> Hello World

看懂 AI 開發工具的計費結構

如何省得到錢

我送出去的那句話，究竟被算了什麼錢？以 GitHub Copilot 與 VS Code 實作，機制跨工具通用

Duran Hsieh · Microsoft MVP · GitHub Star

ABOUT ME · 講師介紹

Duran Hsieh（謝政廷）



開發 · 測試 · 效能調教　GitHub Copilot　DevOps　Azure

專長於開發、測試、效能調教、GitHub、GitHub Copilot、DevOps、Azure 開發相關技術，曾擔任技術顧問，提供國內外企業技術諮詢與數位轉型。台灣第一位 GitHub Star、Study4TW 社群核心成員、GDG Taichung 共同創辦人、微軟最有價值專家、書籍作家、大學業界講師、技術部落客。

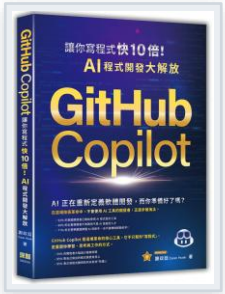
Duran Blogger	https://dog0416.blogspot.com
YouTube · FB	Duran 技術冶煉廠

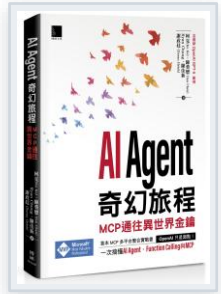
認證 · 社群

Microsoft MVP Most Valuable Professional

著作 BOOKS







AGENDA · 90 分鐘

今天的路線：從一句 Hello World 到團隊落地

開場實驗　同一句話送兩次，價差 6 倍　(5 分)

第 1 章　原理：帳單到底怎麼算　(20 分)

第 2 章　日常習慣：不寫程式也能省　(25 分)

第 3 章　工具與 MCP：Agent 的結構成本　(15 分)

Demo　同一個 bug，四種設定　(10 分)

第 4 章　進階：自建 AI 應用的手段　(8 分)

第 5 章　在地情境與跨工具　(4 分)

總結與清單　(3 分)

實驗 · 同一句話 · 送兩次

在 VS Code 送出 Hello World

第一次

6.64

個計價單位

第二次

< 1

個計價單位

同樣 11 個字元，價格差了 **6 倍以上**。差在哪裡？

拆解 · 那份請求長什麼樣

你打的字，只佔請求的一小角



← 工具在背後自動組出來的

你打的 Hello World ↑

常駐指令檔

copilot-instructions.md、AGENTS.md、Skills 描述，每一輪都帶

工具與 MCP schema

每個工具的名稱、說明、參數格式，開越多越長

對話歷史

之前所有的問答、讀過的檔案、工具結果

比例為示意。跨工具佐證：Claude Code 只回一句「OK」前就先送出約 33,000 token（Systima 2026/07 側錄）

第 1 章 · 原理

帳單到底怎麼算

四個原理：token 單價、無狀態重送、前綴快取、agent 迴圈。看懂這章，後面每個技巧都只是在拉其中一根槓桿

原理 1 · 計價單位

帳單算的是 token，而且有四種價格

文字先被切成 token 再送進模型。英文常見詞多半一個 token，程式碼符號和中文切得更碎。

Hello ␣World

修 好 失敗 的 測 試

切法為示意，實際依各模型 tokenizer 而定

輸入

你送進去的整份請求，基準價

快取輸入

開頭沒變的部分，約一折

輸出

模型寫出來的字，常是輸入的 3–5 倍

推理

看不到的思考，照輸出價計費

例：Claude Sonnet 每百萬 token 輸入 $3、輸出 $15；Copilot AI Credits 依輸入、輸出與快取 token 及模型單價計算

TIP 05 · 計價槓桿 · GitHub Copilot

Copilot 改按 token 計費：省 token 即省額度

原理

Premium request 已由 GitHub AI Credits 取代，依模型與輸入、輸出、快取 token 計價。以前「一次請求」固定扣，現在上下文越肥扣越多。

怎麼做

- 精簡 .github/copilot-instructions.md，它每次請求都載入
- 程式碼補全與 Next Edit suggestions 不扣 AI Credits
- Code review 另計 Actions 分鐘數，別對每個 push 都觸發
- Auto 模型選擇曾享 9 折倍率，改制後以官方文件為準

生效日

06/01

2026 年起所有 Copilot 方案改為依 token 用量計費

來源：GitHub Blog〈GitHub Copilot is moving to usage-based billing〉2026/04/27、GitHub Changelog

FRAME · 先分清楚目標

「省 token」其實是三件不同的事

減少 token 數

送進或產出的 token 真的變少，上下文也更乾淨

指令檔瘦身、Tool Search、RTK、LLMLingua、TOON、RAG

降低計費

token 數不變，但每個 token 變便宜

Prompt Caching、Batch API、模型路由、自架模型

縮短延遲

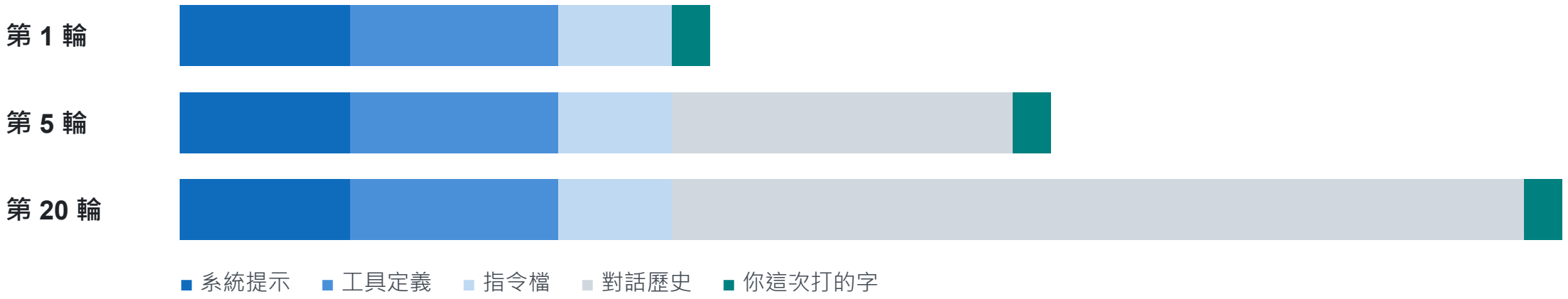
回應更快，但不一定更省錢

快取命中、prefix caching、降低推理強度；Batch 則是反過來用時間換價格

所以各 Tip 的百分比量的是不同東西，不能相加

原理 2 · 無狀態

模型沒有記憶：每一輪都把全部重送一次



你這次打的字往往是最小的一段。前三段每輪固定重送，歷史則越長越貴，整段對話的累計成本會隨輪數近似平方成長。

示意圖，比例非實測

WHY · 問題在哪

Token 不是花在你的問題，而是花在「每輪重送」

LLM 沒有記憶：每一輪都要重送系統提示、工具定義、指令檔與完整對話歷史。真正的成本大頭是這些常駐內容的累積。

31×

第 30 則訊息的成本可能是第 1 則的 31 倍：歷史越長，每輪重讀越貴

55K

接上 GitHub、Slack、Sentry、Grafana、Splunk 五個 MCP，工具定義就先吃掉約 55,000 token

30–50%

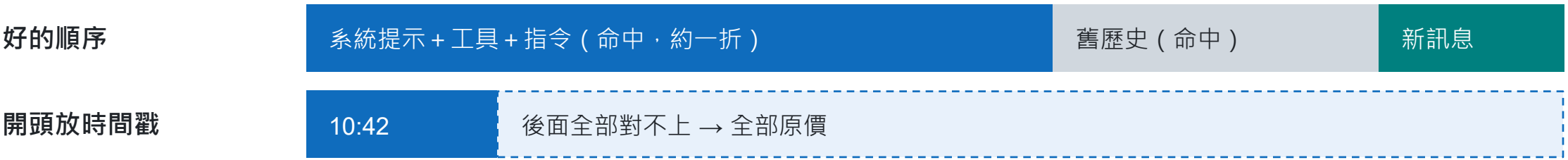
Coding agent 每輪中，系統提示、Skills 與 MCP 清單的開機開銷占比

來源：YouTube〈18 Claude Code Token Hacks〉摘要、Anthropic（經 WorkOS 整理）、ComputingForGeeks

原理 3 · 快取

快取：前綴一字不差，才能便宜重用

模型處理每個 token 時會算出中間結果（KV cache）。下一次請求的開頭若完全相同，供應商就直接重用，這段以約一折計價、回應也更快。



不變的放前面，會變的放最後

對話中別換模型、工具組、指令檔

有時效：常見 5–10 分鐘沒用就失效

原理 3 · 補充

快取能留多久？各家不一樣

同一段對話隔幾個小時再接著問，第一則訊息會不會 cache miss，取決於各家的快取存活時間。Claude Code 訂閱方案為 1 小時，改用 usage credits 或 API key 時降為 5 分鐘；圖中其他數字為整理時點資料，上台前請以官方文件核對。



TIP 01 · 試算

快取試算：為什麼不是 -90%

呼叫 100 次，每次輸入 10,000 token，其中 8,000 是固定前綴；快取讀取以一折計價。

輸入計費單位	不用快取	使用快取
第 1 次	10,000	10,000
第 2–100 次（每次）	10,000	2,800
合計	1,000,000	287,200

每次 2,800 = 變動的 2,000 + 前綴 8,000 × 0.1。未計快取寫入溢價、輸出與推理 token。

輸入費用

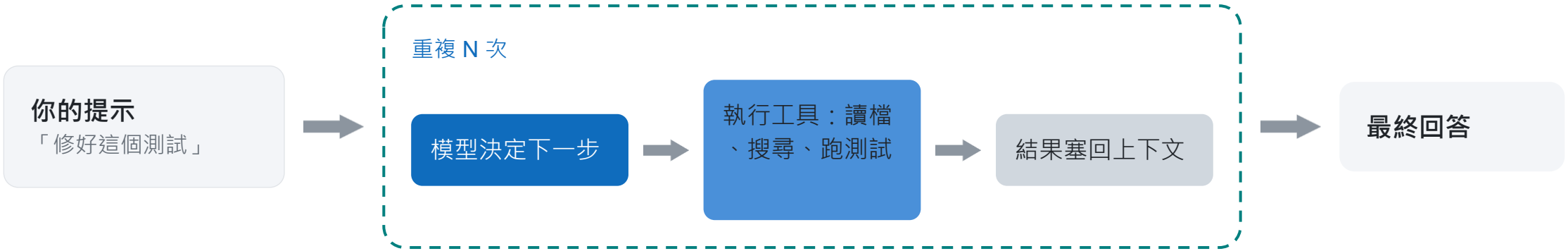
-71.3%

一折的折扣只作用在可快取的那 80%

示例試算，非特定模型報價；實際費率請依各家官方價格重算

原理 4 · AGENT 迴圈

一個任務，其實是十幾次 API 呼叫



每一圈都是完整請求

原理 2 的整條長條，每次工具呼叫都要再送一遍

工具輸出會累積

讀過的檔案、測試 log 都留在上下文，後面每圈都帶著

走錯路最貴

每多一圈探索或重試，成本就疊一層

真實案例 · AGENT 迴圈失控

一個簡單任務，兩分鐘燒掉 240 萬 token

2.4M

VS Code Agent 模式，起始提示只有約 3,000 token。兩分鐘後，累積的輸入 token 已經到 240 萬。

原因：每次工具呼叫都重送整包系統提示與歷史，agent 還讀了大量用不到的檔案。

200K

第 10 次工具呼叫時，單一請求的大小。每一步都比上一步更貴

來源：GitHub Community Discussion #198015（用戶自述）

為什麼帳單會爆炸

單價一直跌，帳單卻一直漲

5–30×

Agent 工作流程每個任務耗用的 token，是一般聊天的 5–30 倍；一個任務可觸發 10–20 次模型呼叫

62%

Agent 帳單花在重送系統提示、工具定義與歷史：付錢讓模型重讀它看過的內容

30%↓

輸入越長品質越差：中段位置的準確率可掉 30%，花更多錢還換來重試

4 月

Uber 2026 全年 AI 預算在 4 月中就用完。省錢的關鍵在架構，不在等降價

來源：Gartner、Cockroach Labs、Stanford、Chroma（18 個前沿模型測試）、Epoch AI；整理自〈LLM 成本優化實戰指南〉

補充 · 成本公式

所有技巧，都在拉這三根槓桿

呼叫次數 × (每次的 token 數 × 每個 token 的單價)

少呼叫

精準提示、先規劃、斷路器、交給確定性工具、一次做對不重試

Tip 09 · 22 · 23 · 26

少送

瘦身常駐內容、清掉舊歷史、過濾工具輸出、只取相關片段

Tip 06–08 · 11–17 · 21 · 25

便宜送

快取命中、批次半價、選對模型與推理強度、自架模型

Tip 01–05 · 19 · 20

第 2 章 · 日常習慣

不寫程式也能省：順序與習慣

先控 output、再瘦結構性 input、最後養成快取友善習慣。這章的每一件事，今天回去就能做

優先順序

省錢的正確順序

1

先控 output

單價最高、不能快取，而且輸出會變成下一輪的歷史，等於付兩次

2

再瘦結構性 input

指令檔、工具 schema、歷史每一輪都重送；即使快取打一折，乘上輪數仍是大頭

3

養成快取友善習慣

結構瘦了之後，讓剩下的前綴盡量走一折；避開會讓快取整段失效的動作

優先 1 · 控 OUTPUT

讓模型少寫、少想、只回該回的

指令檔預設簡短

寫進 copilot-instructions.md：只給 diff 與一句說明，不貼完整檔案、不寫總結報告

推理強度用預設

例行修改不需要高強度思考；難題再調高，推理 token 照輸出價收費

提示裡給格式與長度

「三點說明」「只回程式碼」「表格不超過五列」

問問題用 Ask 模式

只是要解釋，不需要 Agent 去讀檔、跑工具、多繞幾圈

對應附錄 Tip 04、24 · 社群指南估計「只給程式碼」可省程式碼任務 40–70% 輸出（olivomarco，非官方）

TIP 04 · 計價槓桿 · 外部 API

推理強度與輸出長度：按任務給預算

原理

推理 token 與輸出 token 都以輸出單價計費，通常是輸入的數倍。思考預算與回答長度要跟任務難度相符，例行工作不需要深度思考。

怎麼做

- OpenAI：reasoning.effort = minimal / low，text.verbosity = low
- Claude：effort 參數；Claude Code 用 /effort 調整
- 設 max_tokens；要求只輸出 diff、JSON 或固定段落數
- 給明確的完成條件，避免反覆檢查的過度思考

效益

192→0

同一個簡單問題的推理 token：effort 由 high 降到 minimal

來源：OpenAI GPT-5.5 指南與 Cookbook、DataCamp GPT-5 API 教學、AISI Inspect 文件

附錄 · 模型路由

別用大砲打蚊子

60–70%

分類、抽取、格式化、樣板生成等 agent 呼叫，小模型結果相當；旗艦與預算級模型單價可差 15–50 倍

85%

客服等特定領域微調 7B 級小模型，效果常勝過通用大模型，成本降 85%

$40K→$24K

審計 token 用量後，把簡單子任務改路由到便宜模型，月費降四成，不改任何產品功能

任務	模型	例子
簡單任務	小模型或內部開源模型	分類、格式化、摘要
標準生成	中階模型	一般 coding、單一模組修改
複雜推理	旗艦模型	架構設計、跨服務除錯

來源：CallSphere、Redis、Helicone、OpenPipe、Cockroach Labs；整理自〈LLM 成本優化實戰指南〉· 對應附錄 Tip 03

優先 2 · 瘦身結構性 INPUT

每一輪都要帶的東西，越小越好

結構	怎麼瘦	在 VS Code 哪裡改
常駐指令檔	只留每次都用得到的規則；流程搬進 Skill；用 applyTo 限定路徑	.github/copilot-instructions.md 與 *.instructions.md
工具與 MCP	只開這次任務用得到的工具；常用流程做成精簡的自訂 agent	聊天面板 Configure Tools、.vscode/mcp.json
對話歷史	換主題就開新對話；同主題岔出去用 Fork	Ctrl+N、/fork、/compact
搜尋範圍	log、大型 JSON、產出物不讓 agent 搜	search.exclude、.gitignore

官方實驗：VS Code 啟用 tool search 後，中位數用戶整個 session 的 token 降 18%

來源：VS Code Blog 2026/06/17、VS Code Docs〈Optimize AI credit usage〉· 對應附錄 Tip 06、07、11、24

TIP 06 · 上下文工程 · Coding Agent

指令檔瘦身，流程搬進 Skills 按需載入

原理

CLAUDE.md、AGENTS.md、copilot-instructions.md 每一輪都會重送。Skill 平常只有一行描述常駐，內文在觸發時才載入。

怎麼做

- 只放 80% 的 session 都用得到的規則，控制在 200 行內
- 超過 2KB 的流程（migration、PR review）改寫成 Skill
- 次要文件用 @docs/xxx.md 需要時再引用
- 寫成查找表，不要寫成長篇背景說明

試算

1M

5,000 token 的 CLAUDE.md × 200 輪對話，未計快取折扣

來源：KDnuggets、Composio、ComputingForGeeks、Analytics Vidhya 2026/05

TIP 07 · 上下文工程 · Coding Agent

換任務就 /clear，同任務用 /compact

原理

長對話每輪都重讀全部歷史。把狀態寫進檔案而不是留在對話裡，定期壓縮或重開，讓每輪只帶必要的內容。

怎麼做

- 主動壓縮並給保留提示：/compact Keep: 架構決策、待辦
- 結束前寫 session-handoff.md，新 session 讀它接手
- 用 /context、/usage 監看上下文組成與用量
- 取捨：壓縮會改變前綴、讓快取失效，別每幾輪就壓縮

案例

98.5%

一段 100+ 則訊息的對話裡，花在重讀舊歷史的 token 比例

來源：YouTube〈18 Claude Code Token Hacks〉摘要、Composio、GitHub Gist（dholdaway）、MindStudio

TIP 09 · 上下文工程 · Coding Agent

先規劃再動手，給 AI 一張 repo 地圖

原理

走錯方向的 token 全部白花，還要再花一輪修正。先確認方案，再告訴 AI 該先讀哪些檔，避免它掃描整個專案找實作。

怎麼做

- Claude Code Plan mode、Copilot Plan agent 先產出方案
- docs/repo_map.md：模組職責、入口檔、先讀哪些檔
- 提示寫明「只讀與 token 驗證相關的檔案」
- 直接給檔案路徑或貼出函式，而非「看一下 auth」

效益

少讀錯檔
少修正迴圈

錯誤路徑越早攔下，浪費的 token 越少

來源：Composio〈9 Ways to Cut Token Consumption in Claude Code〉、MindStudio、KDnuggets

TIP 22 · 補充 · Coding Agent

規劃與執行分離，走偏就 Undo

原理

規劃需要大量上下文與強模型；執行只需要計畫本身。把兩者拆到不同會話，執行端乾淨又便宜。走偏時回到乾淨狀態，比在壞掉的上下文上疊修正更省。

怎麼做

- 強模型寫 plan.md 或 issue，再開新會話用較便宜模型執行
- 每個里程碑 commit，當作可回復的檢查點
- 走偏時 Undo 或還原檔案、Fork 對話，不要連續追加修正
- 同一錯誤糾正兩次以上，就寫進 AGENTS.md 或 Skill

Google 的比喻

大象規劃
金魚執行

長上下文會話產出計畫，低 token 的新會話照計畫做

來源：Google Cloud〈Guide to AI tokenomics〉、olivomarco/github-copilot-token-optimization

TIP 22 延伸 · Claude Code

Claude Code 還原：/rewind 與 checkpoint

原理

每次送出提示前，Claude Code 會自動記下檔案狀態；走偏時回到任一則提示，不必手動復原。

怎麼做

- 輸入 /rewind，或在空輸入框連按兩次 Esc，開啟選單
- 選一則提示，再選：還原程式碼與對話、只還原對話、只還原程式碼
- 「Summarize」只壓縮對話，不會動到檔案
- 重要變更前仍先 commit：bash 與 subagent 改的檔案不在追蹤範圍

官方限制

bash 改的救不回來

rm、mv、sed -i 這類指令不在 checkpoint 範圍；subagent 的編輯、symlink 檔案也不會還原

來源：Claude Code Docs〈Checkpointing〉、Eon〈Claude Code's /rewind Won't Bring Back What Bash Deleted〉

TIP 22 延伸 · Codex CLI

Codex CLI 沒有穩定的 undo，git 就是保險

原理

Esc 只把對話倒回某一點，不還原程式碼。實驗性的 /undo 各版本表現不一，請用 /help 確認你的版本。

怎麼做

- 每個里程碑 commit，或開工前 git add -A 當檢查點
- 用 git worktree 讓 Codex 在獨立目錄工作，壞了整個丟掉
- Codex CLI v0.124 起 hooks 已穩定：工具執行前自動 commit 檢查點
- 第三方工具（agent-rollback、DiffBack）可逐檔還原，導入前請自行評估

openai/codex #11626

/rewind 還在排隊

社群要的是對話與檔案一起還原；截至 2026/06 仍是開放中的功能請求

來源：Daniel Vaughan〈Error Recovery and Rollback Patterns for Codex CLI〉、openai/codex #11626、#8214

TIP 22 延伸 · 對照

兩個工具的 undo，都不是 git 的替代品

項目	Claude Code	Codex CLI
還原對話	/rewind、Esc Esc	Esc（倒回 fork 點）
還原程式碼	/rewind 內建，只含編輯工具改的檔	無穩定內建，靠 git
盲區	bash 指令、subagent、外部改動	全部靠自己 commit
保險做法	重要變更前先 commit	里程碑 commit、worktree、hooks

結論：undo 只是第一道防線，真正的保險是每個里程碑 commit。上台前，先在測試 repo 讓 agent 改檔再還原一次，確認你的版本行為。

優先 3 · 快取友善習慣

哪些日常動作，會讓快取整段重算？

對話中換模型

不同模型不共用快取

中途改推理強度

VS Code 規劃將它標為會推高成本的動作

開關工具或 MCP

工具 schema 在前綴裡，一變後面全部對不上

改指令檔

對話進行中編輯 copilot-instructions.md

隔太久才回來

快取有存活時間，午餐後接續長對話最貴

壓縮對話

/compact 後歷史換成摘要，下一輪前綴改變

對策：**要換設定就開新對話**；固定的放前面；長對話別中途改裝備

案例 · PROJECTDISCOVERY

把會變的東西搬到最後，命中率 7% → 84%

他們怎麼做

分三層快取斷點：系統提示、靜態工具定義、對話滑動視窗。再把每一步都會變的 working memory 移出前綴、改放訊息尾端，命中率一夜從 7% 跳到 74%。

快取殺手：別放進前綴

即時時間戳、session 或 request ID、動態註冊的 MCP 工具、每次都變的個人化資料。系統提示開頭一行「現在時間」，就能讓一整天 0 命中、成本多 10 倍。

快取命中率
7% → 84%

整體成本
−59%

快取規模
9.8B tokens

來源：ProjectDiscovery（Neo 平台）、Cockroach Labs；整理自〈LLM 成本優化實戰指南〉

實機示範 · 讓計價數字自己說話

每做一個動作，再送一次 Hello World

#	動作	依機制預期	實測 credits
0	新對話，第一次送出	原價 + 寫入快取	[__]
1	同對話，再送一次	命中，約一折	[__]
2	開新對話，同模型同工具	固定前綴仍命中	[__]
3	對話中換模型	失效	[__]
4	Configure Tools 關掉一個 MCP	失效	[__]
5	指令檔加一行	失效	[__]
6	/compact 後再送	歷史段失效	[__]

量測：滑鼠移到回應上看 credits；Cache Explorer 看快取命中率 · 步驟見示範工具包 RUNBOOK 第 5 節

第 3 章 · 工具與 MCP

Agent 的結構成本：工具、輸出與迴圈

工具 schema、工具輸出、讀檔與重試，是 Agent 模式帳單的大頭。這章講怎麼把它們變小

案例 · GITHUB 自己的做法

修剪工具、改用 CLI，真實流程省 43–62%

修剪沒用到的 MCP 工具

40 個工具的 schema 每回合佔 10–15 KB，即使只用 2 個。比對「註冊 vs 實際呼叫」後修剪，每回合省 8–12 KB

固定讀取交給 CLI

PR diff、檔案內容用 gh CLI 在 agent 啟動前先下載，移出推理迴圈。最省的呼叫是不打的那通

每日 Token Auditor

自動揪出用量異常的工作流程，開 Issue 建議修法

一條錯誤設定，64 回合失控

一條錯誤的 allowlist 曾讓 agent 跑了 64 回合迴圈。沒有量測，就看不到

來源：GitHub 工程團隊實測；整理自〈LLM 成本優化實戰指南〉· 對應附錄 Tip 11、23、26

TIP 11 · 工具與 MCP · 外部 API + Agent

Tool Search：工具定義延遲載入

原理

每個 MCP 工具的 schema 每輪都進上下文，用不到也照付。改成只放一個搜尋工具，模型需要時再載入那 3–5 個工具的完整定義。

怎麼做

- 工具 10 個以上或定義超過 10K token 就該延遲載入
- Claude Code 在工具定義超過上下文 10% 時自動啟用
- 關掉當下用不到的 MCP server（每個可達上萬 token）
- 工具超過 30–50 個，選擇準確度會下降

效益

-85%

工具定義 token，且工具選擇準確度不降反升

來源：Anthropic Tool Search Tool（經 WorkOS、MCP Playground 整理）

TIP 13 · 工具與 MCP · Coding Agent

RTK：過濾 CLI 輸出再交給 AI

原理

測試、建置、git 的原始輸出大多是雜訊。在 shell 與 agent 之間放一層 Rust 代理，只回傳失敗項目與摘要，agent 本身無感。

怎麼做

- rtk init -g：Claude Code / Copilot；另支援 Gemini CLI、Cline
- rtk pytest、rtk cargo test：只顯示失敗
- rtk gain 看節省統計；rtk discover 找可優化指令
- 小心 crates.io 同名套件 Rust Type Kit

效益

-90%

測試指令輸出，例：25,000 → 2,500 token

來源：GitHub RTK（Rust Token Killer）README

TIP 14 · 工具與 MCP · Coding Agent

符號級檢索：用 LSP 找程式碼，不整檔讀

原理

Agent 預設逐檔讀取來找實作。透過語言伺服器（LSP）以符號層級查詢定義與參照，只取需要的函式，編輯也以符號為單位。

怎麼做

- Serena MCP：find_symbol、找參照、符號級編輯
- 以 uvx 或 uv tool 安裝，不要從 MCP 市集裝
- 最適合大型、強型別的程式庫
- 只動少數小檔案的任務，不必加掛

效益

讀一個函式
而非整個檔案

專案越大越明顯；同時降低改錯位置的風險

來源：GitHub oraios/serena README、PyPI serena-agent

TIP 15 · 工具與 MCP · 任何模型

Repomix --compress：只餵結構，不餵實作

原理

要讓模型理解整個 repo 的架構時，用 Tree-sitter 解析，只保留匯入、函式簽章、型別與類別結構，移除函式本體與迴圈細節。

怎麼做

- npx repomix@latest --compress
- 加上 --remove-comments、--remove-empty-lines
- --include "src/**/*.ts" 只打包需要的目錄
- 內建逐檔 token 計數，先看再丟；也有 MCP 模式

效益

保留介面
移除實作

適合架構分析、API 設計討論、外部模型的一次性問答

來源：Repomix 官方文件〈Code Compression〉、GitHub yamadashy/repomix

TIP 10 · 上下文工程 · Coding Agent

子代理分工：大量閱讀交給它，只帶回結論

原理

子代理在獨立上下文裡讀日誌、逐字稿、整個目錄，只把摘要交回主對話。主上下文保持乾淨，但子代理會重載自己的上下文。

怎麼做

- 用在大量閱讀：影片逐字稿、長日誌、全庫搜尋
- 要求只回傳結論、檔案路徑與行號
- 小任務不要開子代理，總 token 反而更多
- Claude Code /agents 建立專責子代理

取捨

-90%

主上下文用量（影片分析 20K→2K）；但總 token 可能是單代理的 7–10 倍

來源：GitHub hancengiz/youtube-transcript-mcp、YouTube〈18 Claude Code Token Hacks〉摘要

TIP 26 · 補充 · Coding Agent

確定性的工作交給工具，不交給模型

原理

格式化、lint、型別檢查、測試、程式碼產生都有確定的答案。用工具跑幾乎不花 token 也不會錯；交給 LLM 推論既貴又可能出錯。

怎麼做

- 讓 agent 執行 formatter、linter、編譯器，不要請它「幫我排版」
- 重複的雜務請 agent 寫成腳本，之後直接執行腳本
- 驗證左移：先跑本地單元測試，瀏覽器 E2E 留到最後
- 用 hooks 在存檔或提交時自動跑檢查，只把錯誤回給 agent

原則

能用腳本的不用推論

把 LLM 留給需要判斷的部分

來源：Google Cloud〈Guide to AI tokenomics〉第 3、6 條、社群整理

TIP 23 · 補充 · 無人值守 Agent

Agent 迴圈斷路器：別讓它半夜燒光預算

原理

解不開的 bug 會讓 agent 反覆重試，每輪都帶著越來越肥的上下文。最貴的不是單次呼叫，而是沒人看著的無限迴圈，必須像熔斷器一樣自動切斷。

怎麼做

- 設最大輪數、最大重試次數與單次任務 token 上限
- 偵測重複的工具呼叫或相同錯誤，觸發暫停並通知
- 用事件觸發喚醒 agent，不要讓它輪詢狀態
- 帳戶層設預算告警；Copilot 管理者可設用量預算

SRE 視角

把 token 當資源配額

限流、熔斷、告警，和保護任何共用服務的做法相同

來源：Google Cloud〈Guide to AI tokenomics〉第 10 條、loopguard、openclaw-loop-watchdog

TIP 25 · 補充 · 自建 MCP / 工具

MCP server 端先過濾，別回傳整包 JSON

原理

工具的回傳值會整包進上下文，而且之後每一輪都會重送。在 server 端先依參數過濾、分頁、只回必要欄位，模型只看到回答需要的那幾筆。

怎麼做

- 工具參數提供過濾條件：service、environment、route_id
- 預設分頁與筆數上限，並回傳總筆數讓模型決定是否翻頁
- 只回必要欄位；表格型結果可轉 TOON（Tip 17）
- 錯誤訊息簡短且可行動，不回整段 stack trace

範例 · API Gateway

200→3

查詢 route 設定時，先依 service 與環境過濾，只回傳相關的幾筆（示意）

來源：Anthropic〈Writing effective tools for agents〉、〈Code execution with MCP〉、社群整理

TIP 12 · 工具與 MCP · 自建 Agent

Code execution：讓中間資料留在沙箱

原理

傳統呼叫讓每筆中間結果都經過模型。改成讓模型寫程式呼叫工具，資料在沙箱內流轉、過濾，只把最終結果回傳上下文。

怎麼做

- 把 MCP server 映射成可瀏覽的程式碼 API 目錄
- Anthropic API：Programmatic tool calling（allowed_callers）
- Code Mode：只暴露 search 與 execute 兩個工具
- 注意沙箱安全與稽核；工具少、很少串接時不划算

效益

-98.7%

150K → 2K token（Google Drive 轉 Salesforce 範例）

來源：Anthropic〈Code execution with MCP〉2025/11、Maxim Bifrost、MCP Playground

DEMO

同一個 bug，四種設定

把前三章的技巧一層一層疊上去，看每個成功任務的成本差多少

DEMO · 別人量到的

公開實驗：同樣的趨勢

實驗	前 → 後	性質
VS Code 官方：開啟 tool search	中位數用戶整個 session token -18%	官方 A/B 實驗
Scott Spence：關掉多餘 MCP	MCP 工具 82.0K → 5.7K token	作者實測
Aetherius：加一份 CLAUDE.md	100K → 70K token，10–12 → 7–8 分鐘	同一 iOS 任務
Zilliz：語意程式碼檢索	73K → 44K token，F1 相同	30 題 SWE-bench
ComputingForGeeks：10 種工具	最佳 -43%；RTK 在此任務 0%	同一提示各跑一次

來源：VS Code Blog 2026/06/17、scottspence.com、aetherius-solutions.com、Zilliz claude-context、computingforgeeks.com

第 4 章 · 進階

自建 AI 應用時的進階手段

API 端快取與批次、上下文管理、資料壓縮、語意快取與自架推論。需要開發投入，但省幅最大

TIP 01 · 計價槓桿 · 外部 API

Prompt Caching：重複的前綴只付一成

原理

把不變的前綴（系統提示、工具定義、長文件）算好的 KV 結果存起來，下次前綴完全一致就以折扣價計費。前綴只要變一個字，後面全部失效。

怎麼做

- 靜態內容放前面、動態內容（使用者輸入）放最後
- Anthropic：cache_control 斷點；OpenAI：自動前綴比對
- Gemini：建立具名 cache 物件，可自訂 TTL
- 系統提示納入版控，避免無謂變動；監看快取命中率

效益

-90%

快取命中的輸入 token 成本；長提示延遲最多 -85%

來源：Anthropic Prompt Caching 公告、IntuitionLabs〈LLM Prompt Caching〉2026/09、GMI Cloud

TIP 02 · 計價槓桿 · 外部 API

Batch API：不急的工作一律半價

原理

不需即時回應的請求打包成 JSONL 非同步送出，供應商用離峰算力處理，換取輸入與輸出都打五折，承諾 24 小時內完成。

怎麼做

- 適合：評測集、批次分類/翻譯、文件摘要、夜間程式碼掃描
- OpenAI、Anthropic、Gemini、Azure OpenAI、Bedrock 皆提供
- 可與 Prompt Caching 疊加，折扣再往下
- CI/CD 的非阻斷檢查（如每日報告）優先改批次

效益

-50%

輸入 + 輸出 token，換 24 小時內交付

來源：IntuitionLabs〈Batch AI Processing〉2026/09、Digital Applied 批次定價追蹤 2026/08

TIP 08 · 上下文工程 · 自建 Agent

Context editing：自動清掉過期的工具結果

原理

Agent 迴圈最肥的是舊的檔案讀取、搜尋結果、測試輸出。接近上限時清掉舊 tool call 與結果，要保留的結論改寫進外部 memory 檔。

怎麼做

- Claude Developer Platform：啟用 context editing 與 memory tool
- 自建框架：超過門檻就把舊 tool result 換成佔位字串
- 架構決策、除錯心得寫入 memory，不留在上下文
- 自建聊天：保留最近幾輪原文，舊歷史交給 Haiku / mini 摘要

效益

-84%

100 輪網路搜尋評測的 token 用量，且任務不再因上下文耗盡而失敗

來源：Anthropic〈Managing context on the Claude Developer Platform〉、The Decoder

TIP 17 · 資料壓縮 · 任何模型

TOON：表格資料不要用 JSON 餵

原理

JSON 每筆資料都重複鍵名、引號與括號。TOON 用縮排取代括號，陣列只宣告一次欄位，之後以類 CSV 的列輸出。

怎麼做

- 儲存與 API 照用 JSON，送進 LLM 前才轉 TOON
- 效益最大：欄位一致的物件陣列（日誌、指標、清單）
- 深層巢狀或欄位不一時，compact JSON 可能更省
- 有 TypeScript、.NET、Rust 等實作與 CLI

效益

30–60%

均一陣列的 token 減少；基準測試準確率 73.9%（JSON 69.7%）

來源：TOON 規格（Johann Schopplich）、NuGet ToonFormat、Zartis〈Unpacking TOON〉

TIP 16 · 資料壓縮 · RAG 與長文件

LLMLingua：用小模型刪掉低資訊 token

原理

先用小型語言模型判斷提示中哪些 token 資訊量低並移除，保留指令與問題。LLMLingua-2 以 BERT 級編碼器做 token 分類，速度快 3–6 倍。

怎麼做

- pip install llmlingua；compress_prompt(..., target_token=200)
- 適合 RAG 檢索段落、會議逐字稿、長篇文件
- 已整合 LangChain、LlamaIndex、Prompt flow
- 中文等非英文在高壓縮率下流失嚴重，壓縮率要保守

效益

20×

最高壓縮比；RAG 只用 1/4 token，效能反而 +21.4%

來源：GitHub microsoft/LLMLingua（EMNLP'23、ACL'24）、PromptHub、arXiv 2608.26175

TIP 21 · 補充 · AI 應用

RAG 當壓縮器：只送相關片段，不送整份文件

原理

長上下文模型讓人很想把整份規格、手冊或程式庫塞進去。改用向量檢索只取最相關的幾個片段，token 少很多，也避開長文本中段被忽略的問題。

怎麼做

- 文件切塊、建向量索引，每次只取 top-k（常見 3–5 段）
- 加 reranker 過濾，寧可少而準，不要多而雜
- 取回的片段可再用 LLMLingua 壓縮（Tip 16）
- 函式庫文件用 Context7 MCP 依版本抓取，不貼整份

效益

整份文件
→ 幾個片段

降幅取決於文件大小與 k 值；品質取決於檢索準確度

來源：Microsoft LongLLMLingua（lost in the middle）、Upstash Context7、社群整理

TIP 19 · 資料壓縮 · AI 應用

語意快取：相似的問題直接回答，不呼叫模型

原理

把問題轉成向量，與已回答過的問題比相似度；夠接近就回傳快取答案。省掉的是整次呼叫，包含最貴的輸出 token。

怎麼做

- 前面先放精確比對層（hash），零誤判；再放語意快取
- GPTCache（Python）、Redis 向量快取、Redis LangCache
- 相似度門檻是關鍵：從 0.90 起調，避免答非所問
- 適合 FAQ、客服、維運手冊；寫程式等獨特請求命中率低

效益

40–70%

高重複工作負載的 API 成本降幅

來源：PremAI〈Semantic Caching for LLMs〉、Redis LangCache 文件、Spheron

TIP 20 · 自架推論 · 內部 / 開源模型

vLLM / SGLang Prefix caching：自架也能吃快取

原理

Prompt caching 底層就是 KV cache 重用。自架時由推論引擎直接重用共享前綴的 KV，省下重算 prefill 的 GPU 時間，同樣的卡能服務更多請求。

怎麼做

- vLLM：Automatic Prefix Caching（區塊雜湊）
- SGLang：RadixAttention 樹狀共享，預設啟用
- 多個 agent 共用同一份系統提示與工具定義
- 個人開發用 Ollama；搭配 Q4 / AWQ 量化降低硬體門檻

效益

6.4×

SGLang 在前綴密集工作負載的吞吐提升（最高）

來源：MorphLLM〈SGLang vs vLLM〉2026/09、Techsy H100 實測、GMI Cloud、vLLM 文件

第 5 章 · 在地與跨工具

台灣情境與換工具時的判斷

中文 token、規格書、方案額度、預算上限；換到任何工具，先找四個地方

台灣開發者容易忽略的

五個在地情境

繁中寫的常駐指令檔

同內容繁中約英文 2.1 倍 token，又每輪重送。指令檔用英文，對話照說中文

「改用中文提示比較省」

研究實測不成立，有模型反而多 1.28 倍，失敗重試更貴

直接拖 Word、Excel、PDF 規格書

先用 MarkItDown 轉成 Markdown，只給需要的章節

企業方案的促銷額度已結束

Business、Enterprise 6–8 月每人 $30、$70，9 月起回到 $19、$39

沒設預算上限

在 GitHub 帳單設定 budgets；Copilot code review 另耗 Actions 分鐘

[你的現場案例]

來源：Blake Crosley 多語 token 實測、arXiv 2604.14210、GitHub Blog 2026/04/27、Microsoft MarkItDown

跨工具．機制相同，按鈕不同

換了工具，先找這四個地方

要找的	GitHub Copilot（VS Code）	Claude Code
常駐指令檔	copilot-instructions.md、AGENTS.md	CLAUDE.md
看用量	回應上 hover、Agent Debug Logs	/context、/cost
管工具	Configure Tools、mcp.json	/mcp、settings
管歷史	Ctrl+N、/fork、/compact	/clear、/compact

判斷新工具時問三個問題：它每輪幫我送了什麼？我看得到用量嗎？快取怎麼計價？

Claude Code 的作法：status line 每輪都看得到

```
ctx 13% | last turn: in 120 · cache_r 48000 · cache_w 2300 · out 845 | $0.46
```

欄位	意思
ctx	Context window 使用率
in	上一輪新的輸入 token
cache_r	從快取重用的 token
cache_w	新寫入快取的 token
out	上一輪的輸出 token
$	這個工作階段的累計費用（美元）。設定步驟見附錄 B

如何解讀這些數字

快取正常

`in 120 · cache_r 48000`

大部分內容都走快取，這是理想狀態

第一輪

`cache_w 高 · cache_r 低`

快取才剛建立，通常屬於正常現象

快取可能失效

`cache_r 驟降 · cache_w 暴增`

快取過期或前面內容變了，這一輪會比較貴

Context 快滿了

`ctx 偏高`

用 /compact 壓縮對話，或用 /clear 開始新話題

$ 是這個工作階段的累計費用，不是單輪費用。

總結

帶走什麼

依導入成本分級的 26 個技巧、個人與團隊清單、四步驟落地，最後四句話

SUMMARY · 依導入成本分級

先摘低垂的果實，再動架構

今天就做 · 零程式碼

01 調整提示順序保護快取
04 降低推理強度與輸出長度
06 指令檔瘦身、流程改 Skill
07 /clear、/compact、交接檔
09 先規劃、給 repo 地圖
11 關掉用不到的 MCP
18 常駐內容英文化
22 規劃與執行分離、Undo
24 Ask 模式、applyTo

本週導入 · 裝工具

05 盤點 Copilot AI Credits 用量
10 大量閱讀改用子代理
13 RTK 過濾 CLI 輸出
14 Serena 符號級檢索
15 Repomix --compress
17 表格資料改 TOON
26 確定性工作交給工具

架構級 · 需開發

02 非即時工作改走 Batch
03 模型路由（含內部模型）
08 Context editing + memory
12 Code execution / Code Mode
16 LLMLingua 壓縮 RAG
19 精確比對 + 語意快取
20 自架 prefix caching
21 RAG 只送相關片段
23 Agent 迴圈斷路器
25 MCP server 端過濾

檢查清單 · 個人與團隊

帶走這份清單

個人

- ☐ 指令檔精簡，預設簡短輸出
- ☐ 推理強度用預設，難題才調高
- ☐ 問問題用 Ask，動手才用 Agent
- ☐ 只開需要的工具與 MCP
- ☐ 指定檔案與錯誤訊息，不說「看一下」
- ☐ 換設定就開新對話
- ☐ 大型 log 與資料加入 search.exclude

團隊

- ☐ 共用精簡指令檔範本，納入 code review
- ☐ MCP server 白名單
- ☐ 設定預算上限與每月用量報表
- ☐ 盤點 9 月後的方案內含額度
- ☐ KPI 看每個成功任務的成本
- ☐ 用示範 repo 做一次內訓實測

ROLLOUT · 團隊推動

沒有量測就沒有優化：四步驟落地

01

量測

/context、/usage、Copilot 用量報表、API 回應的 cached tokens 欄位，找出前幾名的消耗來源

02

砍常駐

指令檔、MCP 工具、Skills 清單逐項檢視；共用範本放進 repo，全團隊一致

03

開折扣

確認快取命中率，非即時工作改批次，依難度路由到合適模型

04

設護欄

預算告警與用量上限；每週檢視耗用最高的 session，回饋成指令檔與工具調整

KPI：每個成功任務的成本，而不是 token 減少百分比；小模型重試 8 次可能比大模型一次成功更貴

總結

四句話帶回去

問題在 context，不在 prompt

你打的字很少，貴的是每輪重送的指令、工具與歷史

順序：output → input → 快取

先控最貴的輸出，再瘦結構，最後別打斷快取

對的模型做對的事

六到七成的呼叫用小模型就夠，旗艦模型留給難題

量測決定一切

看每個完成任務的成本，不只看 token 數

砍成本沒有品質把關，只是把 API 帳單換成客訴帳單。

SOURCES · 參考資料

延伸閱讀

官方文件與公告

Anthropic：Prompt caching、Context management、Code execution with MCP、Tool Search
OpenAI：GPT-5.5 指南、GPT-5 Troubleshooting Guide
GitHub Blog：Copilot usage-based billing（2026/04/27）
GitHub Changelog：Auto model selection
Redis LangCache 文件、Repomix 文件

論文

RouteLLM（arXiv 2406.18665）· FrugalGPT（arXiv 2305.05176）
LLMLingua 系列（EMNLP'23、ACL'24）
中文 vibe coding token 成本（arXiv 2604.14210）
跨語言提示壓縮稽核（arXiv 2608.26175）

GitHub 專案

microsoft/LLMLingua · lm-sys/RouteLLM
oraios/serena · yamadashy/repomix
RTK（Rust Token Killer）· drona23/claude-token-efficient
sgl-project/sglang · vllm-project/vllm · TOON

文章與 YouTube

IntuitionLabs：Prompt caching、Batch API（2026/09）
KDnuggets、Composio、ComputingForGeeks、MindStudio、Analytics Vidhya
WorkOS、MCP Playground：MCP token 成本
MorphLLM：SGLang vs vLLM（2026/09）
Google Cloud：AI tokenomics 十一原則
olivomarco：Copilot token 最佳化指南
示範引用：Systima、Scott Spence、Aetherius、Zilliz、VS Code Blog
〈LLM 成本優化實戰指南〉17 篇一線文章彙整
YouTube：18 Claude Code Token Hacks in 18 Minutes

APPENDIX

附錄：26 個省 token 技巧詳解

每個技巧的原理、做法、效益數字與來源，以及同一任務四種設定的完整示範

六大成本優化策略

策略	原理	來源自述節省
Prompt Caching	快取命中的 token 只收約 10% 價格，agent 系統的首選	59–90%
語意快取	相似問題直接回傳快取答案，整個 LLM 呼叫都省下	約 73%
模型路由與小模型	60–70% 的 agent 呼叫用小模型就夠；微調小模型可再省	40–85%
Context 工程與 RAG 壓縮	只送相關片段、分層摘要、狀態外存	50–90%
Prompt 精簡與輸出限制	縮短指令、設 max_tokens；輸出比輸入貴 4–6 倍	20–70%
量測與監控	Helicone、Vantage 等工具找出成本熱點，是一切優化的前提	基礎建設

多招疊加常見 70–90% 總節省；各數字為來源自述，量測對象不同，不能相加 · 整理自〈LLM 成本優化實戰指南〉

MAP · 全貌

五層加補充、26 個 Tip

計價槓桿

01 Prompt Caching
02 Batch API
03 模型路由
04 推理與輸出預算
05 Copilot 改 token 計費

上下文工程

06 指令檔瘦身
07 /compact 與交接檔
08 Context editing
09 先規劃再動手
10 子代理分工

工具與 MCP

11 Tool Search
12 Code execution
13 RTK 輸出過濾
14 符號級檢索
15 Repomix 壓縮

資料壓縮

16 LLMLingua
17 TOON 格式
18 語言與 tokenizer
19 語意快取

自架與補充

20 Prefix caching
21 RAG 只送片段
22 規劃與執行分離
23 迴圈斷路器
24 Copilot 專屬設定
25 MCP 端過濾
26 交給確定性工具

外部 API：01–05、08、11–12 · Coding Agent 日常：06–07、09–10、13–15、22、24、26 · 內部/開源：03、19–21、23、25

檢查清單 · 個人

今天下班前就能做的三件事

1

改指令檔

刪到只剩每次都用得到的規則，加一行「只回 diff 與一句說明」

2

關掉用不到的工具

Configure Tools 只留這個專案真的會用的 MCP server

3

換設定就開新對話

換主題、換模型、換工具組時 Ctrl+N，不在長對話中途改

做完後，用同一句 Hello World 量一次前後差異

附錄 · 語意快取與 RAG 精準化

從「省 token」升級到「整通都不用打」

約 73%

語意快取

「今天天氣如何」和「現在天氣怎樣」命中同一筆快取，毫秒回應、完全跳過 LLM（Redis LangCache）

70–80%

Semantic Highlight

不塞整段 chunk，只送 chunk 內真正相關的句子，RAG token 大減且常提升答案品質（Zilliz）

−94%

階層式壓縮 + 狀態外存

文件先摘要再合併，5 份 × 800 token 只送 200 token；單請求 10,500 → 650 token

分層攔截

多層快取架構

精確比對（次毫秒）→ 語意快取 → session 狀態管理，逐層攔截重複流量（Redis）

來源：Redis LangCache、Zilliz、Ben-itzhak；整理自〈LLM 成本優化實戰指南〉· 對應附錄 Tip 16、19、21

附錄 · 推論端與價格趨勢

價格一直跌，為什麼還要優化架構

量化與高效注意力

8 / 4-bit 量化、Flash Attention、GQA / MLA，是自架推論省記憶體與算力的基本功

KV cache 是長上下文的戰場

沿 token 軸壓縮 KV 加淘汰策略，單節點 KV 容量從 1.2M 提升到 3.7M token

同效能，每年便宜約 50 倍

固定效能的推論價格每年下降 9–900 倍（中位數約 50 倍），但 agent 用量成長更快

監控是企業標配

追蹤每模型、每查詢成本，訂成本基準（例如分類每次低於 $0.01），每月檢視

來源：Hugging Face、Together AI、Epoch AI、Helicone、Vantage；整理自〈LLM 成本優化實戰指南〉· 對應附錄 Tip 20

TIP 03 · 計價槓桿 · 外部 + 內部模型

模型路由：簡單的事交給便宜的模型

原理

大多數請求不需要旗艦模型。路由器先判斷難度：簡單的派給小模型或內部開源模型，困難的才送旗艦，品質幾乎不變、成本大降。

怎麼做

- RouteLLM（lm-sys）：開源路由框架，可調成本門檻
- Cascade（FrugalGPT）：先跑便宜模型，評分不過才升級
- Coding agent：日常用 Sonnet / Haiku，難題才切 Opus
- 內部：格式化、摘要等雜務派給自架開源模型

效益

-85%

MT Bench 成本，同時保有 GPT-4 約 95% 的表現

來源：RouteLLM 論文（UC Berkeley / LMSYS）、FrugalGPT（arXiv 2305.05176）、OpenAI GPT-5 Troubleshooting Guide、KDnuggets

TIP 18 · 資料壓縮 · 任何模型

常駐內容用英文寫，對話照樣說中文

原理

多數 tokenizer 以英文為主訓練，同樣內容的中文常需要更多 token。但研究也發現「整段改用中文提示比較省」並不成立。

怎麼做

- 每輪都重送的系統提示、指令檔、工具描述用英文寫
- 一次性的對話與需求說明用中文即可
- 用目標模型的 tokenizer 實測（tiktoken、HF tokenizer）
- Qwen、GLM 等開源模型對中文較友善，要個別量測

實測

2.1×

同一篇文章的繁體中文 token 數是英文的 2.1 倍（一份多語部落格實測）

來源：Blake Crosley〈Tokenization Visualizer〉、arXiv 2604.14210、arXiv 2608.26175

TIP 24 · 補充 · GitHub Copilot

Copilot 專屬設定：模式、範圍、穩定

原理

Agent 模式會帶上工具清單、讀檔、多輪迴圈；簡單問題用不到這些。指令只在相關檔案載入，對話中保持設定不變，快取才不會失效。

怎麼做

- 簡單問題用 Ask 模式，多步驟任務才用 Agent 模式
- instructions 加 applyTo 路徑，只在符合的檔案載入
- 對話中不換模型、推理強度、工具組與順序；要換就開新的
- DOCX、PDF、PPTX 先用 MarkItDown 轉成 Markdown 再給

社群估計

60–90%

簡單問題改用 Ask 模式可省下的 token

來源：olivomarco/github-copilot-token-optimization（社群指南，非官方）、Microsoft MarkItDown

Status Line 設定步驟

在終端機底部，每一輪都看得到 context、token 與費用

```
ctx 13% | last turn: in 120 · cache_r 48000 · cache_w 2300 · out 845 | $0.46
```

手動設定共五步

1

確認 Node.js

腳本靠它執行

2

建立腳本

決定要顯示哪些數字

3

修改設定檔

讓 Claude Code 執行腳本

4

單獨測試

先確認腳本沒問題

5

在 CLI 確認

看輸入框下方的顯示

適用環境：Windows。腳本用 Node.js 寫，不需要安裝 jq。

步驟 1：確認有 Node.js

在終端機執行：

```
node --version
```

顯示版本號

可以繼續下一步

找不到指令

先安裝 Node.js，再重新開啟終端機

步驟 2：建立腳本

```
const fs = require("fs");
let data = {};
try {
  const raw = fs.readFileSync(0, "utf8").replace(/^/, "");
  data = JSON.parse(raw);
} catch {}

const ctx = data.context_window ?? {};
const u = ctx.current_usage ?? {};
const pct = Math.round(ctx.used_percentage ?? 0);
const cost = (data.cost?.total_cost_usd ?? 0).toFixed(2);

console.log(
  `ctx ${pct}% | last turn: in ${u.input_tokens ?? 0}` +
    ` · cache_r ${u.cache_read_input_tokens ?? 0}` +
    ` · cache_w ${u.cache_creation_input_tokens ?? 0}` +
    ` · out ${u.output_tokens ?? 0} | $${cost}`
);
```

檔案位置

`C:\Users\<你的帳號>\.claude\statusline.js`

運作方式

- 從 stdin 讀入狀態 JSON
- 印出的文字就是 status line
- JSON 解析失敗時顯示 0，不會出錯

步驟 3：修改設定檔

打開 C:\Users\<你的帳號>\.claude\settings.json，在最外層加入：

```
{
  "statusLine": {
    "type": "command",
    "command": "node C:/Users/<你的帳號>/.claude/statusline.js"
  }
}
```

換成你的帳號
路徑要對應你的電腦

路徑用正斜線
不必處理 JSON 的反斜線跳脫

保留原有設定
前一個項目的結尾要補逗號

步驟 4：單獨測試腳本

不必啟動 Claude Code，直接餵一段 JSON：

```
echo '{"context_window":{"used_percentage":12}}' |
node C:/Users/<你的帳號>/.claude/statusline.js
```

應該會看到：

```
ctx 12% | last turn: in 0 · cache_r 0 · cache_w 0 · out 0 | $0.00
```

步驟 5：在 CLI 確認

1. 開一個獨立的終端機視窗，進入專案資料夾
2. 執行 claude
3. 出現信任提示時，選 Yes, I trust this folder
4. 送出一則訊息，看輸入框下方

示範前先跑一次

信任提示的預設選項是 No, exit。先處理掉，現場就不會卡在這裡。

比較快的做法：/statusline

在 claude 裡用文字描述想看什麼，讓它產生腳本和設定：

```
/statusline 顯示 context 使用率、上一輪的 input、cache read、cache write、output token，以及累計費用
```

要在終端機的 claude 裡執行。做完一樣用步驟 5 確認。

常見問題

狀況	怎麼處理
完全沒有出現這一行	檢查 settings.json 的逗號和腳本路徑，再重新啟動 claude
送出訊息前全是 0	正常，第一次呼叫 API 後才有數字
送出訊息後仍是 0	可能是欄位名稱對不上，要看實際收到的 JSON
要看實際的 JSON	暫時把 command 改成 cat > C:/Users/<你的帳號>/sl.json，看完改回來
想移除	刪掉 settings.json 裡的 statusLine 區塊