Coding agent: perché il modello più economico può costarti di più

Maria Cattini | 25/08/2026 | Intelligenza Artificiale

Coding agent AI: Un coding agent modifica il codice, esegue test, attraversa file diversi e può arrivare a dichiarare concluso un lavoro. Il problema nasce quando quel «fatto» non significa davvero «pronto».

È uno dei punti più interessanti emersi da una discussione tecnica che parte da esperimenti con Qwen 3.5 397B A17B e GLM-5.2: invece di chiedersi quale modello costi meno per milione di token, conviene misurare **quanto costa arrivare a una modifica che possiamo davvero accettare e consegnare**.

Possiamo chiamarlo **Cost to Delivery**.

È un criterio molto più utile del semplice prezzo dell'API, soprattutto per chi usa l'intelligenza artificiale per programmare senza essere uno sviluppatore professionista. E porta a una conseguenza pratica: non è affatto detto che convenga affidare tutto a un unico modello.

Che cosa significa davvero Cost to Delivery?

Immaginiamo due coding agent.

Il primo costa 1 euro per eseguire un'attività. Produce rapidamente il codice, ma dimentica un requisito. Serve una seconda revisione, poi una correzione, poi un altro controllo perché la modifica ha avuto una conseguenza inattesa su un altro file.

Il secondo costa 4 euro, ma completa correttamente il lavoro al primo tentativo e supera la revisione.

Qual è quello economico?

Il prezzo della prima chiamata direbbe il primo. Il costo reale del lavoro dice il secondo.

Per valutare un coding agent ha quindi più senso considerare almeno quattro componenti:

Cost to Delivery = implementazione + test + revisione + correzioni necessarie

A queste andrebbe aggiunto un quinto costo che difficilmente compare nelle dashboard dei provider: **il nostro tempo**.

Una mezz'ora passata a capire perché l'agente sostiene di aver completato una modifica che in realtà è incompleta non è gratuita.

È anche il motivo per cui classifiche e benchmark non bastano. Sapere che un modello ottiene un determinato risultato su un test standardizzato dice qualcosa sulle sue capacità. Non ci dice necessariamente quanto lavoro servirà sul nostro repository, con le nostre istruzioni e i nostri vincoli.

Come costruire un workflow con più modelli AI?

Il materiale di partenza propone una separazione interessante:

ChatGPT → architettura e specifica → coding agent → revisione indipendente → correzione → consegna.

Il principio può essere applicato senza legarsi a un marchio preciso.

L'errore sarebbe pensare: «Qual è l'AI migliore per programmare?».

La domanda più utile è: **quale AI voglio usare per ciascuna fase?**

1. Usa il modello più capace per progettare il lavoro

Prima di lasciare che un agente tocchi il repository, fagli avere un contratto preciso.

Non basta:

> Aggiungi questa funzione.

Meglio definire cosa deve cambiare, quali file o componenti possono essere coinvolti, cosa non deve cambiare, quali comportamenti esistenti devono essere preservati e quali condizioni permettono di considerare terminato il lavoro.

Qui ha senso spendere capacità di ragionamento.

Un modello più potente può essere impiegato come una sorta di capo progetto tecnico: studia il problema e produce un piano di implementazione. L'esecuzione può poi essere affidata a un modello meno costoso.

Non è una garanzia di successo. È un modo per evitare di pagare il modello più caro per ogni singola operazione.

2. Spezza l'implementazione quando il compito diventa grande

Un'altra indicazione emersa dal materiale è dividere il piano in sottopiani.

Ha senso soprattutto quando una modifica attraversa molti componenti.

Invece di ordinare all'agente di eseguire contemporaneamente refactoring, nuova funzionalità, aggiornamento dei test e documentazione, possiamo creare fasi verificabili.

La differenza è sostanziale: dopo ogni passaggio possiamo controllare se il contratto iniziale è ancora rispettato.

Per un utente non tecnico significa anche rendere gli errori più leggibili. Se qualcosa si rompe dopo cinque modifiche contemporanee, trovare la causa è difficile. Se ogni blocco viene controllato prima di procedere, il punto in cui il progetto ha deviato diventa più evidente.

3. Non lasciare che chi scrive il codice sia l'unico a controllarlo

Questa è probabilmente la regola più importante.

Se un agente implementa una funzione e subito dopo gli chiediamo «hai rispettato tutti i requisiti?»,

stiamo chiedendo allo stesso sistema di giudicare il proprio lavoro.

Meglio separare i ruoli.

Il workflow può diventare:

pianificatore → esecutore → revisore indipendente → esecutore per le correzioni → verifica finale.

Nel materiale viene usato Codex per una *adversarial review*: non una revisione pensata per confermare il risultato, ma per cercare ciò che è stato dimenticato.

Il revisore dovrebbe confrontare il risultato con la specifica originale, non soltanto leggere il codice e decidere se «sembra buono».

Come dare all'AI il contesto di un progetto grande?

Qui compare un problema molto concreto: un repository può contenere migliaia di file e una quantità di codice impossibile da copiare manualmente in una chat.

Una delle proposte contenute nel materiale è **Repomix**. Il progetto esiste ed è open source: raccoglie un repository in un singolo file strutturato e pensato per essere passato a un LLM. Può anche calcolare i token, rispettare i file da ignorare e comprimere la rappresentazione del codice.

Il comando rapido indicato dalla documentazione ufficiale è:

npx repomix@latest

Non è necessario installarlo globalmente. Per l'esecuzione tramite npx occorre avere un ambiente Node/npm disponibile.

Qui però serve una cautela importante. Creare un unico documento contenente il repository significa concentrare molto codice in un solo file. Prima di caricarlo su qualsiasi servizio AI bisogna verificare cosa contiene, quali file sono stati inclusi e le condizioni relative alla gestione dei dati del servizio scelto.

Repomix dichiara anche di integrare Secretlint per individuare file che corrispondono a formati noti di credenziali ed escluderli dall'output, ma questo non sostituisce una verifica personale.

Google AI Studio e NotebookLM possono diventare la memoria del progetto?

Nel materiale viene suggerito di sfruttare i grandi contesti di Gemini e NotebookLM. L'idea è valida, ma alcuni dettagli richiedono una correzione.

La documentazione Google attualmente disponibile indica, per esempio, un limite di input di 1.048.576 token per Gemini 2.5 Pro, mentre il precedente Gemini 1.5 Pro arrivava a 2.097.152 token. Non bisogna quindi assumere che «Google AI Studio gestisce due milioni di token» indipendentemente dal modello scelto: **il limite dipende dal modello**.

NotebookLM — ora indicato da Google anche come Gemini Notebook in parte della documentazione italiana — può invece funzionare come base documentale. Accetta tra l'altro file Markdown, testo, PDF e documenti Word. Una singola fonte può arrivare fino a 500.000 parole o 200 MB; i limiti sul numero di fonti cambiano in base al piano.

C'è però una distinzione importante: usare NotebookLM come archivio interrogabile del progetto non equivale automaticamente a dare a un coding agent accesso completo e permanente a quel repository.

La combinazione «Claude + NotebookLM tramite MCP» citata nel materiale può essere tecnicamente realizzata attraverso componenti di terze parti, ma non ho trovato nella documentazione ufficiale consultata una configurazione Google/Anthropic che consenta di presentarla come integrazione nativa supportata. Non la considererei quindi un passaggio standard di questa guida.

Quanto conviene spendere per il modello?

Anche qui i prezzi vanno verificati, perché cambiano.

Ad agosto 2026 Anthropic indica Claude Pro a 20 dollari al mese con fatturazione mensile e specifica che comprende Claude Code. Claude Max parte invece da 100 dollari al mese, con livelli di utilizzo superiori rispetto a Pro.

Questo rende ancora più interessante il Cost to Delivery.

Se spendiamo 30 euro di API al mese ma perdiamo ore in revisioni e correzioni, quei 30 euro non rappresentano il costo del nostro sistema.

La metrica da annotare per qualche settimana dovrebbe essere più simile a questa:

1. Costo dell'implementazione: quanto abbiamo speso per far eseguire il compito.
2. Numero di revisioni: quante volte un secondo modello ha trovato problemi.
3. Numero di correzioni: quante nuove sessioni sono servite.
4. Tempo umano: quanto abbiamo dovuto intervenire.
5. Delivery: il risultato ha superato test e verifica finale?

Dopo 20 o 30 attività avremo dati molto più utili del prezzo per token.

Un caso pratico: modificare una funzione senza perdere pezzi

Supponiamo di dover modificare una funzione presente in più punti del progetto.

Non chiediamo subito al coding agent di intervenire.

Passo 1. Un modello con buone capacità di ragionamento analizza repository e richiesta e produce il piano.

Passo 2. Trasformiamo il piano in requisiti verificabili: file coinvolti, comportamenti da preservare, test necessari e condizioni di completamento.

Passo 3. Un coding agent esegue la modifica. OpenCode, per esempio, supporta più di 75 provider LLM e modelli locali, quindi consente di sperimentare senza legare il workflow a un solo modello. La sua stessa documentazione avverte però che soltanto alcuni modelli funzionano bene contemporaneamente nella generazione di codice e nel *tool calling*.

Passo 4. Un secondo modello riceve specifica e risultato e cerca discrepanze.

Passo 5. Le anomalie tornano all'agente esecutore come richieste precise di correzione.

Passo 6. Si eseguono nuovamente test e verifica rispetto alla specifica originale.

Il risultato atteso non è «l'AI ha scritto il codice». È più severo:

la modifica soddisfa il contratto iniziale e abbiamo prove sufficienti per accettarla.

Gli errori che fanno aumentare il Cost to Delivery

Il primo è scegliere il modello esclusivamente dal listino API.

Il secondo è affidare progettazione, esecuzione e controllo allo stesso agente.

Il terzo è consegnare richieste enormi senza criteri di completamento verificabili.

Il quarto è confondere una grande finestra di contesto con una memoria perfetta: poter inserire moltissimo codice non significa che ogni dipendenza verrà interpretata correttamente.

Il quinto è il più insidioso: credere alla frase «task completato».

Un coding agent non dovrebbe decidere da solo quando possiamo fidarci del suo lavoro.

La regola pratica: compra meno token, compra meno errori

Per chi sperimenta con coding agent, la gara tra GPT, Claude, Gemini, GLM, Qwen, MiniMax o altri modelli rischia di distrarre dalla questione utile.

Non serve trovare il modello che costa meno.

Serve costruire il processo che arriva al risultato con **meno rework possibile**.

Un modello potente può progettare. Uno più economico può eseguire. Un modello indipendente può controllare. Gli strumenti di contesto possono rendere leggibile un repository grande.

Poi bisogna misurare.

Se dopo un mese il modello economico richiede sistematicamente quattro cicli di correzione mentre quello più costoso ne richiede uno, il prezzo per milione di token ha già smesso di essere la metrica decisiva.

Il numero da guardare è quello che compare alla fine del lavoro: **quanto ci è costato arrivare a una modifica di cui siamo disposti a fidarci?**
Coding agent AI: Un coding agent modifica il codice, esegue test, attraversa file diversi e può arrivare a dichiarare concluso un lavoro. Il problema nasce quando quel «fatto» non significa davvero «pronto».

È uno dei punti più interessanti emersi da una discussione tecnica che parte da esperimenti con Qwen 3.5 397B A17B e GLM-5.2: invece di chiedersi quale modello costi meno per milione di token, conviene misurare **quanto costa arrivare a una modifica che possiamo davvero accettare e consegnare**.

Possiamo chiamarlo **Cost to Delivery**.

È un criterio molto più utile del semplice prezzo dell'API, soprattutto per chi usa l'intelligenza artificiale per programmare senza essere uno sviluppatore professionista. E porta a una conseguenza pratica: non è affatto detto che convenga affidare tutto a un unico modello.

Che cosa significa davvero Cost to Delivery?

Immaginiamo due coding agent.

Il primo costa 1 euro per eseguire un'attività. Produce rapidamente il codice, ma dimentica un requisito. Serve una seconda revisione, poi una correzione, poi un altro controllo perché la modifica ha avuto una conseguenza inattesa su un altro file.

Il secondo costa 4 euro, ma completa correttamente il lavoro al primo tentativo e supera la revisione.

Qual è quello economico?

Il prezzo della prima chiamata direbbe il primo. Il costo reale del lavoro dice il secondo.

Per valutare un coding agent ha quindi più senso considerare almeno quattro componenti:

Cost to Delivery = implementazione + test + revisione + correzioni necessarie

A queste andrebbe aggiunto un quinto costo che difficilmente compare nelle dashboard dei provider: **il nostro tempo**.

Una mezz'ora passata a capire perché l'agente sostiene di aver completato una modifica che in realtà è incompleta non è gratuita.

È anche il motivo per cui classifiche e benchmark non bastano. Sapere che un modello ottiene un determinato risultato su un test standardizzato dice qualcosa sulle sue capacità. Non ci dice necessariamente quanto lavoro servirà sul nostro repository, con le nostre istruzioni e i nostri vincoli.

Come costruire un workflow con più modelli AI?

Il materiale di partenza propone una separazione interessante:

ChatGPT → architettura e specifica → coding agent → revisione indipendente → correzione → consegna.

Il principio può essere applicato senza legarsi a un marchio preciso.

L'errore sarebbe pensare: «Qual è l'AI migliore per programmare?».

La domanda più utile è: **quale AI voglio usare per ciascuna fase?**

1. Usa il modello più capace per progettare il lavoro

Prima di lasciare che un agente tocchi il repository, fagli avere un contratto preciso.

Non basta:

> Aggiungi questa funzione.

Meglio definire cosa deve cambiare, quali file o componenti possono essere coinvolti, cosa non deve cambiare, quali comportamenti esistenti devono essere preservati e quali condizioni permettono di considerare terminato il lavoro.

Qui ha senso spendere capacità di ragionamento.

Un modello più potente può essere impiegato come una sorta di capo progetto tecnico: studia il problema e produce un piano di implementazione. L'esecuzione può poi essere affidata a un modello meno costoso.

Non è una garanzia di successo. È un modo per evitare di pagare il modello più caro per ogni singola operazione.

2. Spezza l'implementazione quando il compito diventa grande

Un'altra indicazione emersa dal materiale è dividere il piano in sottopiani.

Ha senso soprattutto quando una modifica attraversa molti componenti.

Invece di ordinare all'agente di eseguire contemporaneamente refactoring, nuova funzionalità, aggiornamento dei test e documentazione, possiamo creare fasi verificabili.

La differenza è sostanziale: dopo ogni passaggio possiamo controllare se il contratto iniziale è ancora rispettato.

Per un utente non tecnico significa anche rendere gli errori più leggibili. Se qualcosa si rompe dopo cinque modifiche contemporanee, trovare la causa è difficile. Se ogni blocco viene controllato prima di procedere, il punto in cui il progetto ha deviato diventa più evidente.

3. Non lasciare che chi scrive il codice sia l'unico a controllarlo

Questa è probabilmente la regola più importante.

Se un agente implementa una funzione e subito dopo gli chiediamo «hai rispettato tutti i requisiti?», stiamo chiedendo allo stesso sistema di giudicare il proprio lavoro.

Meglio separare i ruoli.

Il workflow può diventare:

pianificatore → esecutore → revisore indipendente → esecutore per le correzioni → verifica finale.

Nel materiale viene usato Codex per una *adversarial review*: non una revisione pensata per confermare il risultato, ma per cercare ciò che è stato dimenticato.

Il revisore dovrebbe confrontare il risultato con la specifica originale, non soltanto leggere il codice e decidere se «sembra buono».

Come dare all'AI il contesto di un progetto grande?

Qui compare un problema molto concreto: un repository può contenere migliaia di file e una quantità di codice impossibile da copiare manualmente in una chat.

Una delle proposte contenute nel materiale è **Repomix**. Il progetto esiste ed è open source: raccoglie un repository in un singolo file strutturato e pensato per essere passato a un LLM. Può anche calcolare i token, rispettare i file da ignorare e comprimere la rappresentazione del codice.

Il comando rapido indicato dalla documentazione ufficiale è:

npx repomix@latest

Non è necessario installarlo globalmente. Per l'esecuzione tramite npx occorre avere un ambiente Node/npm disponibile.

Qui però serve una cautela importante. Creare un unico documento contenente il repository significa concentrare molto codice in un solo file. Prima di caricarlo su qualsiasi servizio AI bisogna verificare cosa contiene, quali file sono stati inclusi e le condizioni relative alla gestione dei dati del servizio scelto.

Repomix dichiara anche di integrare Secretlint per individuare file che corrispondono a formati noti di credenziali ed escluderli dall'output, ma questo non sostituisce una verifica personale.

Google AI Studio e NotebookLM possono diventare la memoria del progetto?

Nel materiale viene suggerito di sfruttare i grandi contesti di Gemini e NotebookLM. L'idea è valida, ma alcuni dettagli richiedono una correzione.

La documentazione Google attualmente disponibile indica, per esempio, un limite di input di 1.048.576 token per Gemini 2.5 Pro, mentre il precedente Gemini 1.5 Pro arrivava a 2.097.152 token. Non bisogna quindi assumere che «Google AI Studio gestisce due milioni di token» indipendentemente dal modello scelto: **il limite dipende dal modello**.

NotebookLM — ora indicato da Google anche come Gemini Notebook in parte della documentazione italiana — può invece funzionare come base documentale. Accetta tra l'altro file Markdown, testo, PDF e documenti Word. Una singola fonte può arrivare fino a 500.000 parole o 200 MB; i limiti sul numero di fonti cambiano in base al piano.

C'è però una distinzione importante: usare NotebookLM come archivio interrogabile del progetto non equivale automaticamente a dare a un coding agent accesso completo e permanente a quel repository.

La combinazione «Claude + NotebookLM tramite MCP» citata nel materiale può essere tecnicamente realizzata attraverso componenti di terze parti, ma non ho trovato nella documentazione ufficiale consultata una configurazione Google/Anthropic che consenta di presentarla come integrazione nativa supportata. Non la considererei quindi un passaggio standard di questa guida.

Quanto conviene spendere per il modello?

Anche qui i prezzi vanno verificati, perché cambiano.

Ad agosto 2026 Anthropic indica Claude Pro a 20 dollari al mese con fatturazione mensile e specifica che comprende Claude Code. Claude Max parte invece da 100 dollari al mese, con livelli di utilizzo superiori rispetto a Pro.

Questo rende ancora più interessante il Cost to Delivery.

Se spendiamo 30 euro di API al mese ma perdiamo ore in revisioni e correzioni, quei 30 euro non rappresentano il costo del nostro sistema.

La metrica da annotare per qualche settimana dovrebbe essere più simile a questa:

1. Costo dell'implementazione: quanto abbiamo speso per far eseguire il compito.
2. Numero di revisioni: quante volte un secondo modello ha trovato problemi.
3. Numero di correzioni: quante nuove sessioni sono servite.
4. Tempo umano: quanto abbiamo dovuto intervenire.
5. Delivery: il risultato ha superato test e verifica finale?

Dopo 20 o 30 attività avremo dati molto più utili del prezzo per token.

Un caso pratico: modificare una funzione senza perdere pezzi

Supponiamo di dover modificare una funzione presente in più punti del progetto.

Non chiediamo subito al coding agent di intervenire.

Passo 1. Un modello con buone capacità di ragionamento analizza repository e richiesta e produce il piano.

Passo 2. Trasformiamo il piano in requisiti verificabili: file coinvolti, comportamenti da preservare, test necessari e condizioni di completamento.

Passo 3. Un coding agent esegue la modifica. OpenCode, per esempio, supporta più di 75 provider

LLM e modelli locali, quindi consente di sperimentare senza legare il workflow a un solo modello. La sua stessa documentazione avverte però che soltanto alcuni modelli funzionano bene contemporaneamente nella generazione di codice e nel *tool calling*.

Passo 4. Un secondo modello riceve specifica e risultato e cerca discrepanze.

Passo 5. Le anomalie tornano all'agente esecutore come richieste precise di correzione.

Passo 6. Si eseguono nuovamente test e verifica rispetto alla specifica originale.

Il risultato atteso non è «l'AI ha scritto il codice». È più severo:

la modifica soddisfa il contratto iniziale e abbiamo prove sufficienti per accettarla.

Gli errori che fanno aumentare il Cost to Delivery

Il primo è scegliere il modello esclusivamente dal listino API.

Il secondo è affidare progettazione, esecuzione e controllo allo stesso agente.

Il terzo è consegnare richieste enormi senza criteri di completamento verificabili.

Il quarto è confondere una grande finestra di contesto con una memoria perfetta: poter inserire moltissimo codice non significa che ogni dipendenza verrà interpretata correttamente.

Il quinto è il più insidioso: credere alla frase «task completato».

Un coding agent non dovrebbe decidere da solo quando possiamo fidarci del suo lavoro.

La regola pratica: compra meno token, compra meno errori

Per chi sperimenta con coding agent, la gara tra GPT, Claude, Gemini, GLM, Qwen, MiniMax o altri modelli rischia di distrarre dalla questione utile.

Non serve trovare il modello che costa meno.

Serve costruire il processo che arriva al risultato con **meno rework possibile**.

Un modello potente può progettare. Uno più economico può eseguire. Un modello indipendente può controllare. Gli strumenti di contesto possono rendere leggibile un repository grande.

Poi bisogna misurare.

Se dopo un mese il modello economico richiede sistematicamente quattro cicli di correzione mentre quello più costoso ne richiede uno, il prezzo per milione di token ha già smesso di essere la metrica decisiva.

Il numero da guardare è quello che compare alla fine del lavoro: **quanto ci è costato arrivare a una modifica di cui siamo disposti a fidarci?**