che il contributo umano conta di più.

Guida alla scelta

La scelta della modalità si riduce a poche domande pratiche.

Se si vuole ragionare insieme a Claude, vedere cosa propone, aggiustare, ripetere, Chat è l'ambiente giusto. Questo vale per la progettazione, il brainstorming, la scrittura di contenuti dove la qualità testuale è la priorità, e qualsiasi attività dove si vuole approvare ogni passaggio prima di procedere.

Se si sa già cosa serve e le istruzioni si possono dare complete fin dall'inizio, Cowork è la scelta più efficiente. Claude pianifica, esegue e consegna il risultato mentre si fa altro.

Se il compito è sviluppo software e si ha familiarità con il terminale e Git, Code offre l'ambiente più specializzato. Per un non-sviluppatore che si avvicina al vibe coding, Code diventa rilevante quando i progetti crescono in complessità.

Per molti progetti di vibe coding, soprattutto nelle fasi iniziali, Chat con server MCP ben configurati è sufficiente. Aggiungere Cowork e Code al proprio modo di lavorare è una progressione naturale, non un requisito.

> **Suggerimento**
>
> Non serve usare tutte e tre le modalità fin dall'inizio. Conviene acquisire padronanza di Chat, poi passare a Cowork quando emerge il bisogno di delegare compiti completi, e infine esplorare Code quando i progetti richiedono strumenti di sviluppo dedicati.

Pensare prima di costruire

La tentazione più comune nel vibe coding è iniziare subito a chiedere a Claude di scrivere codice. È comprensibile, la tecnologia sembra magica e si vuole vedere subito un risultato, ma un buon risultato è l'ultimo elemento di una catena di passaggi, tutti necessari per arrivare fino in fondo.

Questo progetto è adatto al vibe coding?

Prima di aprire Claude e iniziare a descrivere un'applicazione, vale la pena fermarsi un momento. Non tutti i progetti sono ugualmente adatti al vibe coding, e saperlo prima evita frustrazioni, rischi e lavoro sprecato.

Non si tratta di una distinzione binaria tra "sì" e "no". Si tratta piuttosto di una scala di rischio crescente. Un'applicazione per uso personale che non tratta dati sensibili è il terreno ideale per il vibe coding. Un servizio a pagamento con autenticazione utenti e dati di terzi richiede competenze professionali che vanno oltre la capacità di descrivere bene un'idea.

Per orientarsi, conviene valutare il progetto da più punti di vista.

Destinazione del progetto. Un'applicazione per uso personale, che gira sul proprio computer e che nessun altro userà mai, è il caso più semplice. Se qualcosa non funziona, l'unica persona coinvolta è chi l'ha costruita. Il rischio cresce quando l'applicazione viene condivisa con altri, anche gratuitamente, perché a quel punto qualcun altro dipende dal suo funzionamento. Cresce ancora se viene venduta o se diventa un servizio a pagamento, perché entrano in gioco aspettative, obblighi e responsabilità.

Dati e sensibilità. Un'applicazione che organizza appunti personali è diversa da una che gestisce contatti di clienti, fatture o cartelle cliniche. Quando un progetto tratta dati di altre persone, si entra nel territorio della normativa sulla protezione dei dati personali (GDPR in Europa), con obblighi specifici che prescindono dal modo in cui l'applicazione è stata costruita. Quando tratta dati finanziari, i requisiti di sicurezza e affidabilità diventano ancora più stringenti.

Account e servizi esterni. Molti progetti utili collegano servizi esistenti: leggere email, pubblicare su un blog, interrogare un database. Ognuno di questi collegamenti usa credenziali, come chiavi API, token di accesso o password, che danno all'applicazione permessi sul proprio account. Un errore nel codice che gestisce queste credenziali può esporre l'account a rischi concreti. Il principio guida è il privilegio minimo, ovvero dare all'applicazione solo i permessi strettamente necessari, mai l'accesso completo a un account quando ne basta uno limitato.

Autenticazione. Se l'applicazione gestisce il login di altri utenti, con username, password e sessioni, la complessità aumenta in modo significativo. La gestione dell'autenticazione è uno degli aspetti più delicati dello sviluppo software, e gli errori in quest'area possono avere conseguenze serie. È una delle situazioni in cui il coinvolgimento di uno sviluppatore esperto è fortemente consigliabile.

Esposizione. Un'applicazione che gira solo sul proprio computer è protetta dalla rete locale. Un'applicazione accessibile da internet è esposta a chiunque, compresi utenti con intenzioni non costruttive. La differenza non è teorica: un'applicazione web pubblica riceve tentativi di accesso automatizzati nel giro di ore dal suo deploy. Il capitolo sulla sicurezza approfondirà questo aspetto.

Affidabilità. Per uno strumento personale, un'interruzione di servizio è un fastidio. Per uno strumento che altri usano nel loro lavoro quotidiano, è un problema serio e la domanda da farsi è cosa succede se l'applicazione smette di funzionare una notte, magari nel weekend? Se la risposta è "niente di grave, la sistemo quando posso", il vibe coding è adeguato. Se la risposta è "persone o processi si bloccano", servono garanzie che il vibe coding da solo non può offrire.

Manutenzione. Ogni applicazione che funziona oggi potrebbe smettere di funzionare domani. Le librerie si aggiornano, le API cambiano, i servizi esterni modificano le loro interfacce. Un progetto personale può restare fermo per mesi senza conseguenze. Un progetto usato da altri richiede attenzione continua, e la persona che lo mantiene deve essere in grado di capire cosa si è rotto e perché.

Impatto sulle persone. Chi costruisce un'applicazione è responsabile di ciò che quell'applicazione fa. Questo vale indipendentemente da come è stata costruita e da dove viene eseguita. Il fatto che un progetto sia personale e giri esclusivamente sul proprio computer non crea una zona franca, né dal punto di vista legale né da quello etico. Ci sono usi che restano illeciti o inaccettabili anche in assenza di qualsiasi esposizione esterna, e il vibe coding non sposta la responsabilità dall'autore alla macchina.

In Italia la Legge 132/2025 (https://www.gazzettaufficiale.it/eli/id/2025/09/25/25G00146/sg) sull'intelligenza artificiale, in vigore dal 10 ottobre 2025, rende espliciti alcuni di questi principi. Ogni contenuto generato o modificato con sistemi di AI deve essere dichiarato come tale. La diffusione di immagini, video o audio alterati con AI senza il consenso della persona rappresentata è un reato. L'uso dell'intelligenza artificiale come strumento per commettere un illecito è una circostanza aggravante. Si tratta di una legge-quadro con implicazioni ampie, e chi costruisce applicazioni che generano o manipolano contenuti fa bene a conoscerne i principi fondamentali.

Quando poi l'applicazione coinvolge altre persone, producendo raccomandazioni, classificazioni o decisioni che le riguardano, la responsabilità cresce ulteriormente. Un errore nel codice di un organizzatore di appunti è un inconveniente. Un errore nel codice di uno strumento che influenza scelte su altri può danneggiare qualcuno concretamente.

La regola pratica che emerge da queste dimensioni è lineare. Un progetto per uso personale, che non tratta dati sensibili e non è esposto a internet, è il candidato ideale per il vibe coding. Man mano che le dimensioni di rischio si sommano, aumenta la necessità di coinvolgere competenze professionali. Non per sostituire il vibe coder, ma per affiancare competenze specifiche dove servono.

Il punto non è scoraggiare dall'iniziare. Al contrario, sapere dove si collocano i confini permette di muoversi con sicurezza all'interno di quei confini, e di chiedere aiuto quando serve.

Chiarirsi le idee

Il vibe coding non inizia con un progetto definito. Inizia con qualcosa di più vago: un problema ricorrente nel proprio lavoro, un'operazione che si ripete sempre uguale, un'intuizione su qualcosa che potrebbe funzionare meglio. Il passaggio da quell'intuizione a un progetto concreto è il primo lavoro da fare, e spesso è il più importante.

Claude può aiutare fin da questo momento. Usare Chat per un brainstorming, descrivendo una situazione che non funziona o un'idea ancora grezza e chiedendo a Claude di fare domande, esplorare possibilità, proporre soluzioni, è spesso il modo in cui un'esigenza sfumata diventa un progetto con una forma riconoscibile. "Passo due ore alla settimana a copiare dati da un foglio Excel all'altro" non è ancora un progetto, ma è il punto di partenza di una conversazione da cui un progetto può emergere.

Quando l'idea ha preso forma, conviene fermarsi a rispondere ad alcune domande prima di iniziare a costruire. Quale problema risolve questa applicazione? Chi la userà? Cosa deve fare, concretamente? Cosa non deve fare? Esiste già qualcosa di simile, e se sì perché non va bene?

Sono domande semplici, ma la loro assenza è la causa più frequente di progetti che partono in una direzione, cambiano rotta più volte e finiscono in un risultato che non soddisfa nessuno. L'AI amplifica questa dinamica perché risponde a qualsiasi richiesta, anche a quelle confuse. Di fronte a istruzioni vaghe non si ferma a chiedere chiarimenti: produce qualcosa, e quel qualcosa è spesso plausibile ma sbagliato nelle premesse.

Oltre al "cosa", vale la pena ragionare anche sul "come", almeno nelle sue linee generali. Anche senza competenze tecniche si possono fare scelte consapevoli. L'applicazione deve funzionare solo sul proprio computer o deve essere raggiungibile da altri? Deve leggere dati da qualche parte, e se sì da dove? Deve produrre un file, mostrare un'interfaccia, inviare una notifica? Queste scelte influenzano il tipo di progetto che Claude costruirà e il livello di complessità che ne deriva.

Non serve un documento formale. Serve la chiarezza mentale che permette di spiegare il progetto a un collega in due minuti, in modo che quel collega capisca cosa si sta costruendo e perché. Se non si riesce a farlo con un essere umano, non si riuscirà a farlo con Claude.

La fase di progettazione

Quando si ha chiaro cosa si vuole costruire, il passo successivo è tradurre quell'idea in un piano che Claude possa seguire. La tentazione è saltare direttamente alla costruzione, ma il tempo investito nella progettazione ne fa risparmiare molto di più nelle fasi successive. Un progetto ben descritto produce meno errori, meno riscritture e un risultato più vicino alle aspettative.

Come si progetta dipende dalla modalità di lavoro scelta.

Pianificare in Chat

Chi lavora in Chat non ha bisogno di strumenti dedicati, perché il dialogo con Claude è già, per sua natura, un processo di raffinamento progressivo. Il punto è rendere questo processo intenzionale.

Prima di chiedere a Claude di costruire qualcosa, conviene descrivere il progetto e chiedere esplicitamente di analizzarlo. Quali componenti servono? Quali problemi potrebbe incontrare? Quale approccio è più adatto? Claude risponde con una proposta, si discute, si aggiusta, si ripete. Il codice viene scritto solo quando il piano è chiaro per entrambi.

La differenza tra "fammi un'app che fa X" e "analizziamo insieme cosa serve per fare X, poi costruiamo" è la stessa differenza tra sperare che vada bene al primo tentativo e sapere cosa si sta costruendo.

Pianificare per Cowork

Cowork lavora in autonomia. Una volta avviato un compito, Claude pianifica, esegue e consegna il risultato senza chiedere conferme intermedie. Questo rende la progettazione ancora più importante, perché tutto ciò che non è stato chiarito prima diventa una decisione che Claude prenderà da solo.

In pratica, la progettazione per Cowork è il lavoro che si fa prima di premere invio. Le istruzioni devono essere più complete e strutturate di quanto servirebbe in Chat, perché non ci sarà la possibilità di correggere il tiro durante l'esecuzione. Il risultato che si ottiene dipende quasi interamente dalla qualità delle istruzioni iniziali.

Il pattern più efficace, già descritto nel capitolo precedente, è usare Chat per la fase di ragionamento e poi passare a Cowork per l'esecuzione. In Chat si definisce cosa serve, si discutono le alternative, si arriva a un piano dettagliato. Quel piano diventa l'istruzione per Cowork.

La modalità pianificazione in Code

Code offre quattro modalità operative, selezionabili dal menu a tendina nella barra inferiore dell'interfaccia. La terza opzione, *Modalità pianificazione*, è pensata esattamente per separare il ragionamento dall'esecuzione.

Quando è attiva, Claude può leggere tutto il progetto, analizzare i file, cercare pattern e proporre una strategia, ma non può modificare nulla. Nessun file viene toccato, nessun comando viene eseguito. Il risultato è un piano strutturato che descrive quali file modificare, in quale ordine e perché.

A quel punto si rivede il piano, si chiede di modificarlo se necessario, e si itera fino a quando è soddisfacente. Solo dopo l'approvazione si passa a una delle modalità operative e si lascia eseguire.

Le quattro modalità, nell'ordine in cui appaiono nel menu, corrispondono a livelli crescenti di autonomia. *Richiedi autorizzazioni* chiede conferma prima di ogni modifica. *Accetta automaticamente le modifiche* esegue senza chiedere. *Modalità pianificazione* analizza senza modificare. *Ignora autorizzazioni* accetta tutto, compresi i permessi di sistema, ed è pensata per ambienti controllati.

Descrivere ciò che si vuole

Nel vibe coding la descrizione del progetto non è un prompt nel senso abituale del termine. Non è una domanda a cui Claude risponde, è una specifica di progetto che Claude traduce in codice. La qualità della specifica determina la qualità del risultato.

Una buona specifica si costruisce in due passaggi. Il primo è un **abstract** che descrive il progetto in modo sintetico, legando fra loro i quattro elementi fondamentali: cosa fa l'applicazione (l'obiettivo), in quale situazione e per chi (il contesto), entro quali limiti deve muoversi (i vincoli) e cosa ci si aspetta di vedere quando funziona (l'output atteso). L'abstract dà a Claude il quadro d'insieme prima dei dettagli, e questo migliora sensibilmente la qualità delle decisioni che prenderà durante la costruzione.

Il secondo passaggio è l'**approfondimento** di ciascun elemento. L'obiettivo diventa una lista di funzionalità concrete. Il contesto spiega chi userà l'applicazione e in quale flusso di lavoro si inserisce. I vincoli specificano tecnologie, formati, servizi da integrare o da evitare. L'output atteso descrive l'interfaccia, il comportamento e i casi d'uso principali.

La differenza si vede con un esempio. Una descrizione vaga come "fammi un'app per gestire i miei appunti" lascia a Claude ogni decisione. Un abstract come "un'applicazione web locale per scrivere appunti in markdown, salvarli come file nella cartella Documenti, cercarli per parola chiave e visualizzarli con la formattazione" è già molto meglio, ma lascia ancora aperte domande importanti. Come si scrivono gli appunti, in un editor dedicato o in un campo di testo semplice? Come si assegna il nome al file, lo sceglie l'utente o viene generato dal titolo? La ricerca è sul titolo o anche sul contenuto?

Una specifica completa risponde a queste domande:

> Un'applicazione web locale per gestire appunti in markdown. L'interfaccia ha due pannelli affiancati: a sinistra l'elenco degli appunti esistenti con un campo di ricerca, a destra l'editor. L'editor usa un campo di testo con evidenziazione della sintassi markdown. Sotto il titolo, un'anteprima dal vivo mostra il risultato formattato. Il nome del file viene generato automaticamente dal titolo dell'appunto, convertendo gli spazi in trattini e rimuovendo i caratteri speciali. I file vengono salvati nella cartella Documenti/Appunti. La ricerca funziona sia sul titolo sia sul contenuto del file.

Questo non significa che la prima descrizione debba raggiungere questo livello di dettaglio. Il vibe coding è un processo iterativo, e Claude può aiutare a raffinare la specifica attraverso il dialogo. Ma partire con almeno un buon abstract riduce drasticamente il numero di iterazioni necessarie.

Un criterio pratico per valutare se la descrizione è abbastanza chiara: immaginarla rivolta a un collega competente ma che non sa nulla del progetto. Se quel collega capirebbe cosa deve costruire, Claude lo capirà. Se quel collega farebbe domande, è meglio rispondere a quelle domande prima di premere invio.

Un ultimo aspetto, facile da trascurare. Descrivere cosa l'applicazione *non* deve fare è spesso altrettanto utile quanto descrivere cosa deve fare. "Non deve richiedere un database esterno", "non deve avere un sistema di login", "non deve modificare file fuori dalla sua cartella" sono vincoli che evitano a Claude di prendere decisioni che poi tocca smontare.

Il metodo spec.md + todo.md

Quando un progetto supera la complessità di una singola richiesta in Chat, conviene scrivere la specifica in un file separato. Due documenti di testo nella cartella del progetto, uno per la specifica e uno per le attività, sono sufficienti a dare struttura al lavoro.

Il file **spec.md** è la versione scritta dell'abstract e dell'approfondimento descritti nella sezione precedente. Contiene la descrizione del progetto, il pubblico a cui è destinato, le funzionalità previste, i vincoli tecnici e qualsiasi decisione già presa. Non ha un formato rigido, ma deve rispondere alle stesse domande: cosa fa, per chi, come, con quali limiti. Più la specifica è chiara, meno ambiguità restano a Claude.

Il file **todo.md** è una lista ordinata di attività, ciascuna abbastanza piccola da poter essere completata e verificata in un singolo passaggio. L'idea è scomporre il progetto in passi che Claude esegue uno alla volta, verificando il risultato di ciascuno prima di passare al successivo.

Un esempio concreto. Per l'applicazione di appunti in markdown descritta nella sezione precedente, il todo.md potrebbe avere questa struttura:

```
## Todo

- [ ] Creare la struttura del progetto (cartelle, file HTML, CSS e JS di base)
- [ ] Implementare l'editor markdown con campo di testo ed evidenziazione sintassi
- [ ] Aggiungere l'anteprima dal vivo sotto l'editor
- [ ] Implementare il salvataggio come file nella cartella Documenti/Appunti
- [ ] Generare il nome del file dal titolo dell'appunto
- [ ] Creare il pannello laterale con l'elenco degli appunti esistenti
- [ ] Aggiungere la ricerca per titolo e contenuto
- [ ] Testare il flusso completo: creare, salvare, cercare, riaprire
```

Il flusso di lavoro segue una sequenza naturale. Si parte dal brainstorming in Chat per definire l'idea e si compila spec.md con la descrizione completa, costruendolo insieme a Claude attraverso il dialogo. Quando la specifica è stabile, la generazione del piano di lavoro si può affrontare in modi diversi a seconda della modalità scelta.

In Chat si può chiedere a Claude di scomporre la specifica in attività ordinate e generare direttamente il todo.md. È un compito di analisi che Claude fa bene, perché ha appena lavorato sulla specifica e ne conosce la struttura. Il todo.md va rivisto e aggiustato, ma partire da una proposta generata è più rapido che scriverlo da zero.

In Cowork si può passare direttamente con il solo spec.md. Cowork crea autonomamente il proprio piano di lavoro prima di eseguire, scomponendo il progetto in passaggi che affronta in sequenza. In questo caso il todo.md esplicito non serve, perché la pianificazione è parte del flusso operativo di Cowork.

In Code si può usare la modalità pianificazione descritta nella sezione precedente. Claude legge il spec.md, analizza il progetto e produce un piano strutturato senza toccare nulla. Si rivede il piano, si itera, e solo dopo l'approvazione si passa all'esecuzione.

La differenza tra le tre strade è nel grado di controllo. Con il todo.md scritto in Chat si decide in anticipo l'ordine e la granularità dei passaggi. Con Cowork si delega anche la scomposizione, fidandosi della qualità della specifica iniziale. Con la modalità pianificazione di Code si rivede e approva il piano prima dell'esecuzione.

Dopo ogni attività completata, conviene verificare che tutto funzioni prima di procedere alla successiva. In Code questo è anche il momento giusto per fare un commit, ovvero salvare una versione del progetto a cui si può tornare se qualcosa va storto nei passaggi successivi. Il capitolo sulla sicurezza spiegherà come farlo in pratica.

Questo metodo non è obbligatorio e non serve per ogni progetto. Per un progetto semplice che si costruisce in una conversazione in Chat, sarebbe eccessivo. Ma quando il progetto ha più di quattro o cinque componenti che devono funzionare insieme, avere una mappa scritta di dove si sta andando e a che punto si è fa la differenza tra un progetto che arriva in fondo e uno che si perde per strada.

Il file CLAUDE.md

Nel capitolo precedente il file CLAUDE.md è stato presentato come documento di istruzioni permanenti, condiviso tra Cowork e Code. Qui il tema è pratico: cosa metterci e come farlo evolvere con il progetto.

CLAUDE.md è un file di testo nella cartella del progetto che Claude legge automaticamente all'inizio di ogni sessione. Contiene istruzioni che restano valide per tutta la durata del progetto, a differenza dei messaggi in chat che si perdono quando la conversazione finisce o viene compattata.

Il contenuto tipico rientra in tre categorie.

Decisioni tecnologiche. Le scelte di fondo del progetto: quale linguaggio, quali librerie, quale struttura di cartelle, quale formato per i dati. Scrivere queste decisioni nel CLAUDE.md evita che Claude proponga alternative diverse a ogni nuova sessione. Se il progetto usa React con TypeScript e i dati vanno salvati in JSON, dirlo nel CLAUDE.md significa non doverlo ripetere ogni volta.

Convenzioni. Le regole che il codice deve seguire: come nominare i file, come organizzare le funzioni, quale stile di formattazione usare, in quale lingua scrivere i commenti. Sono preferenze che non hanno una risposta giusta o sbagliata, ma che devono restare coerenti in tutto il progetto.

Errori da non ripetere. Questa è la categoria più utile e quella che cresce più naturalmente. Ogni volta che Claude commette un errore o prende una decisione sbagliata, aggiungere un'istruzione che lo previene per il futuro trasforma un problema in un miglioramento permanente. "Non usare localStorage, i dati vanno salvati su file" oppure "non modificare mai i file nella cartella config senza chiedere conferma" sono esempi di istruzioni nate da errori concreti.

Un CLAUDE.md per un progetto di vibe coding potrebbe avere questo aspetto:

```
# Progetto: Gestore appunti markdown

## Stack tecnologico
- HTML, CSS e JavaScript vanilla (nessun framework)
- I dati vengono salvati come file .md nella cartella Documenti/Appunti
- Nessun database, nessun server esterno

## Convenzioni
- Nomi dei file in italiano con trattini come separatore
- Commenti nel codice in italiano
- Un file JS per ogni funzionalità principale

## Attenzione
- Non creare mai file fuori dalla cartella del progetto
- Non usare localStorage: i dati vanno salvati su file
- Testare sempre il salvataggio e il caricamento dopo ogni modifica
```

Il file non deve essere lungo né formale. Deve contenere le informazioni che servono a Claude per lavorare bene su questo specifico progetto. All'inizio sarà breve, poche righe sullo stack e le convenzioni di base. Col tempo crescerà, arricchito dalle lezioni apprese durante la costruzione.

In Code il CLAUDE.md ha anche una gerarchia. Il file nella cartella principale del progetto si applica a tutto il progetto. File CLAUDE.md in sottocartelle possono aggiungere istruzioni specifiche per quella parte del codice. Un file globale nella cartella dell'utente si applica a tutti i progetti. Per la maggior parte dei progetti di vibe coding, un singolo file nella cartella principale è sufficiente.

Il primo progetto

È arrivato il momento di costruire qualcosa. Questo capitolo guida passo dopo passo nella realizzazione di un primo progetto completo, dall'idea al risultato funzionante, usando Claude Desktop.

Il progetto è volutamente semplice ma utile. Non un esercizio didattico fine a sé stesso, ma uno strumento che risolve un problema concreto e che si continuerà a usare anche dopo averlo costruito.

Scegliere il progetto giusto

Un buon primo progetto ha alcune caratteristiche precise. Deve risolvere un problema reale, altrimenti mancherà la motivazione per portarlo a termine. Deve essere abbastanza semplice da essere completato in una singola sessione di lavoro, altrimenti la complessità rischia di scoraggiare. Deve produrre un risultato tangibile che si può testare immediatamente, perché il feedback immediato è ciò che rende il processo efficace.

Altrettanto importanti sono le caratteristiche da evitare. Un primo progetto non dovrebbe richiedere account su servizi esterni, perché ogni integrazione è una fonte potenziale di problemi. Non dovrebbe trattare dati sensibili, per i motivi discussi nel capitolo precedente. Non dovrebbe essere esposto a internet, per evitare complicazioni di sicurezza e deployment.

Il progetto ideale è uno strumento locale, che gira sul proprio computer, che risolve un problema che si incontra regolarmente.

Come nasce l'idea

Il progetto raccontato in questo capitolo nasce dalla pratica quotidiana, lavorando allo sviluppo di un'altra applicazione, Claude ha proposto di usare la libreria PyMuPDF4LLM per trasformare i PDF in formato markdown prima di passarli al modello. La riduzione di peso era significativa, perché un PDF è un formato progettato per la resa grafica e il suo peso è dovuto in larga parte alla gestione della formattazione, dei font, del layout e di altri elementi che non sono il testo in sé. La conversione in markdown elimina tutto questo e conserva solo il contenuto testuale strutturato.

L'intuizione successiva è stata che questa conversione non serviva solo per quel progetto specifico. Ogni volta che si carica un PDF come documento di riferimento in una conversazione con un modello linguistico, nella stragrande maggioranza dei casi importa il testo, non la formattazione. Un PDF da 15 megabyte potrebbe diventare un markdown da poche centinaia di kilobyte, con tutto il contenuto rilevante intatto, con un conseguente risparmio di token. Per file particolarmente pesanti o conversazioni particolarmente lunghe, la differenza è concreta.

Il passo finale è stato chiedersi come altri avrebbero potuto trarne vantaggio. La libreria PyMuPDF4LLM si usa da terminale o dentro a script Python, il che la rende inaccessibile a chiunque non abbia dimestichezza con la riga di comando. Un'interfaccia grafica avrebbe trasformato uno strumento per sviluppatori in uno strumento per tutti.

Questo è il percorso tipico di un progetto di vibe coding. Non si parte da un'idea astratta, si parte da un problema incontrato lavorando. La soluzione emerge dal contesto e diventa un progetto quando si riconosce che il problema è condiviso.

Preparazione

Con l'idea chiara, il primo passo è preparare l'ambiente di lavoro seguendo il metodo descritto nel capitolo precedente.

La specifica

La conversazione in Chat parte dalla descrizione del problema e dell'idea di soluzione. In questo caso la richiesta iniziale è stata diretta: un'interfaccia grafica standalone in cui l'utente apre l'applicazione, carica o trascina un file PDF e ottiene il corrispondente file markdown. L'applicazione deve funzionare interamente in locale, senza inviare dati a servizi esterni, e deve essere rilasciabile su GitHub per permettere ad altri di installarla.

Claude ha risposto con tre proposte di architettura. La più pratica era un'applicazione Python con Flask come server locale: l'utente lancia uno script e il browser si apre automaticamente sulla pagina di conversione. Nessuna installazione complessa, nessun servizio cloud, nessuna dipendenza da piattaforme esterne.

Questa è una delle dinamiche più importanti dei primi minuti di un progetto. Claude non va direttamente alla soluzione, propone alternative e lascia che sia l'autore a scegliere. La scelta architetturale è una decisione dell'autore, non dell'AI. In questo caso la decisione si è basata su un criterio pratico: Flask è una tecnologia consolidata, la struttura è semplice, e il risultato è un'interfaccia web accessibile a chiunque sappia usare un browser.

L'installazione delle dipendenze

Il progetto richiede Python e due librerie: PyMuPDF4LLM per la conversione dei PDF e Flask per il server locale. L'installazione avviene con un singolo comando da terminale:

```
pip install pymupdf4llm flask
```

Se Python non è già installato sul computer, è necessario scaricarlo dal sito ufficiale python.org. L'installazione è guidata, l'unica accortezza è assicurarsi di selezionare l'opzione "Add Python to PATH" durante il processo, un dettaglio che evita problemi successivi.

Il comando `pip install` è l'equivalente di installare un'applicazione, si esegue una volta e il risultato resta disponibile. Non è necessario capire cosa fa nel dettaglio, così come non è necessario capire cosa fa l'installazione di qualsiasi altro software.

La scelta della modalità

La costruzione è stata fatta in Cowork. La scelta ha una motivazione precisa. Cowork può creare file, organizzare cartelle e lavorare in autonomia sulla struttura del progetto. Per un'applicazione che deve produrre più file (il codice Python, i template HTML, i fogli di stile, gli script di avvio), questa autonomia è un vantaggio significativo rispetto a Chat.

In pratica il lavoro si è svolto con la sequenza descritta nel capitolo precedente: la fase di ragionamento e le scelte architetturali sono avvenute in Chat, poi il progetto è passato a Cowork per la costruzione vera e propria.

Costruzione guidata

Questa sezione segue la cronologia reale del progetto. Non è una cronologia idealizzata, include i problemi incontrati, le soluzioni trovate e le decisioni prese strada facendo, perché è da queste situazioni che si impara il metodo.

Il prototipo

Il primo risultato è arrivato in pochi minuti. Cowork ha generato autonomamente la struttura del progetto, creando il file Python con il server Flask, la pagina HTML con l'area di trascinamento per i PDF, il meccanismo di conversione, la gestione dei file temporanei. Il prototipo si apriva nel browser, accettava un PDF e produceva un file markdown. Insomma funzionava.

La velocità con cui si ottiene un primo risultato funzionante è una delle caratteristiche più sorprendenti del vibe coding. E anche una delle più pericolose, perché genera l'illusione che il progetto sia quasi finito. In realtà il prototipo è solo il punto di partenza. I problemi emergono dall'uso, non dalla costruzione.

I primi problemi

I problemi sono arrivati immediatamente, non dal codice ma dall'uso dell'applicazione.

Le immagini inutilizzabili. La prima versione convertiva le immagini del PDF in stringhe base64, lunghi blocchi di caratteri incomprensibili inseriti direttamente nel markdown. Aperto con un editor di testo base, il risultato è un muro di caratteri senza senso che rende il documento illeggibile. L'opzione è stata rimossa, sostituita con la possibilità di salvare le immagini come file separati.

I percorsi assoluti. Quando si sceglieva di estrarre le immagini come file separati, il markdown conteneva riferimenti a cartelle temporanee del server. Percorsi come `/tmp/uploads/12345/immagine.png` che non

esistono più una volta chiusa l'applicazione. La soluzione è stata creare un archivio ZIP contenente il file markdown e una cartella con le immagini, tutti con percorsi relativi corretti.

I launcher mancanti. Per pubblicare l'applicazione su GitHub servivano script di avvio per Windows e per Mac/Linux, in modo che l'utente potesse lanciare l'applicazione con un doppio clic invece di dover aprire un terminale. Un dettaglio che sembra ovvio a posteriori ma che non era nella specifica iniziale.

Nessuno di questi problemi è stato trovato leggendo il codice, sono stati tutti trovati **usando l'applicazione**, provandola come la userebbe chiunque altro. Questo è il punto chiave del vibe coding per chi non programma, si può anche non capire come funziona il codice, ma è necessario capire se il risultato funziona. La capacità di testare il prodotto è la competenza fondamentale.

La correzione ha seguito ogni volta lo stesso schema. Si descrive il problema a Claude, possibilmente con uno screenshot o un esempio concreto, e si lascia che trovi la soluzione. Non è necessario suggerire come risolvere il problema, ma occorre saperlo descrivere con precisione.

Il design

Una volta risolti i problemi funzionali, l'applicazione faceva quello che doveva fare ma l'aspetto era generico e poco curato. A questo punto è iniziata la fase di lavoro sull'estetica, che ha introdotto una dinamica diversa.

La richiesta a Claude è stata accompagnata da un'indicazione precisa, cioè proporre delle alternative e aspettare prima di applicarne una. La formulazione è stata "raccontale e aspetta prima di applicarle". L'intenzione era valutare le opzioni prima che Claude ne implementasse una, evitando il rischio di dover disfare un lavoro già fatto.

Claude ha descritto cinque stili diversi, ciascuno con una personalità visiva distinta. Due erano interessanti: uno stile editoriale monocromatico e una variante più netta dello stesso approccio. Anziché scegliere sulla base delle descrizioni ho chiesto di preparare i mockup di entrambi per un confronto visivo.

La scelta finale è caduta sullo stile più deciso: angoli netti, tipografia bold, un accento di rosso come unico colore. Un'estetica professionale che comunicava serietà senza bisogno di decorazioni.

Questa fase illustra un principio importante. Nel vibe coding il controllo non si esercita solo sul "cosa" ma anche sul "come" e sul "quando". Chiedere a Claude di proporre senza applicare, di mostrare mockup prima di implementare, di aspettare il giudizio prima di procedere sono tutte forme di controllo sul processo che non richiedono competenze tecniche.

Il problema del trasferimento

L'implementazione del nuovo design ha provocato uno dei problemi più ostinati della sessione, e anche uno dei più istruttivi.

Il trasferimento del file HTML dalla sandbox di Cowork al PC Windows ha corrotto il CSS, rendendo l'interfaccia completamente rotta, testo grezzo senza alcuno stile, pulsanti deformati, layout inesistente. Il

tipo di problema che, a prima vista, sembra catastrofico.

La soluzione è stata mandare uno screenshot a Claude per mostrare esattamente cosa si vedeva. Descrivere il problema a parole sarebbe stato meno efficace, perché il problema era visivo e la sua natura non era chiara. Claude ha riscritto il file compattando il codice per ridurre i problemi nel trasferimento.

La lezione è doppia. Da un lato, quando qualcosa si rompe in modo incomprensibile, mostrare il problema è spesso più efficace che descriverlo. Dall'altro, le difficoltà più ostinate in un progetto di vibe coding spesso non riguardano la logica dell'applicazione ma "dettagli" come trasferimenti di file, configurazione dell'ambiente, compatibilità tra sistemi. Sono problemi che si risolvono con pazienza e iterazione, anche senza competenze di programmazione.

Le feature non previste

A questo punto l'applicazione funzionava e aveva un aspetto professionale. Ma durante l'uso sono emerse esigenze che non facevano parte della specifica iniziale.

Un esempio, il progetto era destinato a GitHub e doveva essere bilingue, inglese come lingua predefinita e italiano come alternativa, con un selettore di lingua per passare dall'una all'altra. Claude in genere mette questo selettore nell'angolo in alto a destra, ma su schermi grandi risulta separato da troppo spazio bianco dall'area di lavoro. La soluzione è stata chiedere di spostare il selettore, la posizione l'ho indicata passandogli uno screenshot con una croce nel punto in cui volevo fosse messo, sotto il pulsante di conversione, una posizione che funzionava bene sia visivamente sia tecnicamente.

La necessità di questa fase conferma che un progetto non è mai completamente definito all'inizio. Nuove esigenze emergono durante la costruzione, dall'uso, dal contesto, dalla consapevolezza crescente di ciò che l'applicazione potrebbe fare. Il metodo spec.md + todo.md descritto nel capitolo precedente è un punto di partenza, non un contratto rigido. La specifica evolve con il progetto.

Il ciclo fondamentale

Nello sviluppo di un'app si ritrova un pattern, indipendentemente dal fatto che il problema sia funzionale, estetico o di contesto.

Descrivere. Si spiega a Claude cosa si vuole ottenere, o cosa non funziona nel risultato attuale. La descrizione può essere un testo, uno screenshot, un messaggio di errore copiato e incollato. Più è precisa, migliore sarà il risultato.

Generare. Claude produce la soluzione. Potrebbe essere il codice iniziale di una nuova funzionalità, la correzione di un problema, la riscrittura di un componente. Non è necessario leggere o capire ciò che Claude produce.

Testare. Si prova il risultato. L'applicazione si apre nel browser? Il PDF viene convertito? Il file markdown è leggibile? Le immagini sono gestite correttamente? Il design corrisponde a quanto richiesto? Ogni domanda ha una risposta immediata e verificabile senza competenze tecniche.

Correggere. Se qualcosa non va, si torna al primo passaggio. Si descrive il problema e il ciclo ricomincia.

La differenza tra le fasi non è nel pattern ma nel suo sviluppo, si passa dal "funziona o non funziona" al "funziona ma non ancora come volevo".

Quando il ciclo si blocca

A volte il ciclo descrivere-generare-testare-correggere non converge. Si descrive un problema, Claude propone una soluzione, la soluzione introduce un nuovo problema, si corregge, e il problema originale riappare.

Riconoscere un loop è il primo passo per uscirne. Se dopo tre o quattro iterazioni sullo stesso problema la situazione non migliora, continuare nella stessa direzione raramente porta a un risultato. Le strategie che funzionano sono di due tipi.

La prima è **cambiare il contesto**. Aprire una nuova conversazione e descrivere il problema da zero, senza lo storico dei tentativi falliti. A volte Claude resta intrappolato in un approccio che non funziona e ripartire con una descrizione pulita gli permette di trovare una strada diversa.

La seconda è **ridurre l'ambizione**. Se l'obiettivo è troppo complesso per essere raggiunto in un singolo passaggio, scomporlo in parti più piccole. Invece di chiedere "fai funzionare il selettore lingua con il layout responsive e l'animazione", chiedere prima solo "fai funzionare il selettore lingua", poi il layout, poi l'animazione. Ogni pezzo verificato singolarmente prima di combinarli.

Quando fermarsi e ripensare

C'è un momento, in ogni progetto, in cui la domanda giusta non è "come risolvo questo problema" ma "questo problema vale la pena di essere risolto". Una funzionalità che richiede dieci iterazioni per funzionare potrebbe non essere essenziale. Un aspetto del design che non si riesce a ottenere come si vorrebbe potrebbe essere accettabile in una forma leggermente diversa.

Questa capacità di giudizio, sapere quando un risultato è abbastanza buono, è una competenza dell'autore che nessuna AI può sostituire.

Cosa abbiamo imparato

Il progetto è passato da un'idea a un'applicazione funzionante con interfaccia grafica, supporto bilingue e un'estetica professionale in una singola sessione di lavoro. Ecco i punti che si ritroveranno in ogni nuovo progetto.

- Le decisioni architetturali sono dell'autore.
- Il vibe coding non elimina il testing, lo rende ancora più importante.
- Un progetto non è mai completamente definito all'inizio.

- Le difficoltà più ostinate riguardano l'infrastruttura, non la logica.

Il progetto descritto in questo capitolo è disponibile come repository pubblico su GitHub (https://github.com/paolodalprato/pymupdf4llm-gui) e può essere installato e utilizzato da chiunque seguendo le istruzioni nel README. Il capitolo successivo affronta progetti più complessi, con meno guida passo-passo e più autonomia nella conduzione del lavoro.

Progetti più complessi

Il primo progetto ha mostrato il ciclo fondamentale (descrivere -> generare -> testare -> correggere), il risultato può essere un'applicazione singola, un framework, un'interfaccia. Funziona, è utile, risolve un problema concreto.

Questo capitolo racconta due progetti di natura diversa. Il primo è un sistema di documentazione fatto di componenti che devono funzionare insieme (configurazione + script + pipeline di automazione + fogli di stile), il secondo è un'arena di discussione tra agenti AI (architettura client-server + chiamate API a servizi esterni + orchestrazione di processi indipendenti).

Dopo il capitolo precedente do per buono che il metodo sia stato acquisito, qui non c'è una guida passo dopo passo ma il racconto si concentra sulle decisioni, sui problemi che emergono quando la complessità cresce, e su ciò che si impara affrontandoli.

Un sistema di documentazione

Il sito che ospita questo manuale è esso stesso un progetto di vibe coding. Non è nato come esercizio ma da un'esigenza professionale precisa e la sua costruzione illustra un tipo di complessità diverso da quello di un'applicazione singola.

Il problema

Nella mia attività di formatore e divulgatore sulle AI generative, condivido regolarmente tutorial e manuali. Per mesi il formato è stato il PDF, pratico da distribuire, familiare a tutti, facile da creare. Ma i PDF hanno un limite strutturale, sono statici, aggiornare un manuale significa rigenerare l'intero file, redistribuirlo, sperare che chi lo aveva scaricato trovi la versione nuova. In un campo che cambia ogni settimana, questo diventa un problema serio.

L'idea era costruire un posto in cui i manuali fossero sempre aggiornati, navigabili, ricercabili, e dove chi arriva potesse vedere subito cosa c'è di disponibile. Non un blog con articoli sparsi, ma una piattaforma di documentazione strutturata. Con l'obiettivo strategico di posizionarmi come riferimento su argomenti specifici, in particolare l'ecosistema di Claude e NotebookLM. E con un'ambizione ulteriore, raggiungere un pubblico internazionale pubblicando ogni manuale sia in italiano sia in inglese.

Le scelte architetturali

La discussione iniziale è avvenuta in Chat. La descrizione del problema e degli obiettivi ha portato Claude a proporre MkDocs con il tema Material for MkDocs, ospitato su GitHub Pages. La scelta si basava su criteri concreti: - i contenuti si scrivono in markdown, un formato leggibile e modificabile con qualsiasi editor di

testo - il deploy è automatico, si pubblica con un commit su GitHub - il tema Material offre navigazione, ricerca, modalità chiara e scura, tutto già pronto - infine il costo è zero, sia per lo strumento sia per l'hosting.

Nessuna di queste scelte è stata proposta da me, non avendo la competenza tecnica per valutare le alternative tra generatori di siti statici, confrontare i sistemi di hosting o scegliere un tema. La divisione del lavoro è stata netta, ho raccontato cosa volevo, Claude ha proposto un ventaglio di ipotesi tecniche, spiegandomi per ognuna le caratteristiche e le conseguenze nell'adottarla. In base a queste spiegazioni ho scelto tra le opzioni proposte.

Questa dinamica è diversa da quella del primo progetto. Con PyMuPDF4LLM GUI la scelta architetturale è stato un singolo passo, con il sistema di documentazione c'è stata una sequenza di scelte seguendo una sorta di struttura ad albero, con ogni nodo che influenzava le proposte del livello successivo.

La costruzione progressiva

A differenza di un'applicazione che si costruisce e poi si usa, il sistema di documentazione è cresciuto per strati successivi. Il primo strato è stato un sito minimale con un solo manuale, una configurazione di base e il deploy su GitHub Pages. Funzionava, era online, si poteva leggere. Il primo manuale pubblicato riguardava proprio MkDocs e GitHub Pages, documentando il processo di costruzione mentre avveniva.

Il secondo strato ha aggiunto la struttura per ospitare più manuali. Il sito è un monorepo, un unico repository che contiene tutte le guide come sezioni separate. La navigazione usa tab tematici nella barra superiore che raggruppano i manuali per categoria, con la sidebar che mostra l'indice del manuale selezionato. Questa organizzazione è stata progettata per scalare: aggiungere un nuovo manuale significa creare una cartella, aggiungere le pagine e aggiornare la configurazione, senza toccare nulla di ciò che già esiste.

Il terzo strato è stato il più complesso: la generazione automatica dei PDF. L'idea era che chi preferisce il formato tradizionale possa scaricare l'intero manuale come PDF, con copertina, indice generato automaticamente, numerazione delle pagine e footer con la licenza. Ma a differenza di un'esportazione banale, il sistema doveva generare un PDF che fosse un prodotto editoriale curato, non un dump del sito web.

Il problema più ostico

La pipeline PDF è stata la parte che ha richiesto più iterazioni. Il sistema finale coinvolge quattro componenti separati che devono lavorare insieme, un plugin di MkDocs che genera un PDF per ogni singola pagina, uno script Python che genera automaticamente un indice analizzando la struttura dei capitoli, un secondo script che assembla i PDF individuali e un workflow di GitHub Actions che orchestra l'intero processo a ogni pubblicazione.

Ciascuno di questi componenti ha funzionato relativamente presto in isolamento. I problemi sono emersi dall'interazione unendo i pezzi, l'ordine di assemblaggio dei PDF non corrispondeva all'ordine dei capitoli nella navigazione, l'indice generato includeva la copertina tra i capitoli, il foglio di stile per la stampa

interferiva con gli elementi del sito web. Ogni correzione a un componente rischiava di rompere qualcosa in un altro.

La lezione è che in un sistema fatto di componenti interdipendenti, il testing deve diventare più esigente. Non basta verificare che ogni pezzo funzioni, bisogna verificare che funzionino insieme. E il ciclo descrivere-generare-testare-correggere si applica non ai singoli componenti ma al sistema nel suo insieme: si pubblica, si scarica il PDF, si controlla il risultato, si corregge.

I risultati

Il sistema di documentazione è online da metà dicembre 2025. In quattro mesi i manuali hanno totalizzato quasi 900 download PDF, con la guida su NotebookLM che da sola ne rappresenta quasi il 90%, distribuiti tra la versione italiana e quella inglese. Il sito si è posizionato in prima pagina su Google per ricerche come "Claude NotebookLM" o "NotebookLM MCP".

Ma il risultato più significativo non è nei numeri. Il sistema di documentazione ha generato altri progetti. La necessità di pubblicare regolarmente sul blog ha portato alla creazione di una skill dedicata per la scrittura e l'ottimizzazione degli articoli. La necessità di misurare i risultati ha portato all'integrazione di un sistema di analytics con tracciamento personalizzato dei download. Ogni componente aggiunto ha esteso le capacità della piattaforma, che a sua volta ha suggerito nuove possibilità.

Questo è un pattern che si incontra quando un progetto di vibe coding smette di essere un'applicazione isolata e diventa un sistema: ogni pezzo che si aggiunge apre la porta ad altri pezzi. Il sistema cresce organicamente, guidato dalle esigenze reali che emergono dall'uso.

Un'arena di discussione tra agenti

Il secondo progetto nasce da un'intuizione diversa. Non dalla necessità di risolvere un problema pratico immediato, ma dall'osservazione di un'assenza, tra le applicazioni basate sull'intelligenza artificiale, nessuna sembrava mettere al centro l'architettura agentica come strumento utilizzabile da tutti.

L'idea

L'obiettivo era costruire qualcosa di pratico, facile da usare e agnostico rispetto al fornitore di AI. Il target era ampio ma con tratti comuni: professionisti autonomi, piccoli studi, piccole e medie imprese. Persone che non sono esperti tecnici ma che hanno una predisposizione ad approfondire gli aspetti pratici dell'AI, ad esempio imparare a usare le API. Persone che avrebbero bisogno di un team di esperti per analizzare un problema da prospettive diverse, ma che non possono permetterselo per motivi di budget o di tempistiche. E persone che trattano informazioni riservate, avvocati con i fascicoli dei clienti, consulenti con i dati finanziari, medici con le cartelle cliniche, e che quindi hanno bisogno di una soluzione che funzioni interamente in locale, senza che i dati escano dal proprio computer.

L'idea concreta era un'arena di discussione strutturata in cui partecipanti virtuali, ciascuno con un ruolo, una prospettiva e uno stile comunicativo definiti, discutono uno scenario fornito dall'utente. Un moderatore apre

la discussione, i partecipanti argomentano per più round rispondendosi a vicenda, e alla fine il moderatore emette un verdetto motivato. Questa situazione, il panel di esperti, trova applicazioni in campi molto diversi, ad esempio analisi legale, revisione clinica, pianificazione strategica, valutazione del rischio, dibattito accademico.

La genesi

Questo progetto illustra una dinamica che il capitolo precedente ha solo accennato, il vibe coding che inizia prima di sapere che si sta facendo vibe coding. Ne avevo già parlato in questo articolo (https://www.ai-know.pro/vibe-coding-la-parola-importante-non-e-coding/).

Nella mia esperienza, il progetto è nato come conversazione in Chat. Ho descritto l'idea, il tipo di applicazione che immaginavo, il pubblico a cui era destinata. Claude ha iniziato a proporre soluzioni, a esplorare le implicazioni architetturali, a suggerire funzionalità. La conversazione è passata gradualmente dal "ne stiamo discutendo" al "lo stiamo progettando" senza un momento preciso in cui la transizione è avvenuta. A un certo punto il progetto esisteva come specifica, e il passaggio alla costruzione, in Cowork, è stato naturale.

Questo percorso è probabilmente quello più comune per chi usa l'AI quotidianamente. Non si parte da una decisione formale ("oggi costruisco un'applicazione"), si parte da un problema o da un'idea e la conversazione con l'AI porta naturalmente verso una soluzione costruibile.

L'architettura

L'applicazione finale ha un'architettura che riflette i requisiti iniziali, è composta da un file HTML che contiene tutta l'interfaccia, la logica e lo stile, e un server proxy in Python che fa da intermediario tra il browser e i servizi AI. Il server proxy usa solo la libreria standard di Python, nessuna dipendenza esterna da installare.

Queste scelte, tutte proposte da Claude, rispondono ai vincoli del progetto. L'assenza di dipendenze rende l'installazione banale, si scarica il progetto, si lancia lo script Python, si apre il browser. Niente `npm install`, niente Docker, niente configurazione complessa. Il supporto multi-provider, Anthropic, OpenAI, Google Gemini, DeepSeek e qualsiasi servizio compatibile con le API di OpenAI, garantisce l'agnosticismo rispetto al fornitore. E il supporto per modelli locali tramite Ollama, LM Studio o altro framework risolve il requisito della privacy, quando si usa un modello locale nessun dato esce dal computer.

Il concetto architetturale più interessante è quello alla base del funzionamento, ogni partecipante alla discussione è una chiamata API indipendente. Ogni agente riceve il proprio ruolo, la trascrizione cumulativa della discussione e le istruzioni per il turno corrente, ma non ha accesso al processo generativo degli altri partecipanti. Non "sa" cosa hanno pensato gli altri, vede solo cosa hanno detto. Questa separazione produce voci autenticamente distinte, perché ogni agente parte da un contesto pulito e costruisce la propria argomentazione in modo indipendente.

La complessità nuova

Rispetto al primo progetto, ADA introduce un tipo di complessità che non si può risolvere semplicemente testando l'interfaccia. L'architettura client-server, con il browser che comunica con il proxy e il proxy che comunica con il servizio AI, aggiunge punti di possibile rottura invisibili all'utente. Un errore di autenticazione con il provider, un timeout su un modello locale lento, un problema di certificati SSL su Windows sono tutti problemi che non si manifestano come "il pulsante non funziona" ma come messaggi di errore tecnici che richiedono interpretazione.

Qui il ruolo dell'autore cambia. Nel primo progetto il testing era prevalentemente funzionale. Il PDF si converte? Il markdown è leggibile? Le immagini sono gestite correttamente? In un progetto con architettura più articolata, una parte del testing diventa diagnostica. Bisogna saper leggere un messaggio di errore, anche senza capirne i dettagli tecnici, e riportarlo a Claude in modo che possa proporre una soluzione. La capacità di comunicare un problema con precisione diventa ancora più importante della capacità di risolverlo.

Un'altra dimensione di complessità riguarda il consumo di risorse. Ogni turno di discussione è una chiamata API che include il documento allegato, la trascrizione cumulativa e le istruzioni. Con un documento di 100 kilobyte, due partecipanti e due round di discussione, il sistema effettua otto chiamate API ciascuna delle quali trasporta l'intero documento. Per i provider cloud questo si traduce in costi, per i modelli locali in tempi di elaborazione. Comprendere questa dinamica non richiede competenze tecniche, ma richiede la capacità di ragionare su come l'architettura influenza l'uso pratico.

Il collegamento con il primo progetto

C'è un collegamento diretto tra questo progetto e quello del capitolo precedente che vale la pena raccontare. PyMuPDF4LLM GUI, il convertitore di PDF in markdown, è nato dall'osservazione di una soluzione implementata dentro ADA. Per gestire il consumo di token dei documenti allegati, Claude aveva proposto di integrare la libreria PyMuPDF4LLM per convertire i PDF in markdown prima di passarli al modello, riducendo drasticamente il peso del testo. Osservando quella soluzione al lavoro, l'intuizione è stata che la stessa conversione poteva essere utile a chiunque lavori con i PDF nelle conversazioni con i modelli linguistici, non solo dentro ADA ma in qualsiasi contesto. Il pensiero successivo è che per rendere facilmente usabile la libreria, progettata per l'uso da terminale o tramite script, serviva un'interfaccia grafica. Il progetto del capitolo precedente è nato così.

Questo percorso, un progetto che genera un'idea per un altro progetto, è una delle dinamiche più produttive del vibe coding. Non avviene per pianificazione ma per esposizione, lavorare a un problema mette in contatto con soluzioni che si applicano anche altrove.

I risultati

Il progetto è disponibile come repository pubblico su GitHub (https://github.com/paolodalprato/agent-discussion-arena) e può essere installato e utilizzato da chiunque seguendo le istruzioni nel README. L'architettura si è rivelata sufficientemente solida da essere adattata anche come skill, dove la stessa logica di discussione strutturata con agenti indipendenti funziona senza bisogno di un proxy esterno, perché Claude stesso diventa il modello che genera gli interventi dei partecipanti.

Cosa cambia quando la complessità cresce

I due progetti raccontati in questo capitolo, insieme a quello del capitolo precedente, permettono di osservare come cambia il lavoro quando la complessità aumenta.

La prima differenza riguarda la **costruzione progressiva**. Un'applicazione semplice si può costruire in una sessione e ottenere un risultato funzionante. Un sistema come la piattaforma di documentazione cresce per strati, ciascuno dei quali aggiunge una capacità e introduce nuove interazioni da gestire. Questo richiede un approccio diverso alla pianificazione: non si progetta tutto all'inizio, si progetta il primo strato funzionante e poi si aggiungono componenti sulla base delle esigenze reali che emergono dall'uso.

La seconda differenza riguarda il **ruolo dell'autore**. Nel primo progetto la competenza principale è il testing funzionale, verificare che il risultato faccia ciò che deve fare. Nei progetti più ambiziosi si aggiungono competenze diverse: la capacità di prendere decisioni architetturali scegliendo tra opzioni proposte da Claude, la capacità di diagnosticare problemi che non si manifestano in modo evidente, la capacità di ragionare sulle implicazioni di lungo termine delle scelte fatte. Nessuna di queste richiede competenze di programmazione, ma tutte richiedono la capacità di pensare in modo sistematico.

La terza differenza è che **un progetto genera altri progetti**. Il sistema di documentazione ha generato la skill per il blog, il sistema di analytics, e una serie di strumenti complementari. ADA ha generato l'idea per PyMuPDF4LLM GUI e successivamente la skill che porta la stessa architettura dentro Claude Desktop. Questo è un effetto che non si può pianificare ma che si verifica regolarmente: lavorare a un problema espone a soluzioni che si applicano anche altrove, e ogni progetto completato diventa un fondamento su cui costruire il successivo.

Infine, c'è una differenza che riguarda la **gestione delle sessioni di lavoro**. Un progetto semplice si esaurisce in una conversazione. Un progetto complesso richiede più sessioni, e questo introduce il problema della continuità: Claude non ricorda le conversazioni precedenti a meno che non gli si fornisca il contesto. Strumenti come il file CLAUDE.md, la specifica del progetto e il file di stato diventano indispensabili non come burocrazia ma come memoria del progetto, il ponte tra una sessione e la successiva.

Lavorare bene con Claude

Il vibe coding non è solo chiedere e ricevere, è una collaborazione che funziona meglio quando si conoscono le regole del gioco. Questo capitolo raccoglie le pratiche operative che fanno la differenza tra un'esperienza frustrante e un flusso di lavoro produttivo.

Gestire gli errori

Gli errori non sono incidenti di percorso, sono il percorso. Ogni progetto descritto nei capitoli precedenti ha attraversato decine di errori prima di funzionare. La differenza tra un principiante bloccato e un principiante produttivo sta nel modo in cui affronta questi errori.

La strategia più semplice è anche la più efficace. Quando qualcosa non funziona, basta copiare il messaggio di errore e incollarlo nella conversazione con Claude, senza cercare di interpretarlo. Claude è progettato per leggere messaggi di errore e nella maggior parte dei casi sa cosa fare. Non serve capire cosa significa "TypeError: Cannot read properties of undefined", basta incollarlo.

Quando l'errore da solo non basta, conviene aggiungere il contesto di ciò che si stava facendo. "Ho cliccato il pulsante Converti e appare questo errore" è molto più utile di "non funziona". La combinazione di errore tecnico e descrizione in linguaggio naturale è spesso tutto ciò che serve per individuare il problema.

Il momento più insidioso arriva quando Claude propone una correzione, la correzione non risolve il problema, e Claude propone la stessa correzione riformulata in modo diverso. Questo è un *loop*, e riconoscerlo è una competenza che vale la pena sviluppare. I segnali sono riconoscibili: la stessa porzione di codice viene modificata per la terza volta, l'errore cambia forma ma non scompare, le spiegazioni di Claude diventano più lunghe ma meno concrete.

Uscire da un loop richiede un cambio di strategia, non più tentativi nella stessa direzione. Alcune possibilità concrete:

- *Riformulare il problema*. Invece di continuare a descrivere l'errore, descrivere il risultato atteso. "Quando clicco Converti dovrebbe apparire il file markdown nella colonna di destra" sposta l'attenzione dalla diagnosi all'obiettivo.
- *Chiedere un approccio diverso*. "Questa strada non funziona, quale alternativa esiste?" autorizza Claude ad abbandonare la soluzione corrente senza doverla difendere.
- *Fare un passo indietro*. A volte il problema è a monte, un'installazione incompleta, una dipendenza mancante, un file nella posizione sbagliata. Chiedere di verificare i prerequisiti anziché insistere sulla correzione può sbloccare situazioni che sembravano irrisolvibili.
- *Aprire una nuova conversazione*. Se la sessione è lunga e il contesto si è accumulato, può essere più efficace iniziare da capo con una descrizione pulita del problema. Claude non porta con sé i pregiudizi

della conversazione precedente.

Mantenere il contesto

Claude ragiona all'interno di una *finestra di contesto*, lo spazio che contiene la conversazione dall'inizio fino all'ultimo messaggio. Tutto ciò che sta dentro la finestra è visibile, tutto ciò che ne esce è perduto. La finestra è ampia ma non è infinita. In una sessione di lavoro intensa, con scambi di codice, screenshot e correzioni ripetute, la si può saturare.

Quando la finestra si avvicina al limite il comportamento di Claude cambia in modo sottile. Le risposte diventano meno precise, i dettagli concordati all'inizio della sessione vengono dimenticati, errori già risolti ricompaiono. Non è un malfunzionamento, è un limite fisico dell'architettura.

La strategia più efficace è prevenire il problema con *sessioni brevi e focalizzate*. Invece di una conversazione-maratona in cui si progetta, costruisce, testa e corregge tutto nello stesso flusso, conviene separare le attività. Una sessione per impostare la struttura del progetto, un'altra per costruire la prima funzionalità, un'altra ancora per risolvere quel bug persistente. Ogni sessione parte con un contesto pulito e un obiettivo preciso.

I *Progetti* in Chat offrono un livello di continuità che le singole conversazioni non hanno. Un Progetto raccoglie istruzioni persistenti e file di riferimento che restano disponibili in ogni nuova conversazione aperta al suo interno. Le istruzioni di progetto non consumano la finestra di contesto allo stesso modo dei messaggi della conversazione, ma forniscono a Claude il quadro d'insieme ogni volta che si riparte. Per un progetto di vibe coding di qualsiasi durata, lavorare all'interno di un Progetto è una scelta quasi obbligata.

Il file *CLAUDE.md*, già descritto nel capitolo 4, svolge un ruolo complementare. Le istruzioni di progetto in Chat definiscono il contesto generale, mentre CLAUDE.md (usato in Cowork e Code) registra le decisioni tecniche accumulate durante la costruzione. Convenzioni adottate, errori da non ripetere, scelte architetturali: tutto ciò che Claude dovrebbe sapere quando apre una nuova sessione sulla stessa base di codice. Senza questo tipo di memoria esterna ogni nuova sessione riparte da zero, con il rischio di reintrodurre problemi già risolti.

Ottenere risposte migliori

Non tutte le richieste hanno bisogno della stessa profondità di ragionamento. Chiedere a Claude di rinominare una variabile è diverso dal chiedergli di progettare l'architettura di un'applicazione. Claude Desktop offre due strumenti per calibrare la qualità delle risposte, entrambi accessibili direttamente dall'interfaccia di Chat.

Il primo è il *ragionamento esteso*. Si attiva dal selettore del modello in basso a destra, dove accanto al nome del modello compare un toggle "Ragionamento esteso" con la descrizione "Ragiona più a lungo per attività complesse". Quando è attivo, l'indicazione "Estesa" appare accanto al nome del modello. In questa modalità Claude dedica più tempo all'analisi prima di rispondere, il che produce risultati migliori su problemi

articolati, quelli dove servono più passaggi di ragionamento o dove la soluzione non è immediata. Per richieste semplici il ragionamento esteso non aggiunge valore e consuma più risorse dell'abbonamento.

Il secondo strumento sono gli *stili di risposta*. Si trovano nel menu "+" a sinistra del campo di input, alla voce "Usa stile". Gli stili predefiniti includono Normale, Apprendimento, Conciso, Esplicativo e Formale, ed è possibile crearne di personalizzati con "Crea e modifica stili". Lo stile non cambia ciò che Claude sa, cambia come lo comunica. "Esplicativo" produce risposte più dettagliate e didattiche, "Conciso" va dritto al punto, "Formale" adotta un registro più strutturato. Nel vibe coding lo stile Esplicativo è utile quando si vuole capire *perché* Claude ha scelto una certa soluzione, non solo ricevere il codice.

La combinazione dei due strumenti è più efficace di ciascuno separatamente. Il ragionamento esteso con stile Esplicativo è la configurazione adatta quando si affronta un problema che non si riesce a risolvere e si vuole che Claude mostri il proprio ragionamento nel dettaglio. Per il lavoro di routine conviene tornare alla configurazione standard, sia per la velocità sia per il consumo di risorse.

Al di là degli strumenti dell'interfaccia, il fattore che incide di più sulla qualità delle risposte è il modo in cui si formula la richiesta. Il principio è la specificità: "il pulsante non funziona" produce risultati peggiori di "il pulsante Converti nella barra superiore non reagisce al clic, il cursore cambia forma ma non succede nulla". Uno screenshot dell'errore, il nome del file, il comportamento atteso e quello effettivo sono informazioni che permettono a Claude di lavorare sulla situazione reale anziché indovinare.

Quando Claude non fa quello che dice d'avere fatto

Questo è probabilmente l'aspetto più insidioso per chi non ha esperienza di programmazione. Claude può generare codice che sembra corretto, usa la sintassi giusta, ha una struttura ragionevole, ma non fa ciò che dovrebbe. Peggio ancora, Claude può affermare che un problema è risolto quando non lo è.

Il fenomeno si manifesta soprattutto dopo ripetuti tentativi falliti. Quando Claude ha provato tre o quattro soluzioni per lo stesso errore senza successo, le risposte successive tendono a diventare meno affidabili. Il codice generato può aggirare il problema anziché risolverlo, ad esempio sopprimendo un messaggio di errore senza eliminarne la causa, oppure restituendo un valore fisso dove dovrebbe esserci un calcolo. In questi casi Claude presenta la modifica con sicurezza, e senza test manuali la si potrebbe accettare come valida.

I segnali di allarme sono pochi ma riconoscibili. Risposte che dichiarano "ora funziona" senza spiegare cosa è cambiato meritano attenzione. Codice che diventa progressivamente più lungo e contorto a ogni iterazione suggerisce che Claude sta accumulando correzioni su correzioni senza una visione chiara del problema. Spiegazioni che diventano vaghe dove prima erano precise indicano che il modello si sta allontanando dal territorio in cui ha dati solidi.

La difesa più efficace è il *test manuale sistematico*. Ogni volta che Claude dichiara di aver risolto un problema, verificare che il comportamento sia effettivamente cambiato. Non basta che il codice non produca errori, deve produrre il risultato atteso. Questa verifica non richiede competenze tecniche, richiede solo la disciplina di provare ogni modifica prima di procedere alla successiva.

Quando il dubbio è forte, una strategia utile è chiedere a Claude di spiegare passo per passo cosa fa il codice che ha appena scritto. Se la spiegazione non corrisponde a ciò che si osserva, il problema è confermato. A quel punto conviene applicare le strategie descritte nella sezione sugli errori: riformulare il problema, cambiare approccio, o aprire una nuova conversazione.

Salvare il lavoro

Costruire un'applicazione funzionante e perderla per un errore è un'esperienza che capita una volta sola, perché dopo la prima volta si impara a salvare. Il *version control* (controllo di versione) è la pratica di registrare lo stato del progetto a intervalli regolari, in modo da poter tornare a una versione precedente se qualcosa va storto. Lo strumento standard si chiama Git.

Git può sembrare intimidatorio ma l'uso di base richiede tre concetti e tre comandi. I concetti sono il *repository* (la cartella del progetto tracciata da Git), il *commit* (un'istantanea del progetto in un dato momento) e lo *staging* (la selezione dei file da includere nella prossima istantanea). I comandi corrispondenti sono `git add` per selezionare i file, `git commit` per registrare l'istantanea e `git log` per consultare la storia.

Il repository locale vive sulla propria macchina, ma la vera sicurezza si ottiene quando una copia esiste anche altrove. *GitHub* è la piattaforma più usata per questo scopo, un servizio gratuito che ospita repository Git accessibili da qualsiasi dispositivo. Caricare il progetto su GitHub richiede un passaggio aggiuntivo dopo il commit, il comando `git push`, e una configurazione iniziale che si fa una sola volta. Il capitolo 9 descrive il procedimento completo.

In Cowork e Code il rapporto con Git è diverso. Code ha Git integrato nativamente e può eseguire commit, creare branch e gestire il repository in autonomia. Cowork non ha accesso diretto a Git ma può interagire con un repository se è disponibile un server MCP con questa funzionalità. In Chat, Git non è accessibile direttamente ma è possibile chiedere a Claude di generare i comandi da eseguire nel terminale.

Il momento giusto per un commit è ogni volta che il progetto raggiunge uno stato funzionante. L'applicazione si avvia e mostra la schermata principale: commit. Il pulsante di conversione funziona: commit. Il layout è quello definitivo: commit. Ogni commit è un punto di ritorno sicuro, e averne molti è sempre meglio che averne pochi. Il messaggio del commit descrive cosa è cambiato in modo leggibile: "aggiunta conversione PDF in markdown" è utile, "aggiornamento" non lo è.

Per chi preferisce non usare Git, l'alternativa minima è copiare l'intera cartella del progetto prima di ogni sessione di lavoro significativa. Una cartella "progetto-backup-10-aprile" è meno elegante di un commit Git ma svolge la stessa funzione di rete di sicurezza. L'importante è avere un modo per tornare indietro.

Sicurezza e qualità

Il capitolo 4 ha proposto un framework per decidere *se* un progetto è adatto al vibe coding, valutando otto dimensioni di rischio prima di iniziare. Questo capitolo si occupa di ciò che viene dopo: *come* proteggere un progetto una volta che si è deciso di costruirlo.

Il punto di partenza è una constatazione pratica: non tutti i progetti richiedono le stesse attenzioni. Un convertitore di PDF che gira sulla propria macchina e un'applicazione che gestisce pagamenti online hanno profili di rischio radicalmente diversi, e trattarli allo stesso modo produce due effetti ugualmente dannosi. Chi costruisce un piccolo tool personale si spaventa inutilmente di fronte a checklist pensate per applicazioni complesse. Chi costruisce qualcosa di più ambizioso si convince di aver fatto abbastanza dopo qualche controllo superficiale. La prima cosa da fare, prima ancora di parlare di rischi e contromisure, è capire a che livello si colloca il proprio progetto.

Quattro livelli di attenzione

La scala che segue aiuta a collocare ciascun progetto nel livello di attenzione appropriato. Ogni gradino aggiunge responsabilità rispetto al precedente, e l'ultimo rappresenta un confine netto: oltre quel punto, le competenze insegnabili in questo manuale non bastano più.

Lo uso solo io, senza dati sensibili

È il caso più semplice e probabilmente il più frequente per chi inizia. Un'applicazione che gira sulla propria macchina, non gestisce credenziali né dati personali di altri, e non è destinata alla condivisione. Il convertitore PDF→markdown del capitolo 5 è un esempio perfetto: riceve un file, lo trasforma, restituisce il risultato. Se qualcosa non funziona, l'unica persona coinvolta è chi lo ha costruito.

A questo livello i controlli di sicurezza sono sostanzialmente assenti. Se l'applicazione funziona come previsto, funziona. L'unica accortezza riguarda chi usa GitHub come backup personale, tenendo il repository privato. Anche in quel caso è buona pratica controllare che il codice non contenga chiavi API o token di accesso a servizi esterni. Un repository privato può diventare pubblico per un clic sbagliato, un account può essere compromesso, e le credenziali che sembravano al sicuro diventano esposte. È un controllo che richiede pochi minuti prima del primo caricamento, e il capitolo spiega come farlo nella sezione sui controlli concreti.

Lo condivido gratuitamente

Il progetto viene pubblicato su GitHub o condiviso con altre persone, ma resta gratuito e non gestisce dati sensibili. La Agent Discussion Arena del capitolo 6 rientra in questa categoria: chiunque può scaricarla e usarla, ma non raccoglie informazioni personali né richiede credenziali.

Il passaggio dalla macchina personale alla condivisione pubblica cambia il quadro in modo sottile ma significativo. Altre persone useranno l'applicazione fidandosi del fatto che faccia quello che dichiara di fare, e niente di più. Questo comporta alcune responsabilità minime: una documentazione che spieghi cosa fa il progetto e come usarlo, una licenza che chiarisca i termini di utilizzo, e una revisione di sicurezza di base prima della pubblicazione. Quest'ultimo punto merita attenzione perché è facile da trascurare: quando si sviluppa per se stessi, un piccolo difetto è un inconveniente; quando altre persone usano il proprio codice, lo stesso difetto diventa un problema di cui si è responsabili.

Lo vendo o ci costruisco un servizio

Il progetto diventa un prodotto commerciale o la base di un servizio a pagamento. Non gestisce ancora dati sensibili, ma il fatto che qualcuno paghi per usarlo cambia la natura delle responsabilità.

È il punto in cui molti vibe coder non si fermano a riflettere, presi dall'entusiasmo di avere qualcosa che funziona e che potrebbe generare un ritorno economico. Ma vendere un prodotto software o offrire un servizio basato su codice porta con sé obblighi che il codice condiviso gratuitamente non ha. Le responsabilità che entrano in gioco includono gli obblighi fiscali legati all'attività commerciale, la necessità di termini di servizio che definiscano cosa il prodotto fa e cosa no, l'aspettativa di affidabilità da parte di chi paga (un'applicazione gratuita che si blocca è un inconveniente, un servizio a pagamento che si blocca è un inadempimento), la responsabilità verso il cliente in caso di malfunzionamenti, e la conformità alle normative applicabili, che variano in base al settore e al tipo di servizio.

Questo manuale non è il luogo per approfondire ciascuno di questi aspetti, ma è il luogo per dire che esistono. Chi arriva a questo livello ha costruito qualcosa di valore, e il passo successivo non è solo tecnico, è imprenditoriale. Informarsi su questi obblighi prima di iniziare a vendere non è prudenza eccessiva, è il minimo necessario.

Gestisce dati sensibili

Il progetto tratta credenziali, dati personali di terzi, informazioni di pagamento, dati sanitari, o qualsiasi altra informazione la cui esposizione causerebbe un danno concreto a qualcuno. A questo livello non conta se l'applicazione la usa una sola persona o un milione: la presenza di dati sensibili cambia radicalmente il profilo di rischio.

Un esempio chiarisce perché. Un'applicazione personale che si collega al proprio conto bancario per aggregare le spese mensili è usata da una sola persona, non è condivisa, non è in vendita. Ma contiene le credenziali di accesso al conto. Se il codice ha una vulnerabilità che espone quelle credenziali, le conseguenze sono le stesse che avrebbe una falla in un'applicazione pubblica: il danno economico è reale, e il fatto che il progetto sia "solo personale" non lo attenua.

Qui il messaggio è netto: far rivedere il codice a un professionista esperto di quel tipo di applicazione. Non "è consigliabile", non "sarebbe meglio": è necessario. Questo manuale può insegnare a riconoscere i rischi e a prendere alcune precauzioni di base, ma non può sostituire la competenza di chi sa leggere il codice con l'occhio allenato a individuare le vulnerabilità specifiche del dominio. Un'applicazione che gestisce dati

bancari richiede un esperto di sicurezza finanziaria. Un'applicazione che tratta dati sanitari richiede un esperto di conformità sanitaria. La competenza di sicurezza non è generica, è specifica al settore.

Questo non significa rinunciare al vibe coding per questi progetti. Significa che il vibe coding produce il prototipo e la struttura, ma prima di usare l'applicazione con dati reali il codice passa attraverso una revisione professionale. È lo stesso principio dello spettro della delega introdotto nel capitolo 1: delegare la costruzione non significa delegare la responsabilità.

Cosa può andare storto

I dati della ricerca offrono un quadro di riferimento utile. Il 2025 GenAI Code Security Report (https://www.veracode.com/resources/analyst-reports/2025-genai-code-security-report/) di Veracode ha testato oltre cento modelli linguistici su ottanta compiti di programmazione in quattro linguaggi diversi, riscontrando che il 45% del codice generato conteneva vulnerabilità classificate nella OWASP Top 10, la lista delle dieci categorie di rischio più critiche per le applicazioni web. Il dato non è migliorato nelle rilevazioni successive fino a inizio 2026, nonostante i progressi dei modelli in altri ambiti. Separatamente, il State of Secrets Sprawl 2026 (https://www.gitguardian.com/state-of-secrets-sprawl-report-2026) di GitGuardian, analizzando milioni di commit pubblici su GitHub nel 2025, ha mostrato che il codice scritto con l'assistenza di strumenti AI espone credenziali con una frequenza doppia rispetto al codice scritto senza assistenza AI.

Questi numeri riguardano sviluppatori professionisti che usano strumenti AI nel loro lavoro quotidiano. Per chi non programma e delega interamente la scrittura del codice, il rischio non è minore, è semplicemente meno visibile. Uno sviluppatore che riceve codice insicuro ha almeno la possibilità di riconoscerlo. Chi non sa leggere il codice si fida del fatto che funzioni, e "funziona" non significa "è sicuro".

I problemi più frequenti rientrano in tre categorie che vale la pena conoscere, non per saperli risolvere tecnicamente, ma per sapere che esistono e poterli nominare quando si chiede una revisione.

Credenziali nel codice

È il problema più comune e il più facile da prevenire, eppure continua a essere la causa di incidenti seri. Quando si costruisce un'applicazione che si collega a un servizio esterno, quel collegamento richiede quasi sempre una chiave di accesso, che può essere una chiave API, un token, o una combinazione di nome utente e password. Queste credenziali funzionano esattamente come le chiavi di casa: chi le possiede entra.

Il rischio nasce da come queste credenziali finiscono nel codice. Quando si chiede a Claude di costruire un'applicazione che si collega a un servizio, il codice generato spesso include la credenziale direttamente nel testo del programma, come se fosse un dato qualsiasi. Finché il codice resta sulla propria macchina il problema non si pone. Ma nel momento in cui viene caricato su un repository, anche privato, quelle credenziali esistono in un luogo diverso dal proprio computer, e il livello di rischio cambia.

L'analisi di GitGuardian ha documentato oltre 28 milioni di credenziali inserite direttamente nel codice pubblico su GitHub nel solo 2025, con un aumento del 34% rispetto all'anno precedente. Il dato più preoccupante riguarda la persistenza dell'esposizione: quasi il 70% delle credenziali identificate come

valide nel 2022 risultava ancora attivo a distanza di tre anni. Chi espone una chiave API e non se ne accorge lascia una porta aperta per anni.

Una precauzione pratica per chi inizia è scegliere servizi che offrano chiavi API di test o sandbox, distinte da quelle di produzione, e che non abbiano la ricarica automatica abilitata. In questo modo, anche se la chiave finisce accidentalmente nel codice, il danno potenziale è limitato. È un'abitudine semplice da adottare fin dal primo progetto, e che diventa naturale con il tempo.

Dati esposti dove non dovrebbero essere

Questo problema è più sottile del precedente e più difficile da individuare per chi non programma. Riguarda la differenza tra ciò che succede "dietro le quinte" di un'applicazione, la parte che l'utente non vede, e ciò che arriva al browser o all'interfaccia visibile.

Un esempio concreto aiuta a capire. Si immagini di costruire una piccola applicazione web per raccogliere le iscrizioni a un evento. I partecipanti compilano un modulo con nome, email e numero di telefono. L'organizzatore ha una pagina riservata dove vede l'elenco delle iscrizioni. Se il codice è scritto senza attenzione, potrebbe rendere accessibili tutti i dati delle iscrizioni a chiunque conosca l'indirizzo della pagina, anche senza autenticazione. Oppure potrebbe inviare al browser informazioni in eccesso rispetto a quelle visualizzate, dati che l'interfaccia non mostra ma che sono presenti nella pagina e accessibili a chiunque sappia dove guardare.

Il codice generato dalle AI tende a questo tipo di errore perché ottimizza per il risultato visibile. Se l'interfaccia mostra le informazioni giuste, il compito sembra completato. La distinzione tra "mostra correttamente" e "protegge adeguatamente" richiede un'attenzione che il modello non ha se non gliela si chiede esplicitamente.

Input non controllati

Ogni applicazione che riceve informazioni dall'esterno, che si tratti di un campo di testo dove l'utente scrive il proprio nome o di un modulo di ricerca, deve decidere cosa fare con quelle informazioni. Il problema nasce quando l'applicazione accetta qualsiasi cosa le venga inviata senza verificarla.

L'analogia più intuitiva è una cassetta delle lettere che accetta qualsiasi oggetto, non solo le lettere. Se qualcuno inserisce qualcosa di diverso da una lettera, l'effetto dipende da cosa c'è dall'altra parte. Nella migliore delle ipotesi non succede nulla, nella peggiore il meccanismo si inceppa o si danneggia. Nel caso delle applicazioni, "inserire qualcosa di diverso" significa inviare al programma istruzioni mascherate da dati normali, che il programma esegue senza distinguerle dai dati legittimi.

Due delle vulnerabilità più frequenti nel codice generato da AI rientrano in questa categoria. Veracode ha rilevato che l'86% dei campioni di codice testati non proteggeva adeguatamente contro uno di questi attacchi, e l'88% era vulnerabile all'altro. Sono percentuali che riguardano il codice generato da oltre cento modelli linguistici diversi, non un singolo strumento.

Anche qui, l'obiettivo non è diventare esperti di queste vulnerabilità. È sapere che il codice che accetta dati dall'esterno senza controllarli è codice a rischio, e che quando si chiede una revisione di sicurezza questo è uno dei punti da verificare.

I controlli che si possono fare

La sezione precedente ha descritto i problemi. Questa si occupa delle contromisure alla portata di chi non programma, graduate sui livelli di attenzione definiti all'inizio del capitolo. Nessuna di queste azioni sostituisce una revisione professionale per i progetti che la richiedono, ma ciascuna riduce il rischio in modo concreto e misurabile.

La revisione di sicurezza con Claude

Il modo più diretto per verificare la sicurezza di un progetto è chiedere a Claude di analizzarlo, ma con un accorgimento importante: farlo in una sessione separata da quella in cui si è costruito il progetto. Nella sessione di costruzione, Claude ha il contesto di tutte le scelte fatte durante lo sviluppo e tende a confermarne la validità piuttosto che metterle in discussione. Una sessione nuova parte senza quel contesto e guarda il codice con maggiore distacco critico, come farebbe un revisore diverso dallo sviluppatore.

Ancora meglio è spostare la revisione in Cowork o in Code, dove è possibile utilizzare plugin dedicati alla revisione del codice e all'analisi di sicurezza. In questi ambienti Claude accede direttamente ai file del progetto e può condurre una revisione sistematica, invece di lavorare su frammenti di codice copiati nella conversazione. Il plugin **Code Review**, costruito e verificato da Anthropic, offre una revisione con agenti specializzati e un filtro basato su livelli di confidenza che aiuta a distinguere i problemi reali dai falsi allarmi. In Code è disponibile anche il comando `/security-review`, integrato senza bisogno di installare nulla, che analizza il progetto cercando le vulnerabilità più comuni: SQL injection, XSS, problemi di autenticazione, gestione insicura dei dati e vulnerabilità nelle dipendenze.

Oltre ai plugin di Anthropic, la pagina claude.com/plugins (https://claude.com/plugins) ospita plugin di terze parti che non sono verificati da Anthropic ma estendono le capacità di revisione. **Semgrep** individua vulnerabilità in tempo reale e guida Claude a scrivere codice sicuro fin dall'inizio. **CodeRabbit** esegue revisioni del codice con oltre quaranta analizzatori, inclusi controlli di sicurezza. **Aikido Security** offre scansioni per credenziali esposte e vulnerabilità nell'infrastruttura. Il catalogo si evolve rapidamente, e vale la pena cercarne di nuovi al momento in cui si legge.

La richiesta di revisione, che si usi un plugin o una semplice conversazione, va formulata in modo esplicito. Ad esempio: "Analizza questo progetto per vulnerabilità di sicurezza. Controlla in particolare se ci sono credenziali nel codice, dati esposti al client che dovrebbero restare sul server, e input non validati". Specificare cosa cercare è importante perché, come ha mostrato uno studio di Backslash Security, i modelli linguistici generano codice significativamente più sicuro quando ricevono indicazioni esplicite sulla sicurezza. Lo stesso principio vale per la revisione: una richiesta generica produce una revisione generica, una richiesta specifica produce risposte più utili.

Il risultato va letto con un'avvertenza importante. Anche in una sessione separata, anche con l'aiuto dei plugin, è sempre Claude che rivede codice generato da Claude. È una verifica utile, non una garanzia. Claude può individuare errori evidenti, segnalare pattern rischiosi e suggerire miglioramenti concreti. Ma condivide con l'istanza che ha scritto il codice gli stessi schemi ricorrenti e gli stessi punti ciechi. Per i progetti di livello due e tre della scala, questa revisione interna è un primo passo, non l'ultimo.

Le regole nel CLAUDE.md

I capitoli 4 e 7 hanno spiegato cosa è il file CLAUDE.md e come usarlo per guidare il comportamento di Claude durante la costruzione di un progetto. In questo contesto, il CLAUDE.md diventa uno strumento di sicurezza preventiva, perché permette di stabilire regole che Claude applica fin dall'inizio, senza doverle ripetere a ogni richiesta.

Alcune regole di sicurezza efficaci da inserire nel CLAUDE.md di un progetto:

- Non inserire mai credenziali, chiavi API o token direttamente nel codice. Usare sempre variabili d'ambiente.
- Validare tutti gli input ricevuti dall'esterno prima di utilizzarli.
- Non inviare al client dati che non sono strettamente necessari per la visualizzazione.
- Richiedere conferma esplicita prima di qualsiasi operazione che cancella o modifica dati in modo irreversibile.
- Se il progetto usa Git, creare un file .gitignore che escluda i file contenenti credenziali (file .env, file di configurazione locale) dal repository.

Non servono competenze di programmazione per scrivere queste regole. Sono indicazioni in linguaggio naturale che Claude interpreta e applica durante la generazione del codice. La differenza tra un progetto con queste regole e uno senza è la stessa che c'è tra un artigiano che lavora con una lista di controlli e uno che si affida alla memoria.

I permessi come protezione

Il capitolo 3 ha descritto i livelli di permesso disponibili in Cowork e in Code. In quel contesto la discussione riguardava la comodità operativa, il modo in cui ciascun livello bilancia autonomia di Claude e controllo dell'utente. Ma quei livelli di permesso sono anche una misura di sicurezza, probabilmente la più immediata a disposizione di chi fa vibe coding.

Il principio è quello del *privilegio minimo*: dare a Claude solo i permessi necessari per il compito in corso, e niente di più. Se Claude sta scrivendo un file di documentazione, non ha bisogno di poter cancellare cartelle. Se sta modificando l'interfaccia di un'applicazione, non ha bisogno di accedere ai file di configurazione del server. In Cowork, la finestra di autorizzazione chiede conferma prima di ogni azione sul filesystem. In Code, il menu a tendina offre quattro livelli di autonomia, dalla richiesta di autorizzazione per ogni operazione fino all'esecuzione libera senza conferme.

La scelta del livello giusto dipende dalla fase di lavoro e dal livello di rischio del progetto. Per i progetti dei livelli zero e uno della scala, un approccio più permissivo accelera il lavoro senza rischi significativi. Per i livelli due e tre, mantenere i permessi restrittivi e autorizzare esplicitamente ogni operazione è una protezione concreta contro errori accidentali, sia di Claude sia propri.

La qualità di ciò che non si vede

Le sezioni precedenti si sono concentrate sulla sicurezza, ovvero su ciò che può causare danni immediati e concreti. Questa sezione affronta un problema diverso, meno urgente ma altrettanto importante sul lungo periodo: la qualità interna del codice. Un'applicazione può funzionare perfettamente, superare tutti i controlli di sicurezza, e contenere al suo interno una struttura che la rende progressivamente più difficile da modificare, estendere e mantenere.

Il debito tecnico

Per capire il concetto, si pensi a una casa costruita in fretta. Le stanze funzionano, il tetto non perde, gli impianti sono a norma. Ma i muri interni sono stati messi dove era più comodo in quel momento, senza pensare a come si sarebbe usata la casa tra cinque anni. Quando si decide di unire due stanze o aggiungere un bagno, si scopre che i tubi passano proprio dove servirebbe abbattere un muro, e che un intervento che doveva essere semplice richiede di rifare mezzo piano.

Il codice generato dalle AI tende ad accumulare questo tipo di problema in modo caratteristico. Quando si chiede una nuova funzionalità, Claude spesso non modifica il codice esistente per integrarla in modo organico: aggiunge nuovo codice accanto a quello vecchio, duplicando logica che già esiste, creando percorsi paralleli che fanno la stessa cosa in modi leggermente diversi. Ogni singola aggiunta funziona, ma l'insieme diventa progressivamente più fragile. Modificare un punto richiede di modificarne altri tre, e ogni intervento rischia di rompere qualcosa che prima funzionava.

Uno studio pubblicato nel 2026 su ArXiv (https://arxiv.org/abs/2603.28592) ha analizzato oltre 6.000 repository pubblici su GitHub contenenti codice generato da AI, riscontrando che il debito tecnico non risolto è cresciuto da poche centinaia di problemi a inizio 2025 a oltre 110.000 problemi a febbraio 2026. Il codice generato dalle AI non accumula solo vulnerabilità: accumula complessità strutturale a un ritmo accelerato rispetto al codice scritto manualmente.

Chiedere a Claude di migliorare

Il debito tecnico non è inevitabile. È il risultato di un modo di lavorare in cui si chiede sempre "aggiungi questa funzionalità" senza mai chiedere "adesso metti in ordine". Il lettore di questo manuale può contrastarlo con tre richieste concrete, da usare a ogni milestone significativa del progetto, ovvero ogni volta che un gruppo di funzionalità è completo e funzionante.

La prima richiesta è *semplificare*. "Ci sono parti di questo codice che fanno la stessa cosa in modi diversi? Unificale." Claude individua le duplicazioni e le consolida, riducendo i punti che dovranno essere modificati

in futuro.

La seconda è *rifattorizzare*. "Il codice è cresciuto molto. Riorganizzalo per renderlo più chiaro e manutenibile, senza cambiare il suo comportamento." Rifattorizzare significa ristrutturare l'interno senza modificare l'esterno: l'applicazione continua a fare le stesse cose, ma il codice diventa più ordinato. È l'equivalente di riorganizzare i mobili in una stanza per avere più spazio, senza cambiare le pareti.

Per entrambe queste operazioni vale lo stesso accorgimento della revisione di sicurezza: è meglio farle in una sessione nuova, separata da quella di costruzione. Claude nella sessione dove ha scritto il codice tende a confermare le proprie scelte strutturali piuttosto che metterle in discussione. Una sessione nuova guarda il codice senza quel contesto e ristruttura con più libertà. In Cowork e in Code, il plugin **Code Simplifier** di Anthropic automatizza parte di questo lavoro, analizzando il codice modificato di recente e proponendo semplificazioni che preservano il comportamento dell'applicazione.

La terza è *documentare*. "Aggiungi commenti al codice che spieghino cosa fa ogni sezione principale e perché." La documentazione interna non serve a chi usa l'applicazione, serve a chi dovrà modificarla in futuro, che sia Claude in una sessione successiva, un altro strumento, o un professionista chiamato a intervenire. Senza documentazione, ogni modifica futura parte da zero nella comprensione del codice.

Queste tre richieste non richiedono competenze di programmazione. Richiedono solo la consapevolezza che il codice ha bisogno di manutenzione periodica, esattamente come qualsiasi altro prodotto del lavoro.

Quando servono competenze professionali

Il vibe coding funziona bene per una fascia di progetti che è più ampia di quanto molti pensino. Ma ha un confine, e riconoscerlo è una competenza in sé.

I segnali che un progetto ha superato la soglia del vibe coding si manifestano in modo graduale. Claude inizia a impiegare più tempo per ogni modifica, perché il codice è diventato complesso da navigare anche per lui. Le modifiche a una parte dell'applicazione rompono regolarmente altre parti che non si stavano toccando. Le sessioni di lavoro si riempiono di tentativi e correzioni, e il rapporto tra tempo investito e risultati ottenuti peggiora visibilmente. Ogni nuova funzionalità richiede workaround, soluzioni provvisorie che si sommano a quelle precedenti.

Quando si presentano questi segnali, il progetto non è fallito. Ha raggiunto un livello di complessità che richiede competenze diverse da quelle che il vibe coding può offrire. Lo spettro della delega introdotto nel capitolo 1 descrive esattamente questa situazione: la capacità di verificare il risultato (il terzo livello dello spettro) non è sufficiente quando il problema è strutturale, interno al codice, e non si manifesta come un difetto visibile ma come una crescente resistenza al cambiamento.

A questo punto le strade possibili sono due. La prima è coinvolgere un professionista che rifattorizzi il codice, mettendo ordine nella struttura accumulata e creando una base solida per la crescita successiva. La seconda, per i progetti dove la complessità è andata troppo oltre, è ricominciare la costruzione con una specifica migliore, informata da tutto ciò che si è imparato nella prima versione. In entrambi i casi, il lavoro fatto non è perso: ha prodotto una comprensione del problema che la versione precedente non aveva, e questa comprensione è il vero valore del progetto.

Condividere e pubblicare

Fin qui tutto ciò che si è costruito è rimasto sulla propria macchina. Il convertitore PDF del capitolo 5, il sistema di documentazione e l'arena di discussione del capitolo 6, gli script e gli strumenti nati lungo il percorso vivono in cartelle locali, accessibili solo a chi li ha creati. Per molti progetti è una condizione perfettamente adeguata: uno strumento personale che funziona bene non ha bisogno di un pubblico.

Ma ci sono situazioni in cui il passo successivo diventa naturale. Un collega chiede di provare quello strumento che gli è stato descritto. Un progetto raggiunge una maturità tale da meritare una condivisione più ampia. Oppure, più semplicemente, si vuole un backup sicuro del proprio lavoro, indipendente dal destino del disco rigido.

Questo capitolo copre quel passaggio: dalla cartella locale alla pubblicazione su GitHub, e da lì, per i progetti che lo richiedono, alla messa online accessibile a chiunque. Il capitolo si chiude con un bilancio dei costi del vibe coding e con alcune indicazioni su dove continuare.

Questo è anche un capitolo di consultazione. Il procedimento per pubblicare su GitHub è lo stesso per ogni progetto, e le istruzioni qui sono pensate per essere autosufficienti. Non è necessario rileggere i capitoli precedenti per seguirle.

Pubblicare il codice su GitHub

Un *repository* è una cartella di progetto con una memoria: oltre ai file contiene la cronologia completa di tutte le modifiche fatte nel tempo. Git è lo strumento che crea e gestisce questa cronologia sulla propria macchina. GitHub è un servizio che ospita una copia del repository in cloud, accessibile da qualsiasi dispositivo.

Pubblicare il codice su GitHub è utile anche quando il progetto è strettamente personale. Usare un repository GitHub, sia pubblico che privato, significa avere un backup remoto del proprio lavoro, protetto dalla rottura di un disco o dalla perdita di un computer. Ma i vantaggi vanno oltre la sicurezza, un repository pubblico diventa un portfolio di ciò che si sa costruire, una risorsa condivisibile con colleghi, un punto di partenza per collaborazioni future. Un repository privato è visibile solo al proprietario e può essere reso pubblico in qualsiasi momento.

Non serve che il progetto sia completo o rifinito. GitHub è pieno di lavori in corso, prototipi funzionanti, esperimenti abbandonati e ripresi. La pubblicazione non è un annuncio ufficiale, è un modo pratico di gestire il proprio lavoro.

Creare un account e un repository

Il primo passo è registrare un account gratuito su github.com (https://github.com).

Per creare un nuovo repository, dalla pagina principale di GitHub si seleziona il pulsante **New**. Le informazioni richieste sono il nome del repository, una breve descrizione e la visibilità (pubblico o privato). Se si sta pubblicando un progetto già esistente sulla propria macchina, è importante **non** inizializzare il repository con un file README, altrimenti GitHub creerà un contenuto iniziale che entrerà in conflitto con il codice locale al momento del primo caricamento.

Installare Git

Prima di usare i comandi descritti in questa sezione, Git deve essere installato sul proprio computer. Su Mac è spesso già presente: aprire il Terminale e digitare `git --version` per verificarlo. Se non è installato, il sistema proporrà automaticamente l'installazione. Su Windows, scaricare l'installer dal sito ufficiale git-scm.com (https://git-scm.com), avviarlo e accettare le impostazioni predefinite. Al termine dell'installazione, aprire il Prompt dei comandi o PowerShell e digitare `git --version` per verificare che tutto funzioni.

I comandi Git, dall'inizio alla pubblicazione

Git è lo strumento che tiene traccia delle modifiche ai file e le sincronizza con GitHub. Il capitolo 7 ne ha introdotto i concetti fondamentali nel contesto del salvataggio locale. Qui gli stessi comandi vengono presentati come riferimento completo, dalla configurazione iniziale alla pubblicazione quotidiana.

Configurazione iniziale (una volta sola, sul proprio computer):

```
git config --global user.name "Nome Cognome"
git config --global user.email "email@esempio.com"
```

Il nome e l'indirizzo email vengono associati a ogni modifica registrata. Devono corrispondere a quelli usati per l'account GitHub.

Primo collegamento con GitHub (una volta per ogni progetto):

La cartella del progetto deve contenere almeno i file del progetto stesso. Se si è seguito il metodo descritto nel capitolo 4, la cartella conterrà anche il file `spec.md` (la specifica di progetto in formato Markdown) e il file `CLAUDE.md`. È buona pratica aggiungere fin da subito un file `.gitignore` (descritto più avanti in questo capitolo) per evitare di pubblicare file che non devono finire nel repository.

```
cd percorso/cartella-progetto
git init
git add .
git commit -m "Primo commit: descrizione del progetto"
git remote add origin https://github.com/utente/nome-repo.git
git branch -M main
git push -u origin main
```

Ogni comando ha una funzione precisa. - `cd` sposta il terminale nella cartella del progetto. - `git init` trasforma quella cartella in un repository Git locale. - `git add .` dice a Git di includere tutti i file presenti. - `git commit -m "..."` registra un'istantanea del progetto con un messaggio che descrive lo stato attuale. -

`git remote add origin` collega il repository locale a quello appena creato su GitHub, usando l'indirizzo che GitHub mostra nella pagina del nuovo repository. - `git branch -M main` imposta il nome del ramo principale. - `git push -u origin main` carica tutto su GitHub. L'opzione `-u` serve solo la prima volta, per stabilire il collegamento predefinito tra locale e remoto.

Ciclo quotidiano (ogni volta che si vogliono salvare e pubblicare le modifiche):

```
git add .
git commit -m "Descrizione di cosa è cambiato"
git push
```

Tre comandi, sempre gli stessi. - `git add .` prepara tutte le modifiche. - `git commit -m "..."` le registra con un messaggio. - `git push` le carica su GitHub.

Il messaggio di commit è l'unica parte che cambia, ed è l'unica che richiede attenzione, una descrizione breve ma significativa di ciò che è stato modificato. "Aggiunta pagina contatti" è un buon messaggio. "Modifiche varie" non lo è, perché tra un mese non dirà nulla su cosa è cambiato.

Cosa escludere dalla pubblicazione

Non tutto ciò che si trova nella cartella del progetto va pubblicato su GitHub. Il file `.gitignore` dice a Git quali file e cartelle ignorare. Si crea nella cartella principale del progetto e contiene un elenco di nomi o pattern da escludere.

Le esclusioni più importanti riguardano le credenziali. Come discusso nel capitolo 8, file come `.env` che contengono chiavi API, password o token di accesso non devono mai finire in un repository, nemmeno privato. Altre esclusioni comuni sono le cartelle generate automaticamente dalle dipendenze del progetto (`node_modules` per JavaScript, `__pycache__` per Python, `venv` per gli ambienti virtuali), i file temporanei del sistema operativo (`.DS_Store` su Mac, `Thumbs.db` su Windows), e le cartelle di output che si possono rigenerare.

Non è necessario conoscere tutte le esclusioni a memoria. Si può chiedere a Claude di generare un file `.gitignore` adatto al proprio progetto, descrivendo le tecnologie utilizzate.

Il README come presentazione

Il file `README.md` è un file di testo in formato Markdown, lo stesso formato usato per i file del manuale e per la specifica di progetto. È la prima cosa che chiunque vede aprendo un repository su GitHub, perché GitHub lo visualizza automaticamente nella pagina principale del progetto. Per un progetto personale è anche una forma di documentazione per il sé futuro: tra sei mesi, rileggendo il README, si ricorderà cosa fa il progetto e come si usa.

Un buon README ha una struttura semplice. Inizia con il nome del progetto e una frase che spiega cosa fa. Segue una sezione che descrive come installarlo, quali dipendenze richiede, come avviarlo. Infine una sezione sull'uso, con un esempio concreto di cosa succede quando lo si avvia.

Chi ha seguito il metodo descritto nel capitolo 4 ha già buona parte di queste informazioni nel file `spec.md`, la specifica di progetto in formato Markdown che descrive cosa fa l'applicazione, come funziona e quali scelte architetturali sono state fatte. Chiedere a Claude di generare un README a partire dalla specifica è un modo rapido per ottenere un punto di partenza da rivedere e adattare.

La licenza

Se il repository è pubblico, chiunque può vedere il codice. Ma "vedere" non significa "usare liberamente", senza una licenza esplicita, il codice è protetto dal diritto d'autore e nessuno ha il diritto di riutilizzarlo.

Aggiungere un file `LICENSE` nella cartella del progetto chiarisce le regole. Le licenze più comuni per progetti software sono la MIT, che permette a chiunque di usare, modificare e redistribuire il codice con pochissimi vincoli, e la GPL, che in più obbliga chi modifica il codice a dover condividere le modifiche con la stessa licenza. Per contenuti non software, come documentazione o materiale didattico, la famiglia Creative Commons offre opzioni più adatte, inclusa la CC BY-NC che consente l'uso non commerciale.

Non è necessario essere giuristi per scegliere. Il sito choosealicense.com (https://choosealicense.com) guida la scelta con domande semplici. La regola pratica è partire dalla propria intenzione, chiunque può usarlo liberamente (MIT), chi lo modifica deve condividere (GPL), solo uso non commerciale (CC BY-NC).

Automatizzare la pubblicazione

Una volta che il repository è configurato e collegato a GitHub, il ciclo quotidiano di pubblicazione è sempre identico, preparare le modifiche, registrarle con un messaggio, caricarle. Tre comandi nella stessa sequenza, ogni volta. Per un non-sviluppatore che usa il terminale solo per questo, ricordare la sintassi esatta è un attrito inutile.

La soluzione è uno script: un file che contiene quei tre comandi e chiede soltanto il messaggio di commit. Si salva nella cartella del progetto e si avvia con un doppio clic (su Windows) o dal terminale (su Mac e Linux).

Su Windows, creare un file chiamato `pubblica.bat` nella cartella del progetto, con questo contenuto:

```
@echo off
echo.
echo === Pubblicazione su GitHub ===
echo.

cd /d "%~dp0"

set /p messaggio="Descrivi le modifiche: "

if "%messaggio%"=="" (
    echo Nessun messaggio inserito. Operazione annullata.
    pause
    exit /b
)

git add .
git commit -m "%messaggio%"
git push

echo.
echo Pubblicazione completata.
pause
```

Per avviarlo è sufficiente un doppio clic sul file. Lo script si posiziona automaticamente nella cartella giusta (`cd /d "%~dp0"` significa "spostati nella cartella dove si trova questo script"), chiede il messaggio, e se il messaggio è vuoto annulla l'operazione senza fare nulla.

Su Mac e Linux, creare un file chiamato `pubblica.sh` nella cartella del progetto, con questo contenuto:

```
#!/bin/bash
echo ""
echo "=== Pubblicazione su GitHub ==="
echo ""

cd "$(dirname "$0")"

read -p "Descrivi le modifiche: " messaggio

if [ -z "$messaggio" ]; then
    echo "Nessun messaggio inserito. Operazione annullata."
    exit 1
fi

git add .
git commit -m "$messaggio"
git push

echo ""
echo "Pubblicazione completata."
```

Prima di poterlo usare, è necessario renderlo eseguibile aprendo il terminale nella cartella del progetto e digitando `chmod +x pubblica.sh`. Da quel momento si avvia con `./pubblica.sh` dal terminale.

Lo script è volutamente minimale, fa esattamente ciò che serve e nulla di più. Ma è anche un micro-esempio di vibe coding. Si può chiedere a Claude di personalizzarlo aggiungere un controllo che mostra l'elenco dei file modificati prima di procedere, includere la data nel messaggio di commit, inviare una notifica quando la pubblicazione è completata. Ogni personalizzazione è un esercizio del metodo che il manuale ha descritto fin qui.

Rendere il progetto accessibile

Pubblicare il codice su GitHub permette ad altri di vederlo e scaricarlo, ma non di usarlo direttamente. Per vedere una pagina web o usare un'applicazione, serve che il progetto sia *online*, raggiungibile con un browser da qualsiasi dispositivo. Questo passaggio si chiama deployment.

Non tutti i progetti lo richiedono. Molti strumenti costruiti con il vibe coding funzionano perfettamente come applicazioni locali, senza bisogno di essere raggiungibili dall'esterno. Il convertitore PDF del capitolo 5 è un buon esempio: gira sulla propria macchina, si apre nel browser locale, fa il suo lavoro. La domanda da porsi è se qualcuno deve poter usare questo progetto senza accedere al mio computer.

GitHub Pages per siti statici

Se il progetto è un sito composto da file HTML, CSS e JavaScript, senza un componente server, GitHub lo può pubblicare gratuitamente attraverso il servizio GitHub Pages. Il procedimento parte dalle impostazioni del repository, nella sezione **Settings**, poi **Pages**, si seleziona il branch da cui pubblicare (di solito *main*) e la cartella di origine. In pochi minuti il sito è raggiungibile a un indirizzo del tipo `utente.github.io/nome-repo`.

Il sistema di documentazione descritto nel capitolo 6 funziona esattamente con questo meccanismo: ogni volta che il codice viene caricato su GitHub, una procedura automatica ricostruisce il sito e lo pubblica. GitHub Pages è adatto a documentazione, portfolio, landing page, applicazioni che girano interamente nel browser. Non è adatto a progetti che richiedono un server, come quelli costruiti con Flask o Node.js.

Piattaforme di deployment per applicazioni web

Per progetti con un componente server, come le applicazioni Flask costruite nel capitolo 5, servono piattaforme che ospitano e mantengono attivo il server. Vercel, Render e Railway sono tra le più utilizzate, e tutte offrono un livello gratuito sufficiente per progetti personali e piccoli strumenti.

Il meccanismo di base è simile per tutte, si collega il proprio repository su GitHub (quello remoto, non la cartella sul proprio computer) alla piattaforma, si configura il modo in cui l'applicazione deve essere avviata, e da quel momento ogni caricamento su GitHub aggiorna automaticamente la versione online. L'applicazione riceve un indirizzo web pubblico e resta accessibile senza che sia necessario tenere acceso il proprio computer. In pratica, il flusso di lavoro non cambia: si lavora in locale, si pubblica su GitHub con lo script o con i tre comandi, e la piattaforma di deployment si occupa del resto.

I dettagli di configurazione variano da piattaforma a piattaforma e cambiano nel tempo, quindi non è utile descriverli qui in modo procedurale. L'approccio più efficace è chiedere a Claude quale piattaforma è adatta al proprio progetto, descrivendo le tecnologie utilizzate e il tipo di applicazione. Claude può guidare la configurazione passo per passo, adattandola alla situazione specifica.

Quando il deployment non serve

La maggior parte dei progetti nati dal vibe coding non ha bisogno di essere online, strumenti personali, automazioni, convertitori, analisi di dati funzionano benissimo in locale. Pubblicare il codice su GitHub è già un passo significativo, che protegge il lavoro e lo rende condivisibile. Il deployment aggiunge valore solo quando qualcun altro deve poter usare il progetto senza avere accesso alla macchina su cui è stato costruito.

Quanto costa davvero

Il vibe coding non è gratuito. Non è nemmeno particolarmente costoso, ma un bilancio corretto richiede di considerare tutte le voci, comprese quelle che non si pagano in denaro.

L'abbonamento

Claude Desktop è disponibile in diversi piani. Il piano gratuito permette di esplorare la Chat con Claude ma ha limiti significativi per il vibe coding, dal numero di messaggi ridotto alla non disponibilità di Cowork e Code, che questo manuale ha descritto come strumenti centrali per la costruzione di progetti. Il piano Pro è sufficiente per la maggior parte dei progetti descritti in questo manuale. Il piano Max è pensato per sessioni intensive o progetti molto complessi, dove il volume di messaggi del piano Pro non basta.

I prezzi cambiano nel tempo, quindi è preferibile consultare la pagina ufficiale di Anthropic (https://www.anthropic.com/pricing) per le cifre aggiornate. Ciò che resta costante è la logica, si paga un abbonamento mensile che dà accesso a un certo volume di utilizzo, non un costo per singolo progetto.

L'hosting

Per chi decide di mettere un progetto online, i costi dipendono dalla piattaforma e dal volume di utilizzo. GitHub Pages è gratuito per siti statici. Le piattaforme di deployment come Vercel, Render e Railway offrono livelli gratuiti che bastano per progetti personali con traffico modesto. I costi iniziano quando il progetto ha molti utenti simultanei o richiede risorse server significative, una situazione che raramente si presenta per i progetti descritti in questo manuale.

L'unico costo fisso, facoltativo, è un dominio personalizzato. Registrare un nome come *mioprogetto.it* costa pochi euro l'anno e non è indispensabile: le piattaforme forniscono un indirizzo funzionante incluso nel servizio.

Il tempo

Il costo più significativo del vibe coding non è economico ma temporale. Il metodo riduce drasticamente il tempo rispetto all'imparare a programmare da zero, ma non è istantaneo. Un progetto richiede ore, non minuti, i progetti complessi, come quelli descritti nel capitolo 6, richiedono giorni o settimane di sessioni di lavoro distribuite nel tempo.

La curva di apprendimento esiste ed è reale. I primi progetti sono più lenti perché ogni passaggio è nuovo, configurare l'ambiente, formulare le richieste in modo efficace, testare i risultati, gestire gli errori. Con l'esperienza si sviluppa un'intuizione per cosa funziona, le sessioni diventano più produttive, i problemi si riconoscono prima.

Le alternative a confronto

Il vibe coding non è l'unica opzione per chi ha bisogno di uno strumento su misura, e non è sempre la migliore. Un confronto onesto con le alternative aiuta a capire dove si colloca.

Commissionare il progetto a uno sviluppatore professionista è la scelta che produce, potenzialmente, il risultato di qualità più alta. Il costo è significativamente maggiore, ma il prodotto sarà più robusto, meglio strutturato e più facile da mantenere nel tempo. Lo svantaggio è la dipendenza, ogni modifica, anche piccola, richiede un nuovo intervento (e un nuovo costo). Non si impara nulla del processo, e il progetto resta una scatola chiusa.

Gli strumenti no-code come Bubble, Glide o Airtable offrono un percorso più guidato. Per casi d'uso standard, come un modulo di raccolta dati o una semplice applicazione gestionale, possono essere più rapidi del vibe coding. I limiti emergono quando le esigenze si allontanano dai template previsti. In più, il progetto vive all'interno della piattaforma, cambiare strumento significa spesso ricominciare da zero.

Imparare a programmare è l'investimento con il rendimento più alto a lungo termine, ma anche quello che richiede più tempo. Per chi ha un altro lavoro principale, dedicare centinaia di ore all'apprendimento di un linguaggio di programmazione è raramente praticabile.

Rinunciare è sempre un'opzione legittima. Non tutti i problemi meritano uno strumento dedicato, e a volte una soluzione manuale, per quanto meno elegante, è più che sufficiente.

Il vibe coding si colloca in uno spazio preciso tra queste alternative. È adatto a chi ha bisogno di strumenti su misura, ha il tempo per costruirli iterativamente, e trova valore nel processo stesso di costruzione, nel capire a fondo il problema mentre si cerca la soluzione.

Dove andare da qui

Questo manuale ha coperto un percorso che va dal concetto di vibe coding alla pubblicazione di un progetto, passando per la configurazione dell'ambiente, la pianificazione, la costruzione, la gestione degli errori e la sicurezza. Ma il percorso del lettore non finisce con l'ultimo capitolo.

Risorse per continuare

La documentazione ufficiale di Anthropic è il punto di partenza più affidabile per approfondire le funzionalità di Claude. Il sito docs.claude.com (https://docs.claude.com) raccoglie guide, tutorial e riferimenti tecnici aggiornati. Per chi sviluppa con le API, docs.anthropic.com (https://platform.claude.com/docs/en/home) offre la documentazione tecnica completa.

Il sito dell'autore, ai-know.pro (https://www.ai-know.pro/), pubblica regolarmente contenuti in italiano sulle AI generative, inclusi articoli sul vibe coding e guide pratiche sugli strumenti di Claude.

Per le risorse su argomenti specifici, il suggerimento più utile è anche il più semplice, chiedere direttamente a Claude. A differenza di un elenco statico in un manuale, Claude può indicare documentazione aggiornata, tutorial recenti e risorse pertinenti al problema specifico che si sta affrontando.

Un panorama in evoluzione

Il capitolo 1 ha descritto come il termine *vibe coding* sia nato da un tweet nel febbraio 2025 e sia diventato parola dell'anno in pochi mesi. Da allora la pratica si è estesa oltre la programmazione: *vibe design*, *vibe ops*, *vibe deploying* sono espressioni che emergono in modo indipendente in contesti diversi, a conferma che il metodo, delegare a un'AI e verificare il risultato, si applica a qualsiasi dominio.

Gli strumenti cambiano con una velocità che rende qualsiasi descrizione dettagliata obsoleta nel giro di mesi. Ciò che oggi richiede configurazione manuale domani potrebbe essere automatico. Ciò che oggi è un limite potrebbe essere risolto dal prossimo aggiornamento.

La competenza che resta stabile, indipendente dallo strumento specifico, è quella che il manuale ha cercato di sviluppare: formulare problemi con chiarezza, valutare risultati con spirito critico, iterare fino a ottenere ciò che serve. Sono capacità che non dipendono da Claude, da Git o da una piattaforma di deployment. Sono il nucleo del metodo.

Il vibe coding non è un'identità da acquisire e non è un punto di arrivo. È uno strumento, tra i molti disponibili, per risolvere problemi che prima richiedevano competenze specialistiche. Il valore non sta nella tecnologia che si usa, ma nei problemi che si riesce a risolvere. Continuare a costruire ciò che serve, con gli strumenti che si hanno, è il modo migliore per andare avanti.